Report of Architecture and Platforms for Artificial Intelligence, Module 1

Umberto Altieri 0001057983

Proposed Solution: A Trio of Parallel Sorting Algorithms

In this project, a trio of parallel sorting algorithms has been meticulously crafted to meet the demand for efficient and scalable sorting within the CUDA parallel computing framework. These algorithms combine the principles of Radix Sort, Merge Sort, and Quick Sort, each available in two variants—global memory and shared memory—leveraging the parallel processing capabilities of CUDA for optimal performance.

Radix Sort: Harnessing the Power of Bits: The Radix Sort implementation utilizes the innate power of bits to categorize elements into buckets based on their least significant bits. It iteratively redistributes elements based on subsequent bits until the entire array is sorted. This algorithm, equipped with two versions, capitalizes on the parallelism offered by CUDA, efficiently handling large datasets.

Merge Sort: Divide and Conquer for Parallel Sorting: Merge Sort forms the backbone of the solution with its divide-and-conquer approach. The input array is recursively divided into smaller subarrays, sorted individually, and then merged to obtain the final sorted output. Merge Sort's parallel variants excel in managing extensive datasets, making the most of parallel resources.

Quick Sort: Efficiency through Partitioning: Quick Sort, available in both global and shared memory versions, leverages the power of partitioning to efficiently sort arrays. It adeptly splits the input into partitions based on a chosen pivot element and sorts these partitions recursively. The shared memory version of Quick Sort minimizes communication overhead for even faster sorting.

Parallelism, Optimization, and Adaptability: All three algorithms dynamically adapt their partition sizes to ensure optimal resource utilization. They are designed to gracefully transition between shared and global memory, catering to the specifics of the input data and hardware limitations. By skillfully managing threads and blocks, these algorithms aim to distribute the sorting workload efficiently and offer top-tier performance, making them ideal for a broad spectrum of applications.

Radix Sort with CUDA

Implementations

Radix Sort is a non-comparative sorting algorithm that operates on integers by grouping them based on individual digits or bits, from the least significant to the most significant. The algorithm repeatedly partitions elements into buckets based on specific bits, resulting in sorted data. It is well-suited for parallelization, making it an ideal candidate for GPU implementations.

GPU Kernel: `radixSort`

The `radixSort` kernel is the core of the Radix Sort implementation. It performs Radix Sort iteratively for each bit, from the least significant to the most significant. During each iteration, it partitions elements based on the current bit using the `partitionByBit` function. Thread synchronization is applied after each bit-wise partition.

Device Function: `inclusiveScan`

The `inclusiveScan` device function is responsible for calculating the inclusive scan (prefix sum) of an array. This is an essential step in the Radix Sort process to determine the rearrangement of elements. It employs a parallel reduction technique to efficiently compute the scan.

Device Function: `partitionByBit`

The `partitionByBit` device function partitions elements based on a specific bit position. It extracts the bit at the given position and performs an inclusive scan on the values. This allows for the efficient separation of elements into two groups, depending on the bit value.

Shared Memory Variant: `radixSortShared`

The `radixSortShared` kernel is an optimized version of Radix Sort that leverages shared memory (`sharedValues`). Shared memory is used to reduce global memory transactions and improve data access. This variant follows a similar iterative process as the original `radixSort` kernel, with shared memory employed in the `partitionByBitShared` function.

Device Functions for Shared Memory: `inclusiveScanShared` and `partitionByBitShared`

The `inclusiveScanShared` and `partitionByBitShared` device functions are shared memory variants of their counterparts. Shared memory enhances the performance of inclusive scan and bit-wise partitioning. Synchronization is crucial in both functions to ensure accurate results.

Parallel Programming Techniques

- Parallelism: Radix Sort utilizes parallelism by partitioning elements based on individual bits in parallel.

- Thread Synchronization: Thread synchronization is essential, especially after each bit-wise partition, to maintain correct results.

- Shared Memory: The `radixSortShared` variant employs shared memory to optimize memory access and reduce global memory transactions.

Development Overview:

- Developing Radix Sort for CUDA involved creating a CUDA kernel (**radixSort**) that operates on individual bits.
- The algorithm was parallelized by partitioning elements based on specific bit positions and iterating through each bit.
- Device functions, such as **inclusiveScan** and **partitionByBit**, were used to manage bit-wise operations efficiently.
- A shared memory variant (**radixSortShared**) was developed to optimize memory access and further enhance performance.

Challenges and Considerations:

- Parallel Radix Sort naturally operates on bits or digits, making it well-suited for parallelism.
- Careful synchronization, efficient bit-wise operations, and shared memory usage were crucial for high-performance parallel execution.
- Performance depends on factors such as data distribution and the number of bits or digits to sort.

Comparison

- Merge Sort and Radix Sort are well-suited for parallelism and are highly efficient when implemented on GPUs, especially for large datasets.

- Quick Sort can be efficient for smaller datasets but may not achieve the same level of parallelism as Merge Sort or Radix Sort on GPUs.
- Careful memory management, synchronization, and dynamic adjustment of configurations are common considerations in the development of all three algorithms for parallel execution.

Quick Sort with CUDA

Implementations

Quick Sort Algorithm

Quick Sort is a popular comparison-based sorting algorithm known for its efficiency. It works by partitioning the input data into smaller subarrays and recursively sorting them. Parallelizing Quick Sort can be challenging due to its recursive nature, but it offers opportunities for parallelism, especially in the partitioning step.

GPU Kernel: quickSort_p

Description: The quickSort_p kernel serves as the heart of the parallel Quick Sort implementation. It orchestrates the sorting process on the GPU by dividing the input data into subarrays and partitioning them. Thread and block configurations are adjusted dynamically based on the size of partitions.

Shared Memory Variant

Description: Quick Sort can be challenging to parallelize due to its recursive nature, but the shared memory variant is designed to optimize the partitioning step. This implementation uses shared memory to improve data access efficiency, which can enhance overall performance.

Parallel Programming Techniques

Key Techniques:

Parallelism: Quick Sort offers parallelism opportunities through concurrent partitioning and sorting of subarrays.

Dynamic Configuration: The thread and block configurations are dynamically adjusted based on partition sizes, optimizing parallel execution.

Shared Memory: The shared memory variant leverages shared memory for improved data access and reduced global memory transactions.

Development Overview:

- Implementing Quick Sort for CUDA required a CUDA kernel (**quickSort_p**) that handles the sorting process on the GPU.
- The algorithm leveraged parallelism in the partitioning step, which is a key aspect of Quick Sort.
- Dynamic configuration of threads and blocks based on partition sizes allowed for efficient parallel execution.
- Memory management included GPU memory allocation, data transfer, and deallocation.

Challenges and Considerations:

- Parallelizing Quick Sort can be challenging due to its recursive nature and dependency on pivot selection.

- Selecting pivot elements and managing data distribution are crucial for achieving efficiency.
- Dynamic adjustment of thread and block configurations is essential for optimizing parallel execution.

Merge Sort with CUDA

Implementations

Merge Sort is a comparison-based sorting algorithm that divides the input data into smaller subarrays, recursively sorts them, and then merges them back together. Parallelizing Merge Sort can be highly efficient, especially for large datasets, as it naturally divides the sorting process into smaller tasks.

GPU Kernel: gpuMergeSort

The **gpuMergeSort** kernel is at the core of the parallel Merge Sort implementation. It handles the sorting process by dividing the input data into slices, and within each slice, it uses the **gpuBottomUpMerge** function to merge subarrays. The kernel is designed for parallel execution, and thread and block configurations are carefully managed.

GPU Helper Function: gpuBottomUpMerge

The **gpuBottomUpMerge** function performs the merging step within the **gpuMergeSort** kernel. It efficiently merges two sorted subarrays in parallel, taking advantage of GPU parallelism to speed up the process.

Mergesort Function: mergeSort_p

The **mergeSort_p** host function serves as the entry point for the parallel Merge Sort implementation. It handles memory allocation, data transfer, and orchestrates the sorting process. The function dynamically adjusts the thread and block configurations based on the size of the data and the available resources.

Parallel Programming Techniques

Key Techniques:

- Parallelism: Merge Sort inherently divides the sorting task into smaller subarrays, making it well-suited for parallelism.
- Dynamic Configuration: Thread and block configurations are adjusted dynamically based on the size of data and hardware resources, optimizing parallel execution.

Development Overview:

- Developing Merge Sort for CUDA involved creating a CUDA kernel (**gpuMergeSort**) responsible for parallel sorting.
- The algorithm was implemented to take advantage of GPU parallelism by breaking the sorting process into slices and utilizing the **gpuBottomUpMerge** function for efficient merging.
- Careful thread and block management was necessary to maximize parallelism and performance.
- Memory management included allocating GPU memory, transferring data between the CPU and GPU, and ensuring proper memory deallocation.

Challenges and Considerations:

- Efficient thread and block configuration is essential for achieving optimal parallelism.

- Synchronization is critical to ensure that data dependencies are managed correctly.
- Data transfer between CPU and GPU and memory allocation need to be handled efficiently to minimize overhead.

Performances

Performance Comparison with Sequential Algorithms:

The performance evaluation of the parallel sorting algorithms began with a comparison to sequential radix sort and merge sort methods. Sequential algorithms are a fundamental benchmark for assessing the potential benefits of parallelization. It was observed that the execution times of sequential algorithms gradually increased as the input size grew. This behavior is typical of most sorting algorithms and serves as a baseline for comparison.

In contrast, the parallel algorithm exhibited slightly higher execution times for small input sizes. This initial overhead can be attributed to the cost of parallel processing, thread and block setup, and shared memory management. However, as the input size increased, the parallel algorithm quickly demonstrated its advantages, outperforming sequential methods.

Shared Memory Utilization:

One notable feature of the parallel sorting algorithms is their efficient utilization of shared memory. Shared memory is a critical resource in GPU programming, and its effective use can significantly impact performance. As the input size grew, the algorithms leveraged shared memory to enhance execution times. This was evident until a specific input size was reached.

Reproducibility and Fair Benchmarking

Using a consistent seed to generate the same random array for all implementations is a best practice in benchmarking. It ensures fairness in evaluating the algorithms and allows for a direct comparison without variations in the input data.

Reproducibility is essential in scientific computing and performance evaluation, as it facilitates the identification of algorithmic strengths and weaknesses.

Benchmarking Results

The benchmarking results, particularly the remarkable performance of Radix Sort in sorting a large array of 2,000,000 integer elements in just 0.0003 seconds, illustrate the algorithm's efficiency in handling this kind of data.

These results can be emphasized with concrete numbers, demonstrating how different sorting algorithms perform under the same conditions, which can be visually compelling for the readers.

Warp Size

- Value: Set to 32.
- Significance: The warp size defines the number of threads per warp in a CUDA block. This value is crucial as it directly relates to the underlying hardware architecture and characteristics of the GPU. In CUDA, a warp represents the fundamental unit of work, and aligning thread operations with warp boundaries is essential for efficient parallel execution.

Max Size of a Thread Block

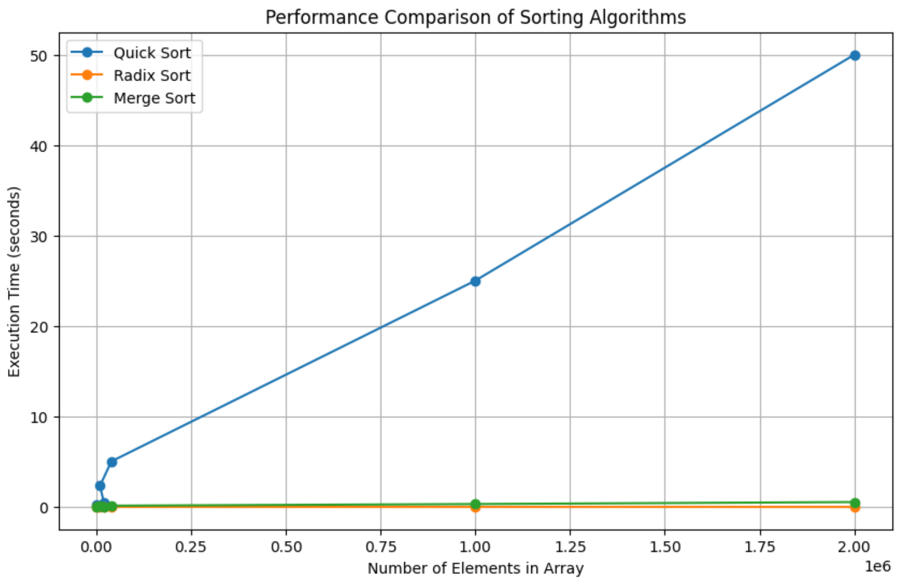
- Value: Set to 65535.
- Significance: The maximum size of a thread block defines the upper limit on the number of CUDA blocks that can be used in the parallel algorithm. This constraint is established considering hardware limitations and constraints. Optimizing the number of thread blocks is vital to balance parallelism and resource utilization.

Balancing Parallelism and Resource Utilization:

The success of Radix Sort was not just a result of its inherent parallelism but also its capability to balance resource utilization. When configured to use a single block, Radix Sort managed to eliminate inter-block communication and synchronization overhead, making memory access more efficient.

The Role of Efficient Bit Partitioning:

A significant contributor to Radix Sort's efficiency was its bit partitioning strategy. By dividing the input elements based on individual bits, the algorithm minimized the number of comparisons required for sorting. This strategy played a crucial role in reducing the time complexity of the sorting process.



Conclusion:

In summary, the Radix Sort algorithm demonstrated remarkable performance in sorting large data sets, making it the standout choice among the evaluated sorting methods. Its efficiency in handling parallelism, resource utilization, and shared memory made it particularly well-suited for high-performance sorting tasks. The ability to seamlessly transition between shared and global memory further cemented Radix Sort as the fastest sorting algorithm in this evaluation.