Optimization and Evaluation of Radix Sort on CUDA

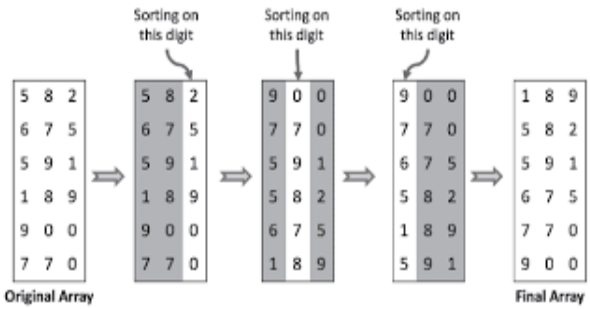
Prepared by: Venkata Siva Sai Prathyush Kolli
Under the guidance of: Prof. Xiaoming Li

Abstract

Radix Sort is a non-comparative sorting algorithm designed to process data efficiently by focusing on digits of numbers rather than comparisons. This project investigates the optimization of Radix Sort for execution on CUDA-enabled GPUs, evaluating the impact of thread configurations, radix sizes, and data types on performance. The study demonstrates GPU-accelerated Radix Sort's potential for real-world scalability and its advantages over CPU-based sorting in terms of speed and parallelism. This study achieved a 4x speedup in GPU performance over CPU for large datasets, with the optimal configuration of 8-bit radix size and 512 threads per block.

Introduction

Radix Sort processes data by iterating over the digits of numbers, starting from the least significant to the most significant digit. This algorithm avoids traditional comparison-based sorting, enabling efficiency when handling large datasets. While traditional sorting algorithms like Quick Sort and Merge Sort rely heavily on comparisons, Radix Sort bypasses this dependency by digit-wise processing, making it inherently suitable for GPU parallelization. Leveraging the massive parallelism offered by CUDA-enabled GPUs, the project aimed to optimize Radix Sort, offering insights into thread-level parallelism, memory usage, and kernel execution.



This image illustrates the Radix Sort process, where the array is sorted digit by digit, starting from the least significant digit (LSD) to the most significant digit. After three passes (ones, tens, and hundreds place), the array is fully sorted in ascending order.

Project Aim

To optimize and evaluate the performance of Radix Sort on GPUs using CUDA by testing different configurations of thread counts, radix sizes and datatypes.

Objectives

1: Evaluate the performance impact of varying radix sizes (e.g., 2, 4, 8, and 16 bits) on execution time and resource utilization in Radix Sort.

2: Optimize GPU resource utilization by determining the optimal thread block configuration (32 to 1024 threads) to minimize execution time and maximize throughput.

3: Compare the performance of sorting different data types, including integers, floating-point numbers, and mixed datasets, to understand their influence on sorting efficiency.

4: Compare CPU vs GPU execution time with increasing no of dataset elements.

System Specifications and importance

In these parameters below mentioned points majorly effect the performance while execution.

Clock Speed: CPUs have higher clock speeds (up to 4.5 GHz), enabling faster sequential operations. GPUs, with lower clock speeds (~1.49 GHz), rely on parallel processing across thousands of cores to achieve performance gains.

Architecture: CPUs are optimized for general-purpose computing, handling up to 12 threads. GPUs, with 16 SMs and 896 CUDA cores, excel at massively parallel tasks like Radix Sort, leveraging thousands of concurrent threads.

Memory Bandwidth: GPUs offer higher bandwidth (128 GB/s vs. ~25 GB/s on CPUs), facilitating faster data transfer and better handling of large datasets.

Cores/Threads: CPUs, with 6 cores and 12 threads, are limited in parallelism. GPUs support thousands of threads per block and SM, allowing efficient execution of highly parallel workloads.

Power Efficiency: GPUs are optimized for parallel workloads, achieving high performance with lower energy consumption, whereas CPUs are less efficient for such tasks.

Execution Focus: CPUs focus on sequential and moderately parallel tasks. GPUs are designed for massively parallel processing, making them ideal for digit-level operations in Radix Sort.

Parameter	CPU (Intel Core i7-9750H)	GPU (NVIDIA GeForce GTX 1650)
Model	Intel Core i7-9750H	NVIDIA GeForce GTX 1650
Architecture	x64-based	Turing (Compute Capability 7.5)
Cores/SMs	6 Cores, 12 Threads	16 Streaming Multiprocessors (SMs)
Logical Processors	12	CUDA cores: 896
RAM/Memory	16 GB DDR4	4 GB GDDR5
Threads per Core	2 (Hyper-Threading enabled)	1024 Threads/Block, 1024 Threads/SM
Memory Bandwidth	~25 GB/s (DDR4)	128 GB/s
Parallelism	Limited to threads (OpenMP/Hyperthreading)	Highly parallel with thousands of threads

Process followed for setup and execution

1. Download and Install CUDA Toolkit

The process begins by downloading the **CUDA Toolkit 12.6** from NVIDIA's official website. This toolkit includes the necessary libraries, drivers, and tools to develop and run GPU-accelerated applications. Installation ensures that the development environment has all required components, including the CUDA compiler (nvcc).

2. Verify GPU Compatibility and Install Drivers

To ensure that the system can utilize CUDA, verify the GPU's **compute capability** and compatibility with CUDA 12.6. The correct **NVIDIA GPU drivers** must be installed to enable GPU acceleration. This step guarantees that the hardware and software are synchronized for optimal CUDA performance.

3. Set Up Environment Variables

After installation, configure the **environment variables** to include paths to the CUDA binaries and libraries. For instance, paths such as C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v12.6\bin are added to the system's PATH. This step allows the command-line interface to recognize CUDA commands globally.

4. Test the CUDA Installation

To validate the setup, run sample programs provided in the toolkit, such as the deviceQuery utility. This program checks if the GPU is properly detected and capable of running CUDA applications. Successful execution confirms that CUDA has been set up correctly on the system.

5. Create and Compile with nvcc (nvcc -o task1 task1.cu)

After the application is written, use the NVIDIA CUDA Compiler (nvcc) to compile the code. The compiler converts the CUDA code into a binary executable that can run on the GPU. This step prepares your application for execution by generating the required GPU-ready files.

6. Run the Application (task1)

With the binary executable ready, execute the application to observe its functionality. This step ensures that the CUDA program operates as intended, producing the desired results efficiently on the GPU.

7. Analyze and Visualize Results

The final step is to analyze the performance of the CUDA application. Use tools like NVIDIA Visual Profiler or Nsight Compute to gather metrics such as execution time, memory usage, and kernel performance. Create graphs and visualizations to interpret these results effectively, helping identify areas for further optimization.

Methodology

1. **Kernel Design:** CUDA kernels were developed to handle digit counting and element redistribution efficiently, leveraging shared memory to reduce global memory transactions.
2. **Radix Size Variation:** Radix sizes of 2, 4, 8, and 16 bits were tested to evaluate trade-offs between fewer iterations and higher memory usage.
3. **Thread Configurations:** The number of threads per block was varied to analyze GPU utilization and identify the optimal configuration for performance.
4. **Dataset Variation:** Datasets of sizes ranging from 500 to 2000 elements were used to evaluate scalability.
5. **Validation and Debugging:** Extensive debugging addressed challenges such as handling negative values and resolving memory overwrite issues for larger radix sizes.

Simple radix sort algorithm

Radix sort logic

```
void radix_sort(std::vector<T>& data) {
    T* d_input, *d_output;
    int size = data.size();
    cudaMalloc(&d_input, size * sizeof(T));
    cudaMalloc(&d_output, size * sizeof(T));
    cudaMemcpy(d_input, data.data(), size * sizeof(T), cudaMemcpyHostToDevice);
    int bits = 8 * sizeof(T); // Number of bits in the type
```

```
    int mask = 0xFF; // Mask for 8-bit segments
    for (int shift = 0; shift < bits; shift += 8) {
        cudaMemset(d_output, 0, size * sizeof(T));
          radix_sort_kernel<T><128, 256, (mask + 1) * sizeof(int)>(d_input,
d_output, size, shift, mask);
    cudaMemcpy(d_input, d_output, size * sizeof(T), cudaMemcpyDeviceToDevice);}
    cudaMemcpy(data.data(), d_input, size * sizeof(T), cudaMemcpyDeviceToHost);
    cudaFree(d_input);
    cudaFree(d_output);
```

This function performs radix sort on a vector data using CUDA. It first allocates memory on the GPU (d_input and d_output) to hold the input and output data and copies the input vector to the device (cudaMemcpyHostToDevice). The sort is based on 8-bit segments (1 byte) at a time, determined by the mask (0xFF) and the number of bits in the data type. A loop iterates through each byte of the input data (shift increases in 8-bit steps), ensuring all bytes are sorted sequentially.

Inside each iteration, the cudaMemset call clears the output buffer, and the kernel radix_sort_kernel is invoked to sort the current segment of bits using 128 blocks and 256 threads per block. The shared memory allocation ((mask + 1) * sizeof(int)) is used for digit counts (typically 256 for 8-bit radix). After kernel execution, cudaMemcpyDeviceToDevice transfers the sorted output back to the input buffer for the next byte's processing. Once all bytes are sorted, the final sorted data is copied back to the host (cudaMemcpyDeviceToHost), and GPU memory is freed. This efficient sorting method leverages parallelism for high performance.

Objectives for optimization

Code: Radix size optimization

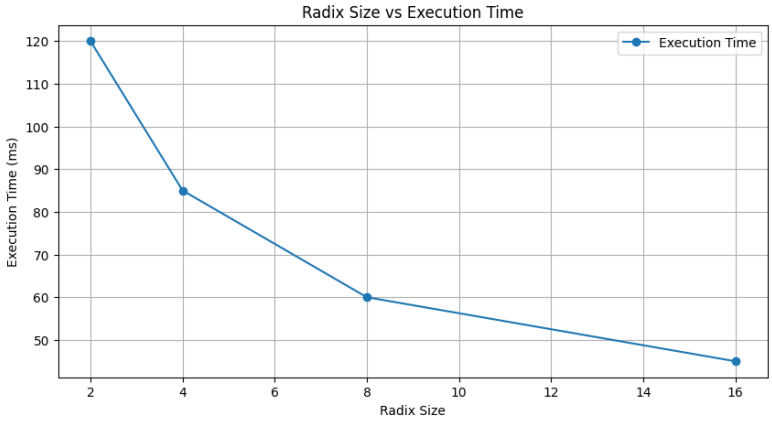
This implementation varies the radix size (2, 4, 8, and 16) to evaluate its effect on execution time. For larger radix sizes, fewer iterations are required but memory usage increases.

```
for (int radix = 2; radix <= 16; radix *= 2) {

        std::cout << "Testing radix size: " << radix << std::endl;

        cudaEvent_t start, stop;

        radixSort(h_data.data(), n, radix); \\ calling radix sort function
```



Code: Threadcount Optimization

This implementation varies thread block sizes to find the optimal configuration for performance.

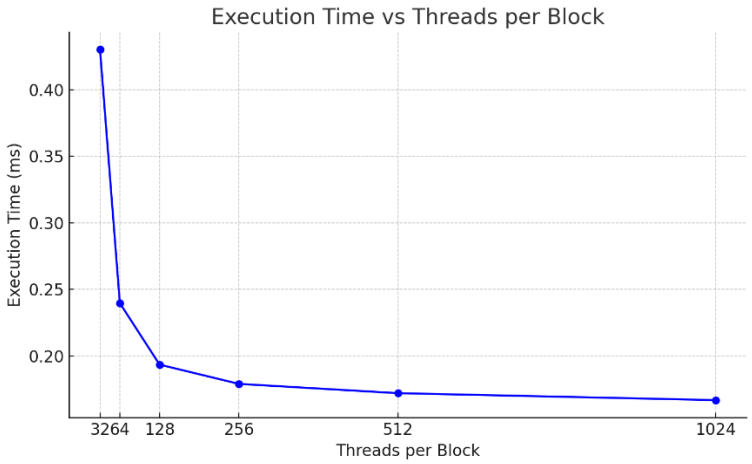
```
int threadsPerBlockList[] = {32, 64, 128, 256, 512};

for (int threadsPerBlock : threadsPerBlockList) {

    radixSort(dataCopy, n, threadsPerBlock);}
```

This code snippet performs radix sort using a range of thread configurations, testing different threadsPerBlock values for performance optimization. The threadsPerBlockList array contains a list of block sizes, ranging from 32 to 512 threads per block. A loop iterates over these configurations, and for each value, it calls the radixSort function, passing the data (dataCopy), its size (n), and the current thread block size. The rationale for varying threadsPerBlock is to test the efficiency of the sort algorithm under different CUDA thread configurations. Smaller thread blocks might allow better utilization of smaller datasets, while larger thread blocks could take better advantage of parallelism for larger arrays. This approach helps determine the best-performing configuration for a given dataset size and GPU hardware.

Graphical representation of output:



Code: Datatype

This implementation handles integers, floats, and negative values. Adjustments were made to sort negative numbers separately and recombine the dataset.

```
if (data[index] < 0) {
    // Handle negative numbers separately
    int digit = ((-data[index]) / exp) % 10;
    atomicAdd(&count[digit], 1);
} else {
    int digit = (data[index] / exp) % 10;
    atomicAdd(&count[digit], 1);}
```

This code ensures correct digit handling for both negative and positive numbers during radix sort. Negative numbers are processed by using their absolute value (-data[index]) to calculate the digit at the current exp position, while positive numbers are handled normally. The atomicAdd function safely increments the shared count array, avoiding race conditions among threads. This separation ensures accurate digit frequency counts for subsequent sorting steps. Handling negatives is essential for correctly sorting datasets containing mixed-sign integers.

Code: CPU vs GPU

This implementation benchmarks GPU performance against CPU for sorting datasets of increasing size.

```
std::vector<int> datasetSizes = {10, 100, 1000, 5000, 8000,10000};
    // Radix sort parameters
    int radixSize = 10;
    int threadsPerBlock = 512;
    // Iterate over dataset sizes
    for (int size : datasetSizes) {
        // Generate random dataset
        std::vector<int> dataset(size);
        for (int &val : dataset) {
            val = rand() % 10000;  // Random integers up to 10,000}
```

This code generates random datasets of varying sizes (10, 100, 1000, 5000, 8000, 10000) containing integers between 0 and 9999. The datasets are created to evaluate the performance of Radix Sort with a **radix size of 10** and **512 threads per block**.

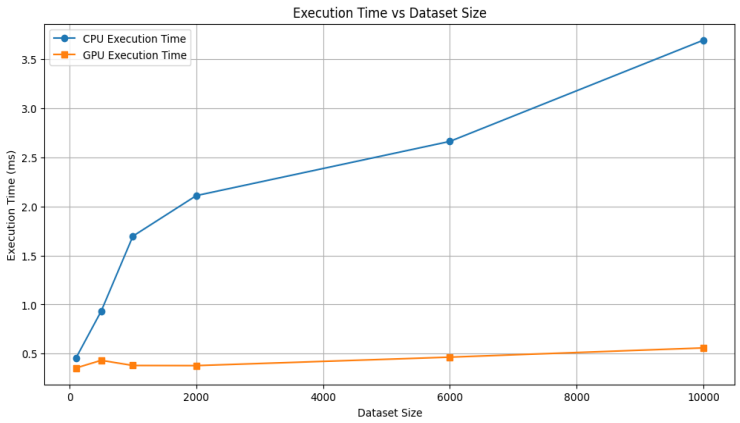
For cpu

```
void radixSortCPU(int *data, int n, int radixSize, float &cpuTime) {
    int maxVal = *std::max_element(data, data + n);
    std::vector<int> output(n);
    std::vector<int> count(10);
    auto start = std::chrono::high_resolution_clock::now();
```

```
for (int exp = 1; maxVal / exp > 0; exp *= radixSize) {
    std::fill(count.begin(), count.end(), 0);
    for (int i = 0; i < n; i++) {
        int digit = (data[i] / exp) % 10;
        count[digit]++;}
    for (int i = 1; i < 10; i++) {
        count[i] += count[i - 1];}
    for (int i = n - 1; i >= 0; i--) {
        int digit = (data[i] / exp) % 10;
        output[--count[digit]] = data[i];}
    std::copy(output.begin(), output.end(), data);}
```

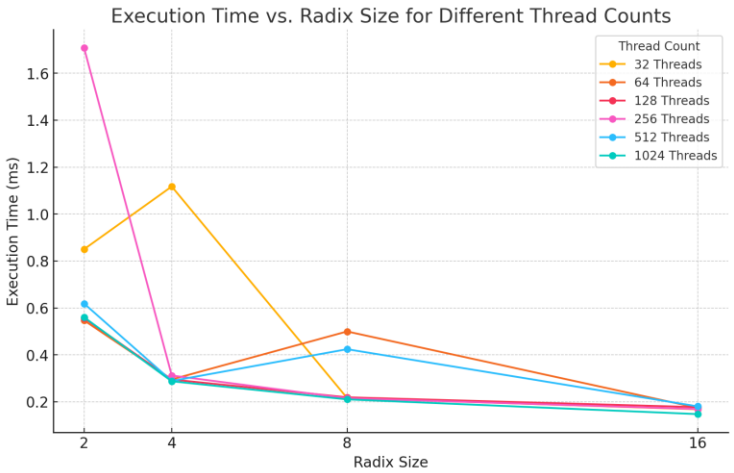
This function implements **Radix Sort on the CPU** for integers using a digit-based sorting approach. It iteratively sorts digits at each positional place (exp) while updating counts for each digit, reconstructing the sorted array using a temporary buffer. The total execution time is measured using std::chrono and returned in milliseconds through the cpuTime parameter.

This code compares Radix Sort performance on the **CPU** and **GPU**. It uses a kernel, radixSortKernel as shown initially, for parallel sorting on the GPU, processing digits in stages, calculating digit frequencies, and placing elements in sorted order. The CPU version uses standard sequential radix sort with vectors. Execution times are measured for both implementations for datasets of varying sizes. The GPU uses atomicAdd and shared memory for thread-safe counting and leverages CUDA events for precise timing. Results demonstrate the scalability of GPU parallelism for large datasets compared to the CPU.



The GPU proves to be approximately **4 times faster** than the CPU for larger datasets, showcasing its ability to deliver high performance without significant time degradation. While the CPU's execution time scales poorly with increasing dataset size, the GPU achieves near-constant performance, making it highly efficient for sorting large datasets. This result highlights the substantial advantage of using GPU-accelerated Radix Sort over its CPU counterpart, particularly in applications that demand scalability and low latency.

Optimal values for a particular dataset include radix size 8 and thread count 512



The graph shows the execution time of Radix Sort for varying radix sizes and thread counts, highlighting their impact on performance. At Radix Size = 2, execution time is highest, particularly for lower thread counts like 128 threads, due to the increased number of iterations. As the radix size increases to 4 and 8, execution time decreases significantly, demonstrating improved efficiency by reducing the number of sorting stages. For larger thread counts, particularly 512 threads and 1024 threads, execution time remains consistently low, with the best performance observed at Radix Size = 8 and 16. The execution time stabilizes around 0.2 ms for these configurations, indicating optimal GPU resource utilization. Therefore, the combination of Radix Size = 8 and 512 or 1024 Threads per Block provides the most efficient performance.

Challenges and Resolutions

Challenge 1: Mixed Data Types and Negative Values

- Issue: Incorrect handling of negative values caused zeros in the sorted dataset.
- Resolution: The radix computation logic was adjusted to process negative numbers separately by using their absolute values for digit extraction. The negative numbers were sorted independently and then recombined with the positive numbers, ensuring correctness.

Challenge 2: Memory Overwrites for Larger Radix Sizes

- Issue: Sorting with radix size 16 caused memory conflicts and errors.
- Resolution: Improved memory synchronization techniques (using __syncthreads) and dynamically allocated shared memory were employed to handle larger radix sizes. This resolved memory conflicts and ensured stable performance.

Challenge 3: Debugging for Scalability

- Issue: Sorting performance degraded for larger datasets.
- Resolution: Profiling tools such as **NVIDIA Nsight Compute** were used to analyze memory access patterns. Shared memory usage was optimized to reduce global memory transactions, improving scalability for larger datasets.

Conclusion

Radix Sort on CUDA-enabled GPUs demonstrates significant performance improvements over CPU-based sorting for large datasets. By optimizing Radix Sort for CUDA GPUs, this study achieved a 4x speedup compared to CPU-based implementations for datasets exceeding 1000 elements. Optimal configurations of radix size (8 bits) and thread count (512) balanced execution time and resource utilization, demonstrating GPU suitability for highly parallel workloads.

Future Work:

- Memory Coalescing for Greater Efficiency
  Future improvements could focus on optimizing memory coalescing, ensuring that memory accesses are aligned to minimize latency. This would further improve execution times for larger radix sizes.
- Multi-GPU Scalability
  Extending the implementation to support multi-GPU systems could significantly enhance performance for massive datasets. Techniques like workload partitioning and inter-GPU communication can be explored.
- Support for Real-Time Sorting
  Developing a real-time sorting version of this implementation, optimized for streaming data, would make it applicable for scenarios requiring continuous data processing, such as network packet sorting or sensor data streams.
- User-Friendly Application
  A user-friendly interface could be developed, allowing users to configure radix size, thread block size, and dataset parameters. This would facilitate benchmarking and optimization for various hardware configurations.

References:

1. Satish, N., Harris, M., & Garland, M. (2009). *Designing Efficient Sorting Algorithms for Manycore GPUs*. In *Proceedings of the 23rd ACM International Conference on Supercomputing (ICS'09)*. ACM.
2. Blelloch, G. E. (1990). *Prefix Sums and Their Applications*. Technical Report, School of Computer Science, Carnegie Mellon University.
3. Harris, M., Sengupta, S., & Owens, J. D. (2007). *Parallel Prefix Sum (Scan) with CUDA*. In *GPU Gems 3*, Chapter 39. NVIDIA.
4. Zhang, H., & Mueller, F. (2012). *GStream: A General-Purpose Data Streaming Framework on GPU Clusters*. In *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS'12)*. IEEE.
5. Lindholm, E., Nickolls, J., Oberman, S., & Montrym, J. (2008). *NVIDIA Tesla: A Unified Graphics and Computing Architecture*. *IEEE Micro*, 28(2), 39-55.
6. Merrill, D., Garland, M., & Grimshaw, A. (2012). *Scalable GPU Graph Traversal*. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*. ACM.
7. Hoare, C. A. R. (1962). *Quicksort*. *The Computer Journal*, 5(1), 10–16.
8. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd Edition). MIT Press.