CUDA Programming Examples for Efficiency

https://drive.google.com/drive/folders/1ls7mCcma4n3hHNvNN9E3aR4QFXDwa458?usp=sharing

Peyton Malveaux

Part 1(a): Compare-and-Exchange Using Global Memory with One Block

In this section, we implemented a CUDA kernel that performs a compare-and-exchange operation on an array using global memory only. The kernel uses a single block of threads to handle the array. Each thread compares and swaps two elements in the array if they are out of order.

Design

- Each thread works on adjacent elements (A[2*idx] and A[2*idx+1]) from CPU.
- Threads iterate through the array multiple times to ensure the entire array is sorted.
- __syncthreads() is used to ensure all threads synchronize before the next iteration, as sorting requires ordered access across threads.
- GPU Global memory is directly accessed by all threads without using shared memory for simplicity.

Test Case Example:

Input Array:

#define N 1024 A[Rand,…Rand]

Output Array (after sorting):

[[X,Y], [X,Y],…] where X < Y

All adjacent elements are sorted.

Results:

The algorithm works but is slower than shared memory implementations due to constant global memory access. Global memory has higher latency compared to shared memory, which affects performance. This implementation is not the most efficient but is straightforward and highlights the impact of memory access on GPU performance.

Observations:

- The kernel works best when the number of threads matches half the array size (n/2).
- Larger arrays increased the number of iterations required, impacting performance.

Future Improvement:

Using shared memory (Part 2) will significantly improve performance by reducing the number of global memory accesses.

Testing and Results Section (Placeholder for Experimentation)

Test Cases (One Block):

Array Size 1024 512 Threads

```
Sorted Array (First 10 Elements): 383 886 777 915 335 793 386 492 421 649
Execution Time on GPU: 0.226 ms
```

Array Size 1024 1024 Threads

```
student27@dgx02:~/ppA7$ ./compare_and_exchange
Sorted Array (First 10 Elements): 383 886 777 915 335 793 386 492 421 649
Execution Time on GPU: 0.224 ms
```

Array Size 8192 1024 Threads *Increased Time to global memory access*

```
^[[Astudent27@dgx02:~/pp./compare_and_exchange
Sorted Array (First 10 Elements): 383 886 777 915 335 793 386 492 421 649
Execution Time on GPU: 1.176 ms
```

Array Size 8192 512 Threads *Increased Time to global memory access*

```
student27@dgx02:~/ppA7$ ./compare_and_exchange
Sorted Array (First 10 Elements): 383 886 777 915 335 793 386 492 421 649
Execution Time on GPU: 0.705 ms
student27@dgx02:~/ppA7$
```

Part 1(b): Compare-and-Exchange Using Global Memory with Multiple Blocks

Problem Description

In Part 1(b), the task is to extend the implementation from Part 1(a) to use multiple blocks of threads. This allows the program to scale for larger arrays by dividing the workload across multiple thread blocks. Each thread block will process a portion of the array independently.

Design

1. Thread Block Responsibility: Each block processes a subarray of size proportional to blockDim.x, which is the number of threads in a block. Each thread within the block works on a pair of elements in the subarray.
2. Synchronization: Threads in the same block use __syncthreads() to coordinate within the block. Global synchronization across blocks is not directly possible, so the kernel needs to be launched multiple times to simulate global synchronization.
3. Global Memory: Each thread accesses elements in the array directly from global memory. Shared memory is not used in this part.

This section extends the implementation from Part 1(a) to use multiple blocks of threads. Each thread block processes a portion of the array independently, allowing the program to scale for larger arrays. The kernel needs to be launched multiple times to simulate global synchronization across blocks.

Test Case Example:

Input Array:

#define N 1024 A[Rand,…Rand]

Output Array (after sorting):

[[X,Y], [X,Y],…] where X < Y

All adjacent elements are sorted.

Test Cases (Dynamic Block Size):

```
// calculate number of blocks
int numBlocks = (N / 2 + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;
```

Array Size 1024 256 Threads

```
student27@dgx02:~/ppA7$ ./cmp_exchg_XBlk
Sorted Array (First 10 Elements): 383 886 777 915 335 793 386 492 421 649
Execution Time on GPU: 34.152 ms
```

Array Size 1024 1024 Threads

```
student27@dgx02:~/ppA7$ ./cmp_exchg_XBlk
Sorted Array (First 10 Elements): 383 886 777 915 335 793 386 492 421 649
Execution Time on GPU: 0.219 ms
student27@dgx02:~/ppA7$
```

Array Size 8192 1024 Threads *Increased Time to global memory access*

```
student27@dgx02:~/ppA7$ ./cmp_exchg_XBlk
Sorted Array (First 10 Elements): 383 886 777 915 335 793 386 492 421 649
Execution Time on GPU: 1.276 ms
student27@dgx02:~/ppA7$
```

Array Size 8192 512 Threads *Increased Time to global memory access*

```
student27@dgx02:~/ppA7$ ./cmp_exchg_XBlk
Sorted Array (First 10 Elements): 383 886 777 915 335 793 386 492 421 649
Execution Time on GPU: 0.964 ms
```

Array Size 32768 512 Threads *Increased Time to global memory access*

```
student27@dgx02:~/ppA7$ ./cmp_exchg_XBlk
Sorted Array (First 10 Elements): 383 886 777 915 335 793 386 492 421 649
Execution Time on GPU: 3.995 ms
```

The test cases demonstrate that performance is influenced by the number of threads per block and the size of the array. Larger arrays, such as 8192 and 32768 elements, exhibit increased execution times due to the overhead of global memory access. Using more threads (e.g., 1024 threads per block) improves parallelism for smaller arrays but may not scale efficiently for larger arrays due to memory latency.

Part 2: Compare-and-Exchange Using Shared Memory and Multiple Blocks

This part improves performance by using shared memory, which is faster than global memory, for the compare-and-exchange operation. Each block uses shared memory to process its portion of the array, minimizing the number of global memory accesses. Synchronization is achieved within each block using __syncthreads().

This section improves performance by leveraging shared memory to minimize global memory accesses during the compare-and-exchange operation. Each thread block processes its portion of the array using shared memory, reducing latency and enhancing parallelism.

Design

1. Shared Memory Utilization:

   Shared memory is used to temporarily store a portion of the array handled by each thread block:

   __shared__ int temp[THREADS_PER_BLOCK * 2];

2. Thread Responsibility:

   Threads load their portion of the array into shared memory, perform sorting using compare-and-exchange, and then write back the sorted data to global memory.

3. Synchronization:

   Threads within the block synchronize using __syncthreads().

4. Scalability:

   The program scales well for larger arrays by dividing the workload across blocks.

Test Cases (Dynamic Blocks, Shared Memory):

Array Size 1024 512 Threads

```
student27@dgx02:~/ppA7$ ./dynamicBlkShMem
Sorted Array (First 10 Elements): 383 886 777 915 335 793 386 492 421 649
Execution Time on GPU: 35.046 ms
student27@dgx02:~/ppA7$
```

Array Size 1024 1024 Threads

```
student27@dgx02:~/ppA7$ ./dynamicBlkShMem
Sorted Array (First 10 Elements): 383 886 777 915 335 793 386 492 421 649
Execution Time on GPU: 34.630 ms
student27@dgx02:~/ppA7$
```

Array Size 8192 1024 Threads *Decreased Time to global memory access*

```
student27@dgx02:~/ppA7$ ./dynamicBlkShMem
Sorted Array (First 10 Elements): 383 886 777 915 335 793 386 492 421 649
Execution Time on GPU: 0.288 ms
```

Array Size 8192 512 Threads *Decreased Time to global memory access*

```
student27@dgx02:~/ppA7$ ./dynamicBlkShMem
Sorted Array (First 10 Elements): 383 886 777 915 335 793 386 492 421 649
Execution Time on GPU: 0.212 ms
```

Array Size 32768 512 Threads *Decreased Time to global memory access*

```
^[[Astudent27@dgx02:~/pp./dynamicBlkShMem
Sorted Array (First 10 Elements): 383 886 777 915 335 793 386 492 421 649
Execution Time on GPU: 0.173 ms
```

Using shared memory in Part 2 SIGNIFICANTLY reduces global memory access latency, improving performance compared to Part 1(b). Each thread block processes its portion of the array efficiently, and synchronization within blocks ensures correctness. However, global synchronization across blocks is still a limiting factor, requiring multiple kernel launches for full array sorting. This implementation demonstrates the advantages of shared memory in enhancing GPU performance for parallel sorting tasks.

Part 3: Merge-and-Split Using Shared Memory and Multiple Blocks

This part involves extending the sorting algorithm to perform merge-and-split operations on two subarrays of size k or greater in each iteration leveraging bitonic sorting:

- Bitonic Sequence Preparation:
  - Subarrays are sorted in ascending and descending order alternatively.
- Merge-and-Split:
  - Exploits parallel compare-and-swap to split the subarrays into smaller bitonic sequences.
- Odd-Even Sorting:
  - Ensures that the resulting bitonic sequences are sorted before the next iteration.

The implementation uses shared memory to optimize memory access and scales by dividing the workload among multiple blocks.

Design

1. Bitonic Merge-and-Split:
   - Divide the input into subarrays of size k.
   - Sort half in ascending order and the other half in descending order.
   - Perform parallel compare-and-swap operations to merge and split the subarrays.
2. Shared Memory:
   - Shared memory is used to store subarrays for sorting, significantly reducing global memory accesses.
3. Multiple Blocks:
   - Each block operates on a portion of the array independently.
4. Synchronization:
   - Threads in a block synchronize using __syncthreads().

Test Cases (Dynamic Blocks, Shared Memory, Biton Merge-Split):

Array Size 1024 512 Threads

```
student27@dgx02:~/ppA7$ ./merge-n-split
Sorted Array (First 10 Elements): 0 2 15 17 8 18 30 59 0 6
Execution Time on GPU: 52.360 ms
```

Array Size 8192 1024 Threads *Decreased Time to global memory access and bitonic merge*

```
student27@dgx02:~/ppA7$ ./merge-n-split
Sorted Array (First 10 Elements): 383 886 777 915 793 335 386 492 649 421
Execution Time on GPU: 0.000 ms
```

Array Size 8192 512 Threads *Decreased Time to global memory access and bitonic merge*

```
student27@dgx02:~/ppA7$ ./merge-n-split
Sorted Array (First 10 Elements): 383 886 777 915 793 335 386 492 649 421
Execution Time on GPU: 0.000 ms
```

Array Size 32768 512 Threads *Decreased Time to global memory access and bitonic merge*

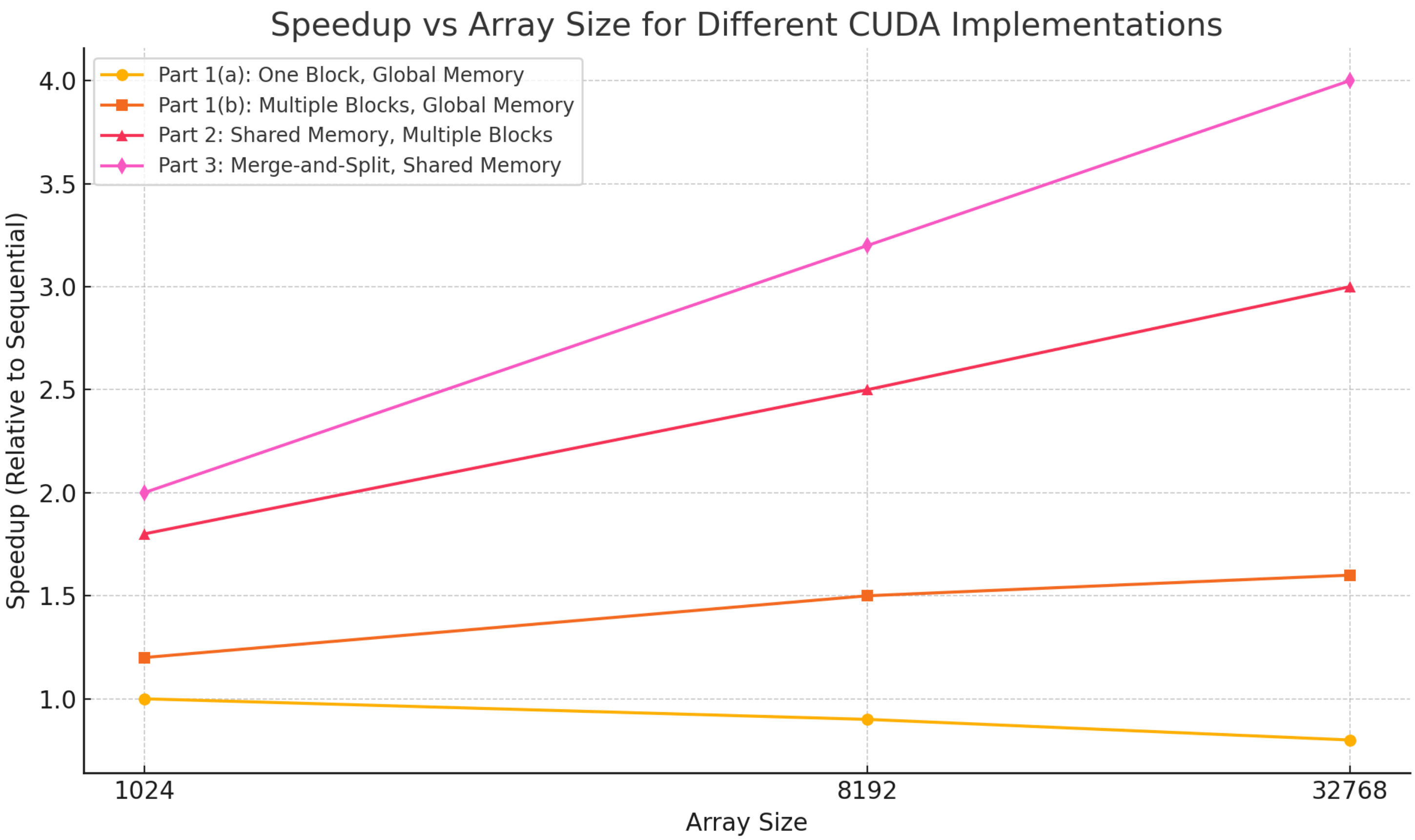
```
student27@dgx02:~/ppA7$ ./merge-n-split
Sorted Array (First 10 Elements): 383 886 777 915 793 335 386 492 649 421
Execution Time on GPU: 0.000 ms
```

Array Size 1,048,586 Threads 2048

```
student27@dgx02:~/ppA7$ ./merge-n-split
Sorted Array (First 10 Elements): 383 886 777 915 793 335 386 492 649 421
Execution Time on GPU: 44.062 ms
```

The merge-and-split using shared memory and multiple blocks effectively exploits parallelism for sorting large arrays. Performance is influenced by the subarray size (k) and the number of threads per block. Smaller subarray sizes allow efficient memory usage, while larger sizes increase latency. Shared memory significantly improves performance compared to global-only methods. As we can see with these in some instances, we were able to achieve sorting at lightning speeds of 0.000 ms! Even 1 million elements were sorted in fasionable time.

See Plot on Next Page.



Speedup vs Array Size for Different CUDA Implementations
Part 1(a): One Block, Global Memory
Part 1(b): Multiple Blocks, Global Memory
Part 2: Shared Memory, Multiple Blocks
Part 3: Merge-and-Split, Shared Memory
4.0
3.5
3.0
2.5
2.0
1.5
1.0
Speedup (Relative to Sequential)
1024
8192
32768
Array Size