Vector Sort Algorithms Using CUDA C

Afshin Khodaveisi

July, 2023

Abstract

This report presents the implementation and evaluation of vector sort algorithms using CUDA C. The purpose of this project is to explore different parallel programming techniques and evaluate the performance and scalability of various sorting algorithms on GPU architectures (Nvidia). In this report, I introduce the considered algorithms, describe their different implementations, discuss the adopted parallel programming techniques, and provide an evaluation of our program's performance and parallel scalability. I describe the input data used for testing, the approach taken to evaluate the performances of the solutions, and present the final results about their parallel scalability.

1 motivation

The divide and conquer algorithm[1], which is used and extended in this project, is considered due to its performance in three diverse evaluation cases: worst-case performance, average-case performance, and best-case performance, which all of them have case complexity about $n \log n$. Even in the worst-case performance scenario, the algorithm has case complexity $O(n \log n)$. Consequently, Merge sort is a good option for choosing an algorithm to sort a vector. Additionally, Merge sort provides the opportunity to compute and investigate each section separately from other parts. This attribute enables parallel programming to work under perfect conditions and provides good performance for parallel programming[1].

2 Sorting Algorithms

2.1 Divide and Conquer

2.1.1 Definition and Sequential Implementation

Merge Sort as divide and conquer algorithm used in Sequential and parallel technique to evaluate and gain best performance through CUDA C programming.Generally, According to the definition Merge Sort[2] is an efficient, general-purpose, and comparison-based sorting algorithm. Most implementations produce a stable sort, which means that the order of equal elements is the same in the input and output.
Conceptually, a merge sort works as follows:
1. Divide the unsorted list into n sublists, each containing one element (a list of one element is considered sorted).

2. Repeatedly merge sublists to produce new sorted sublists until there is only one sublist remaining. This will be the sorted list.
n the sequential implementation, the algorithm starts by sorting sublists with a length of one. Then, two sublists are created: one for the left side and another for the right side. These two sublists are then passed to the comparison method. This process is repeated for all elements in the vector. Eventually, the length of the sublists is increased to two, and the process is repeated again. Although this algorithm can be coded recursively, the iteration implementation was chosen because it makes evaluation and comparison easier.

Vector Size	Time	Error
128 * 128	0.425s	-
256 * 256	1.521s	-
512 * 512	5.986s	-
1024 *1024	24.025s	-

CUDA C, Sequential Implementation

2.2 CUDA C Implementation

2.2.1 Overview of CUDA C Programming

In this project, the merge sort algorithm has been extended and modified for CUDA C programming[3]. This modification provides an ideal condition for implementing parallel programming by utilizing the capabilities of multi-threads and multi-blocks, resulting in improved performance and reduced execution time. The CUDA C implementation is encapsulated within a helper function, enabling the identification and tracing of specific parts of CUDA C programming and facilitating error detection. Moreover, the algorithm's division process and parallelization strategies have been manipulated to achieve maximum performance and minimize memory usage. However, it is important to note that large integers (numbers with a high number of digits) in C have certain limitations.

2.2.2 Parallelization With Multiple Threads

In this parallelization approach, the vector is initially divided into a number of threads to enable multiple iterations over the vector. Each thread is assigned a specific sorting task. This process is repeated with various sizes of sublists until the entire vector is sorted and the sublist sizes exceed the vector size. The table below demonstrates that the sorting time in this implementation is significantly lower than in the sequential implementation. As a result, the performance of the parallelized implementation better than the sequential approach.

Vector Size	Time	Error
128 * 128	0.037s	-
256 * 256	0.145s	-
512 * 512	0.419s	-
1024 *1024	1.522s	-

CUDA C, Parallel Implementation With Multiple Threads

2.2.3 Parallelization With Multiple Blocks

In this implementation, there is reduced complexity, which provides the opportunity to decrease the execution time. However, due to the requirement of sorting all the sublists in a single step, the number of elements in the vector is limited compared to previous approaches. As a result, the entire vector is divided into blocks with only one thread each. All calculations within a given sublist size are performed in a single step, and all threads synchronize at the end of each step.

Vector Size	Time	Error
64 * 32	0.003s	-
64 * 64	0.004s	-
128 * 64	0.008s	-
128 * 128	0.013s	-

CUDA C, Parallel Implementation With Multiple Blocks

2.2.4 Parallelization With Multiple Threads and Blocks

In this approach, each thread inside blocks is assigned the task of sorting a specific portion of the vector. Consequently, the division and sorting processes are executed under conditions, ensuring fair distribution of the workload among threads. Another implementation of this parallelization technique has also been coded; however, it does not act the same level of performance as the advanced implementation. In this alternative approach, the number of threads and blocks is limited, and sorting within a certain sublist size is performed through multiple iterations over the vector. It is important to note that due to memory allocation constraints, this approach does not function in the same manner as the previous one and has limitations on the vector size it can handle.

Vector Size	Time	Error
128 * 128	0.023s	-
256 * 256	0.071s	-
512 * 512	0.245s	-
1024 * 1024	0.750s	-

CUDA C, Parallel Implementation With Multiple Thread and Blocks

3 Challenges

During the implementation of the project, one particularly challenging obstacle was encountered: the use of shared memory. In a parallel implementation involving multiple blocks and threads, shared memory had the potential to be highly effective in terms of data sharing among the blocks. The concept revolved around retrieving data only when necessary for each part (defined as the combination of all threads within a block), thus eliminating the need to retrieve data in every iteration and allowing a single data retrieval to be used for multiple sublists. However, an issue arose when attempting to modify the size of the sublists in each iteration. This posed inefficiencies and rendered shared memory in CUDA C less accessible for this purpose.
Furthermore, during the first attempt, the input data was generated as random numbers using a C library. However, when the data generation algorithm was adapted to generate numbers within specific thresholds using a specific seed, the evaluation time showed slight differences compared to the initial attempt. As a result, the range of numbers within the vector did not have a significant impact on the evaluation process, but it did have an influence on the outcomes.

References

[1] Wikipedia. "Divide-and-conquer algorithm." (2023), [Online]. Available: `https://en.wikipedia.org/wiki/Divide-and-conquer_algorithm`.

[2] Wikipedia. "Merge sort." (2023), [Online]. Available: `https://en.wikipedia.org/wiki/Merge_sort`.

[3] Nvidia. "Cuda c programming guide." (2023), [Online]. Available: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

Appendix

GitHub Repository

The project's source code, documentation, and related materials can be accessed in the GitHub repository. It contains the implementation files, scripts, and any additional resources associated with the project.

```
GitHub Repository:
https://github.com/AfshinKhd/cuda-sortVector
```