CS267 Final Project - Parallelizing Stereo Vision for Driverless Applications

Apratim Banerjee
MEng EECS
apratimbanerjee@berkeley.edu

Terence Neo
MEng EECS
terence.neo@berkeley.edu

Zeid Solh
MEng EECS
zeidsolh@berkeley.edu

April 2024

1 Introduction

View the code at: GitHub Repo.

Stereo vision (depth perception using two cameras) plays a crucial role in self-driving cars by building a 3D understanding of the surrounding environment. However, processing these images in real-time, essential to quick reactions, can be computationally expensive, presenting a critical bottleneck to self-driving car technology.

One possible application of parallel computing that we will work on for the course project is the calculation of Disparity Maps for Stereo Vision. Calculating the disparity map, which represents the differences in image location of the same 3D point from two cameras, is the core of stereo vision. The algorithm has a huge potential for parallelism, and the speed at which the algorithm runs will be extremely useful in real-time perception tasks in autonomous systems, enabling the processing of high-resolution images and videos.

Since parallel computing can be applied to algorithm steps that are not interdependent, such as analyzing different pixels of an image, we can distribute this workload across multiple cores or processors. We will explore parallelization techniques like SIMD, OpenMP, OpenMP SIMD, CUDA, CUDA SIMD, parallelizing on the CPU and GPU, bench-marking the algorithms, and comparing the performance improvements.

Naively, we expect that adopting parallelization techniques will speed up the performance proportionately to the number of cores or processors used.

1.1 Key prior work

[1] Detailed efforts to optimize the computation of disparity maps, which are crucial for deriving depth information from stereo camera images. Spryn explores several parallelization strategies to enhance performance, including multi-threading with OpenMP and SIMD instructions, and leveraging GPU compute capabilities through CUDA and OpenCL frameworks. **[2]** Discusses a GPU-based parallel algorithm for generating disparity maps. The algorithm aims to reduce computational costs

and improve real-time computer vision performance. By minimizing redundant operations and applying a median filter for noise reduction, the study achieves a significant speedup, demonstrating the potential of GPU for efficient stereo matching and disparity map calculation in real-time applications with low computing cost constraints. **[3]** Aims to parallelize disparity map creation using the CPU and GPU, focusing on a block-matching algorithm. It provides a Python implementation for CPU and PyCUDA for GPU, with the GPU version achieving significant speed improvements. This work builds on the concepts of disparity map generation and optimization through hardware acceleration.

2 Method

Computing depth from a pair of images in a stereo camera involves matching each pixel in the image from the left camera with the corresponding pixel in the right camera. The further apart the pixels are in terms of X-Y coordinates, the closer the object is to the camera. We used a block-matching algorithm by converting input images into blocks of arrays, computing the disparity map for a given block in parallel, and then converting the disparity map to the appropriate data type and scale.

We collected data on our own for testing. We conducted multiple tests, first on images of varying resolutions, and then videos of varying resolutions of the UC Berkeley Campus. We captured the images and videos on a smartphone with two cameras (iPhone 15), simultaneously capturing images from each camera as the left and right stereo images.

To process the video, we used the OpenCV library. We split the videos into multiple frames, and then, similar to how we processed images, we resized and converted them to mean-adjusted grayscale images. Then, after processing the frames, we convert the disparity maps of each frame back to a video and store them in an output folder.

Beginning with no parallelization, we explore parallelizing via multi-threading using the OpenMP library, using Single Instruction Multiple Data (SIMD) instructions, and CUDA.

2.1 Algorithm

The further apart the pixels are in terms of their coordinates, the closer the object is to the observer. Pixels with larger disparity values (closer to the camera) are shown in a lighter color in a disparity map.

In our experiments, we used the block-matching algorithm to calculate our disparity map. The algorithm takes a two step process:

1. In the first step, it considers a square block around each pixel in the left image, comparing this square block to various locations in the right image. A loss function is then calculated for each block position. If a perfect match is found, the loss function returns zero. However, due to differences in conditions (i.e., lighting), exact matches are rare. In such instances, the pixel's position can be estimated by fitting a parabola to the three nearest matching points.

2. In the second step, it calculates the distance between the best-match pixels giving the disparity score.

We will use a standard loss function of the Sum of Absolute Differences (Equation 1) for our purposes.

$$\sum_{x,y} \left(|\text{left\_block}(x,y) - \text{right\_block}(x,y)|\right) \tag{1}$$

As we are capturing the image using two cameras on the same phone, we will assume that the pictures are stereo-rectified, meaning that the best match in both images will have the same Y-coordinate, and the X-coordinate of the best match in the right image will be greater than or equal to the X-coordinate of the same pixel in the left image. This assumption will significantly reduce the search space and speed up calculations.

The disparity map algorithm is expressed in pseudo-code below.

```
For each Y, X in Left_image: # Step A
    losses = []
    For each candidate pixel in Right_image: # Step B
        loss = Compute loss functions over pixel neighborhood in the
             block of both right and left images # Step C
        losses.append(loss)

    bestidx = argmin(losses)

    A = losses[bestidx-1]
    B = losses[bestidx]
    C = losses[bestidx+1]
    Parabola = FitParabola((-1, A), (0, B), (1, C))
    (MatchX, MatchY) = Minimize(Parabola) across all candidate
        pixels in Right_image
    D = bestidx + MatchX

    Disparity[X,Y] = D
```

Listing 1: Disparity Map Algorithm

2.2 Order of Complexity and Computational Intensity

From the above pseudo-code, we hypothesized that this algorithm will run with an order of complexity of:

$$O(\text{Block Size} \times \text{Scan Steps} \times \text{Image Resolution}) \tag{2}$$

Where the block size is the square block of pixels we used to calculate the loss, scan steps is the number of steps we scanned over to search for pixel candidates.

Next, to calculate the computational intensity of the disparity map algorithm, we'll analyze the number of floating-point operations (FLOPs) and the amount of data moved.

Number of Floating Point Operations (FLOPs)

- The main computation occurs in computing the loss function (Equation 1), where the sum of absolute differences (SAD) is computed over a block. This operation involves computing the absolute difference between corresponding pixels in the left and right images and summing them up.
- Each pixel comparison requires one subtraction operation and one absolute value operation, which can be considered as two FLOPs.
- The total number of FLOPs per block comparison is thus approximately

$$2 \times \text{block width} \times \text{block height} \tag{3}$$

Amount of Data Moved

- Data movement occurs mainly when accessing pixel values from the left and right images
- Each pixel access involves reading one byte because the pixel values in the grayscaled image are stored as unsigned chars
- The amount of data moved per block comparison is hence approximately

$$2 \times \text{block width} \times \text{block height bytes} \tag{4}$$

Therefore, the computational intensity can be calculated as:

$$\text{Computational Intensity} = \frac{\text{Number of FLOPs}}{\text{Number of bytes moved}} = \frac{2 \times \text{block width} \times \text{block height}}{2 \times \text{block width} \times \text{block height}} = 1 \tag{5}$$

This indicates that the algorithm is balanced between computation and data movement, with approximately equal amounts of computation and data movement involved in each block comparison.

3 Parallelization Techniques Used

From the above pseudo-code, we can see that several algorithm parts can potentially be parallelized, as labelled Steps A, B, and C in the pseudo-code.

- Step A: Calculation of the disparity value for each pixel in the Left_image is independent of each other
- Step B: The disparity value between a pixel in Left_Image, and each pixel in the Right_Image is also independent.
- Step C: The loss value of each pixel pair in the neighborhood can also be calculated independently.

3.1 SIMD

SIMD takes advantage of the ability to perform a Single Instruction on Multiple pieces of Data. By performing a SIMD operation, efficiency is increased because it reduces the total number of operations required.

SIMD's limitations lie in the fact that the exact same operation has to be used on multiple pieces of data, reducing flexibility. Fortunately, the computation of the loss function is a great candidate for SIMD.

Without SIMD, the loss function can be calculated with nested for loops that iterate over the square block.

```
1  loss = 0;
2  for y from 0 to height:
3      for x from 0 to width:
4          loss += Abs(Left_Image(x,y) - Right_Image(x,y))
5  return loss
```

Listing 2: Loss function pseudo-code

With SIMD, we can only loop over the y values, passing each row into a SIMD SAD instruction, which computes the loss for the entire row in parallel. This reduces the nested for loops into single for loops, reducing the order of complexity.

3.2 OpenMP

`#pramga omp parallel` for is used for each of the for loops, utilizing multi-threading to parallelize the disparity map computation, allowing for concurrent execution of independent tasks.

Notably, Step A, which calculates the disparity value for each pixel in the left image, exhibits significant speedup due to its inherent independence. However, Step B, which involves computing the disparity value between a pixe l in the left image and each pixel in the right image, experiences minimal improvement. This is likely because the overhead associated with thread creation and destruction outweighs the benefits of parallelization for such fine-grained tasks. Similarly, Step C, calculating the loss value for each pixel pair in the neighborhood, suffers a substantial slowdown. This can be attributed to the parallelization of nested loops, overwhelming the system and diminishing performance.

3.3 CUDA GPUs

The parallelization methods discussed earlier are limited because CPUs typically support only a few simultaneous threads, limiting the scalability of parallel workloads. This approach benefits from the inherent parallelism within the blocking algorithm and the massive number of threads available on GPUs compared to CPUs.

We ran the code on a CUDA implementation of the blocking algorithm. The implementation utilizes pre-allocated memory on the GPU to store intermediate results, improving performance by minimizing data transfers between CPU and GPU. This can improve performance by reducing data transfers between CPU and GPU.

4 Challenges

Parallelizing the blocking algorithm presents several challenges that mainly stem from the inherent characteristics of the algorithm, the hardware architecture, and the parallelization techniques employed.

4.1 Granularity of Parallelism

The blocking algorithm involves several nested loops, each potentially parallelizable. However, parallelizing at the lowest level of granularity (every single for loop) may lead to excessive overhead and inefficient resource utilization. Finding the optimal balance between task granularity and parallelization overhead is crucial for maximizing performance.

4.2 Load Balancing

Inherent workload distribution imbalance across threads can arise due to variations in image characteristics and block-matching complexities. Uneven workload distribution can lead to underutilization of resources and inefficient parallel execution. Designing effective load-balancing strategies to evenly distribute workload among processing units could maximize parallel efficiency. We would have liked to explore further if we had more time.

4.3 Sub-Optimal Memory Access Patterns

Concurrent memory accesses by multiple processing units can lead to contention and performance degradation. Optimizing memory access patterns, minimizing data movement, and exploiting memory hierarchies effectively is another potential area for exploration if we had more time.

5 Experiments

Our primary objective was to assess the performance improvements that each parallel computing method could offer and identify the optimal configurations for real-world autonomous driving applications.

To validate our implementation, we took high-resolution and low-resolution images and videos of the UC Berkeley Campanile and the Memorial Glade, to benchmark the performance enhancements facilitated by Single Instruction, Multiple Data (SIMD), OpenMP, and CUDA technologies. The following **variables** were altered throughout the experiments to measure their impact on performance:

1. **Resolution:** We tested images at two resolutions: high (3024x4032) and low (750x1000), to understand how an increased number of pixels affects computation time and parallelization efficiency.

2. **Threads:** The number of threads was varied (1, 2, 4, 8, 16, 32, 64, 128) to explore how the time taken for computation scales with increased concurrency. This variable was particularly critical for OpenMP and OpenMP SIMD parallelization techniques, where thread management can significantly influence performance.

3. **Parallelization Methods:** We applied different parallelization strategies: single-threaded, single-threaded with SIMD, OpenMP, OpenMP with SIMD, CUDA, and CUDA with SIMD. We also compared the different methods in their best run-time configuration to assess which one is the fastest performing technique.

4. **Block Size and Scan Steps:** These parameters were adjusted to test their effect on the computational complexity and the efficiency of the block-matching algorithm. We also tested them for linearity and scalability. Larger block sizes and more scan steps can potentially increase accuracy but also raise the computational burden.

5.1 Image Specifications

We analyzed the results for images of Campanile (low-resolution and high-resolution). Below are the specifications of both the images.

Parameter	**High-Resolution**	**Low-Resolution**
Image Size	(3024x4032)	(750x1000)
Block Size	7	7
Left Scan Steps	50	50
Right Scan Steps	50	50
Disparity Metric	Sum_Absolute_Difference	Sum_Absolute_Difference

Table 1: Comparison of Parameters

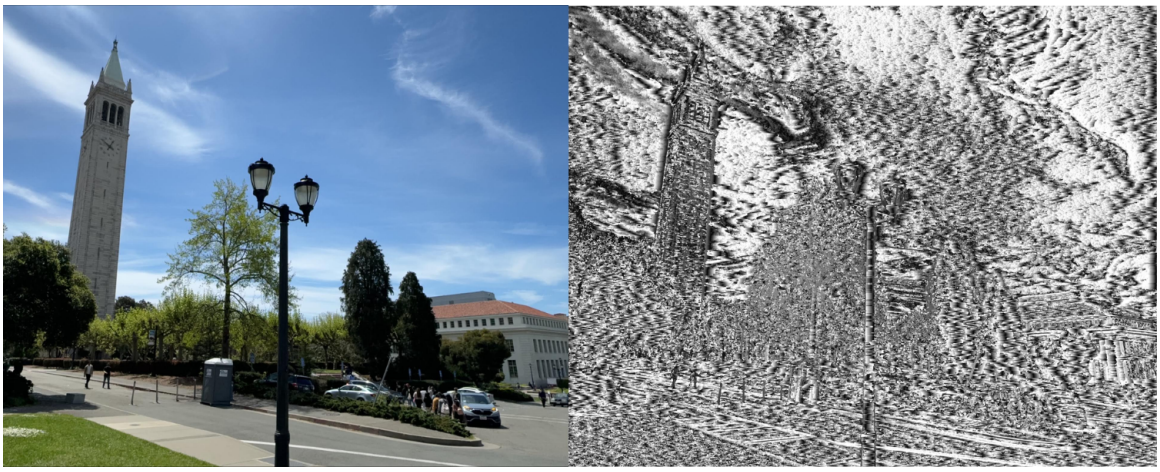


Image 1: Disparity Map of Campanile

5.2 Single-Threaded Performance

Here, we are comparing the Image Size (3024x4032) vs. the Image Size (750x1000). The graph below presents performance metrics for image processing at two different resolutions: High-Resolution and Low-Resolution. We are using one thread here.

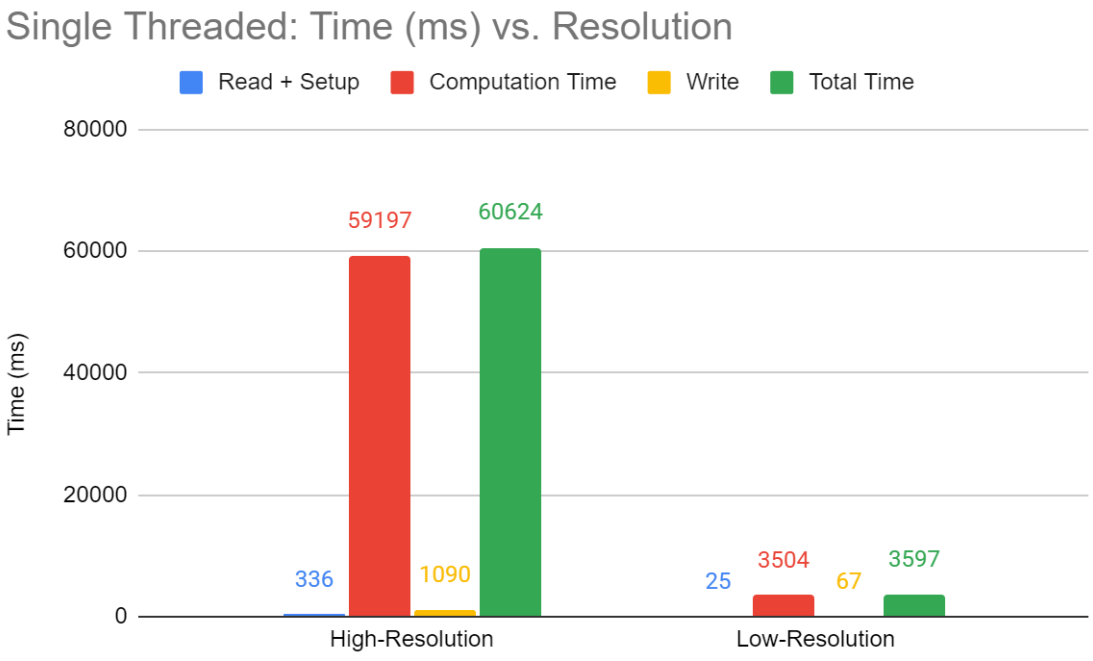


Figure 1: Single Threaded High and Low Resolution Processing in ms

Name	Threads	Time	Read + Setup	Computation Time	Write	Total Time(ms)
High-Resolution	1	60385	336	59197	1090	60624
Low-Resolution	1	3580	25	3504	67	3597

Table 2: Single Threaded High and Low Resolution Processing. Time in ms

Table-2 explains the variables used in this graph. We will be using these variables for the subsequent subsection graphs.

5.3 Single-Threaded SIMD Performance

Single Instruction, Multiple Data (SIMD) significantly improves performance (**Figure 2**) compared to single-threaded execution (**Figure 1**). This enhancement demonstrates the efficiency of performing multiple operations in parallel within a single processor instruction.

Name	Threads	Time	Read + Setup	Computation Time	Write	Total Time(ms)
High-Resolution	1	22454	222	21373	1002	22598
Low-Resolution	1	1403	26	1284	103	1415

Table 3: Single Threaded SIMD High and Low Resolution Processing. Time in ms

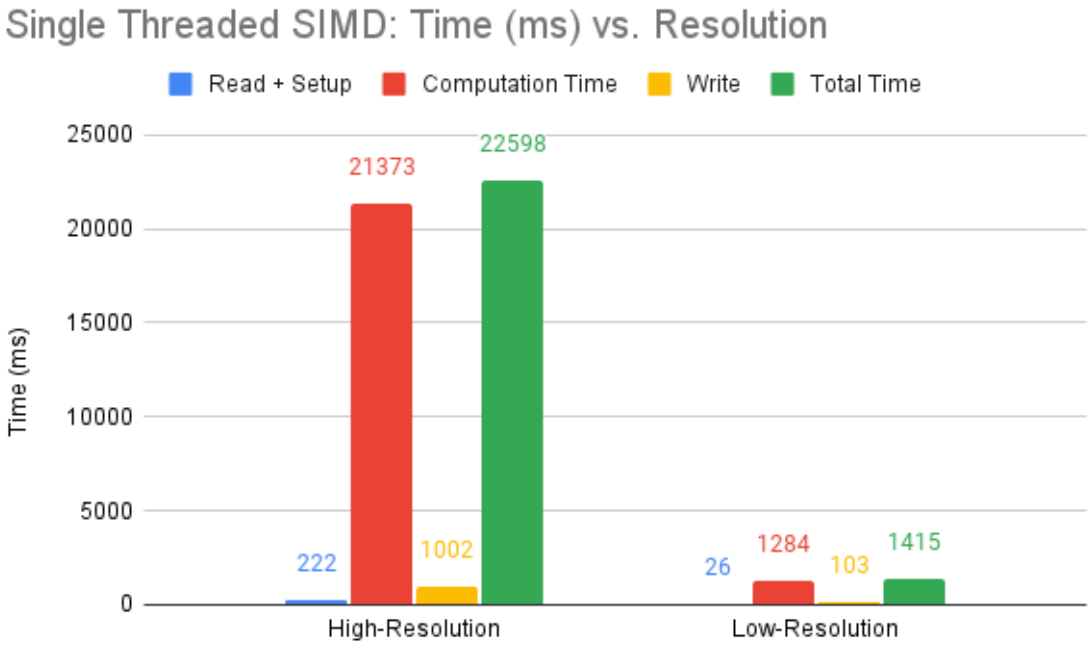


Figure 2: Single Threaded High and Low Resolution Processing in ms

5.4 OpenMP

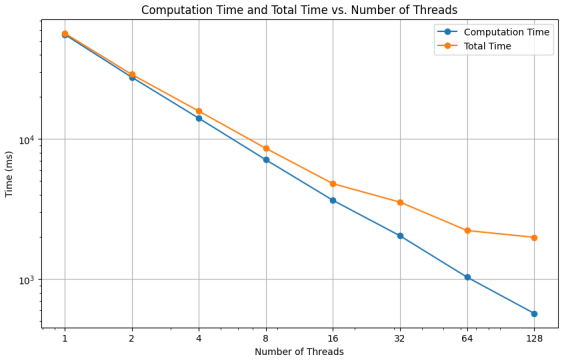
Our experiments with OpenMP on the high and low resolution images aimed to validate the hypothesis that parallelizing the disparity map calculation would linearly reduce computation time. We conducted a series of tests to evaluate the scalability of OpenMP by varying the number of threads and assessing the impact on processing time and system resource utilization. The specific objectives of our OpenMP experiments were as follows:

1. To determine the optimal number of threads for various stages of the disparity map algorithm (Steps A, B, and C as labeled in **Listing 1:Disparity Map Algorithm** that would yield the best performance in terms of speed and resource efficiency.

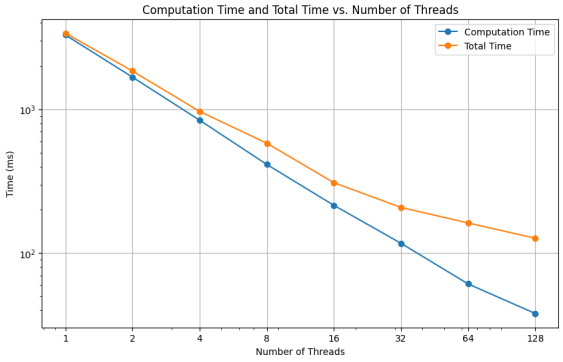
2. To assess the performance gains from using OpenMP in comparison to single-threaded and other parallel processing approaches like SIMD and CUDA.

3. To understand the overhead introduced by thread management in OpenMP and its implications on the computational efficiency and scalability of our application.

Note that we are varying the number of threads and measuring scaling as a performance indicator for OpenMP. As we increased the number of threads, the computation time reduced linearly, as can seen in **Figure 3**. The read, setup, and write time mostly remained constant. Below is the recorded data.



(a) OpenMP High-Resolution Performance in ms

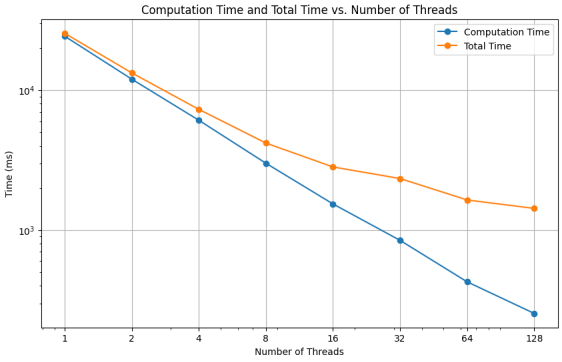


(b) OpenMP Low-Resolution Performance in ms

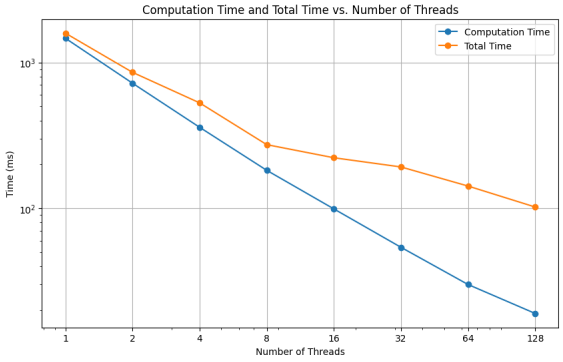
Figure 3: OpenMP Computation Time and Total Time vs Number of Threads

5.5 OpenMP SIMD

OpenMP and Single Instruction, Multiple Data (SIMD) operations aims to assess the effects of multi-threading and data-level parallelism. OpenMP SIMD extends the conventional OpenMP parallelization framework by allowing the simultaneous execution of the same operation on multiple data points, which can significantly enhance processing throughput for computationally intensive tasks such as disparity map calculation. The objectives of OpenMP SIMD were similar to OpenMP as highlighted in **Section 5.4**. The performance of OpenMP SIMD is recorded in **Figure 4**



(a) OpenMP SIMD High-Resolution Performance in ms



(b) OpenMP SIMD Low-Resolution Performance in ms

Figure 4: OpenMP SIMD Computation Time and Total Time vs Number of Threads

5.6 CUDA

Our experiments with CUDA were designed to assess the feasibility and effectiveness of GPU-based parallelism The main goals of these experiments included:

1. Benchmark the CUDA implementation against CPU-based implementations (both single-threaded and parallelized with OpenMP and SIMD) to quantify performance improvements

and identify any potential bottlenecks.

2. Evaluate the impact of GPU architecture on the disparity map computation, with focus on how well different stages of the algorithm map to the parallel processing capabilities of GPUs.

CUDA parallelism gave us a tremendous performance boost as shown in **Figure 5**. Although more limited than those on CPUs, the GPU's SIMD-like instructions still offer substantial computational power.

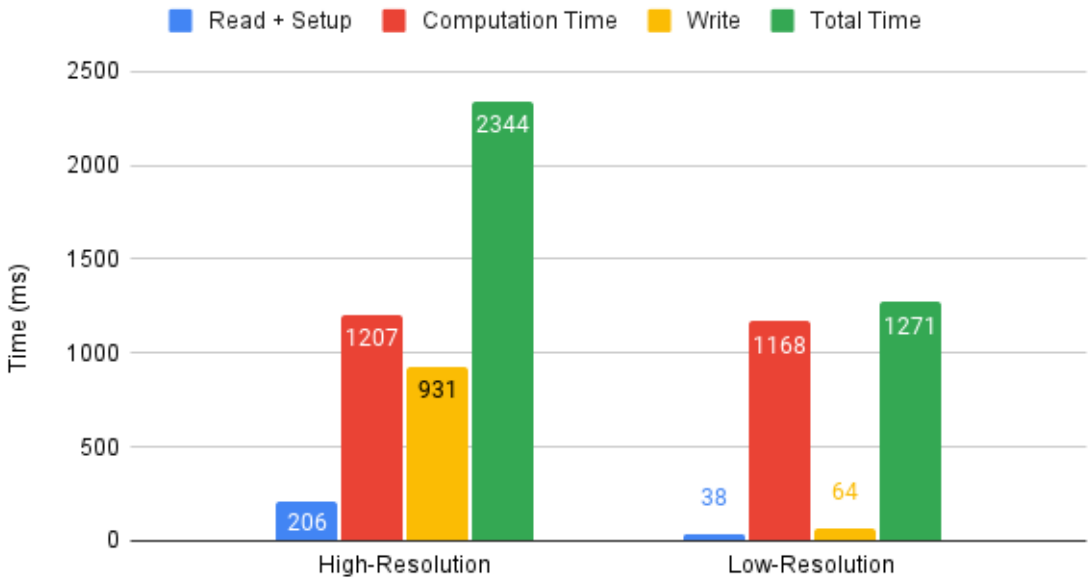


Figure 5: CUDA Performance in ms

5.7 CUDA SIMD

Our motivations for CUDA SIMD were similar to the CUDA experiments highlighted in **Section 5.6**. We aimed to exploit the synergies between CUDA's powerful GPU-based parallel processing and SIMD's ability to perform the same operation on multiple data points simultaneously. The expectation to get a better runtime as compared to CUDA. However, we got similar results with not much improvements (**Figure 6**) most likely due to inefficient SIMD Utilization. SIMD within CUDA is beneficial when the operations performed are highly uniform and can be perfectly vectorized. If the disparity map calculations involve varying conditions that lead to divergent code paths (branching), the SIMD units may not be fully utilized.

Figure 6: CUDA SIMD Performance in ms

5.8 Comparison of All Algorithms

This is the most important section which compares and benchmarks the computation time of all algorithms (**Figure 7**).

5.8.1 Analysis and Insights

1. **OpenMP vs. Single Threaded Approaches:** OpenMP and OpenMP SIMD significantly outperform the single-threaded approaches. This enhancement suggests a substantial benefit from multi-threading, as the algorithm likely capitalizes on parallelizing independent calculations across multiple CPU cores. The drastic reduction in computation time demonstrates the efficiency of effective multi-core utilization.

2. **SIMD Improvements:** The integration of SIMD instructions markedly improves performance in both single-threaded and OpenMP environments. This improvement is anticipated as SIMD enables processing multiple data points simultaneously within a single instruction cycle, which is beneficial for the repetitive operations typical in disparity map calculations.

3. **Relative Performance of CUDA:** Despite CUDA's capabilities for extensive parallel computations on a GPU, both CUDA versions underperform compared to OpenMP. Possible reasons include:

    - **Overhead of Memory Transfers:** The necessity of transferring data between CPU and GPU memory introduces significant overhead, offsetting the computational speed advantages for tasks not heavily dominated by computation.
    - **Granularity and Optimization Issues:** The CUDA implementation is not optimized for the specific computational and memory access patterns required.

4. **Superiority of OpenMP:** The outstanding performance of OpenMP, especially when combined with SIMD, suggests that disparity map calculations involve many small, independent tasks perfectly suited for multi-core CPUs. The efficient management of thread life cycles in OpenMP minimizes overhead, making it preferable over managing a large number of GPU threads with additional architectural demands.

5.8.2 Chart Results

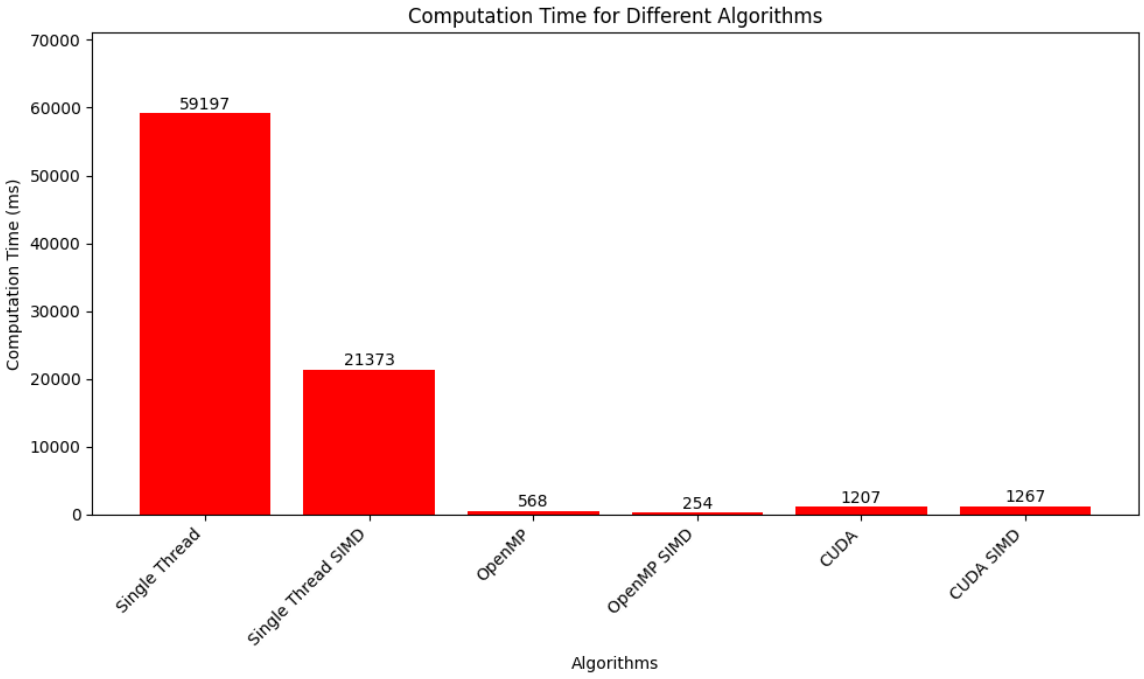


Figure 7: All Algorithms' Time Comparsion (Time in ms)

5.9 Multiple Simulation Runtime Comparison

In order to further probe the superority of OpenMP performance as compared to CUDA, we conducted an experiment involving multiple simulations of the same experiment. This particular set of tests was designed to evaluate how each parallelization strategy handles repeated runs.

5.9.1 Experiment Setup

The experiments involved running multiple simulations consecutively without altering the initial conditions or parameters between runs. Our results were recorded as shown in **Figure 8**.

5.9.2 CUDA's Performance Edge

In these tests, CUDA demonstrated superior performance over OpenMP, a result that is somewhat counterintuitive given the single-simulation outcomes where OpenMP frequently outperformed

CUDA. The reasons behind CUDA's enhanced performance in multiple simulation environments are twofold:

1. **Maximized GPU Throughput:** CUDA excels in scenarios where the computational load is consistently high and can be evenly distributed across the GPU's many cores. In multiple simulation tests, the continuous operation mode allows CUDA to utilize the GPU's parallel processing capabilities to their fullest extent without the frequent start-stop penalties that might affect CPU-based computations.

2. **Minimal Pre-processing Overhead:** One of the significant advantages of CUDA in repeated simulation scenarios is the reduced need for pre-processing. In CUDA implementations, once the initial data is transferred to the GPU memory, further computations on this data do not require repeated transfers or pre-processing steps.

5.9.3 Graphical Analysis

Refer to the chart below, which visually compares the cumulative computation times for OpenMP and CUDA in their best run-time configuration.

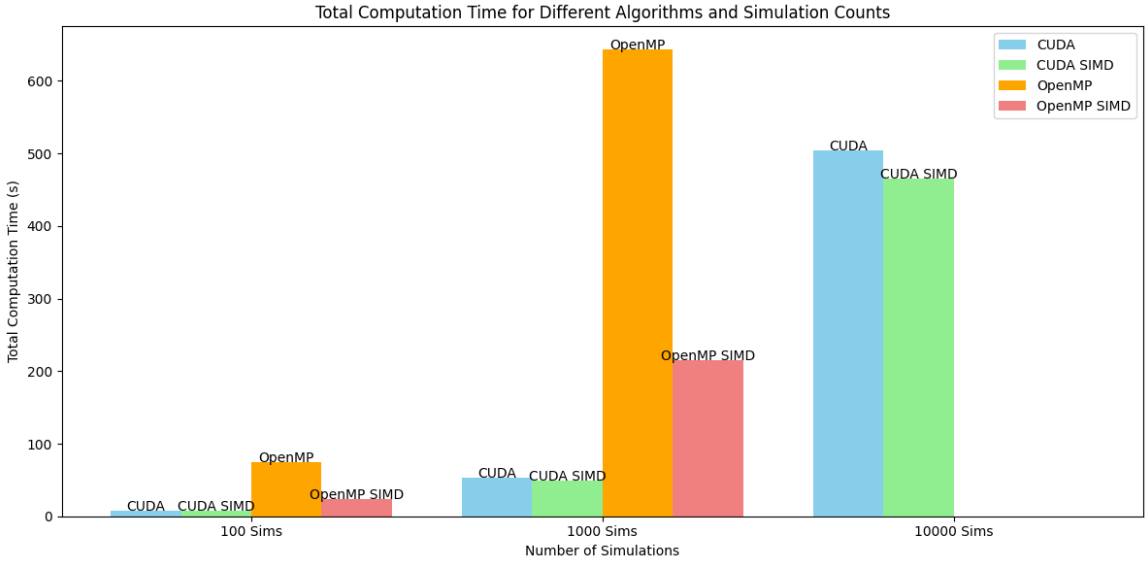


Figure 8: OpenMP and CUDA performance on multiple simulations (time in seconds)

5.10 Varying the Complexity Function Parameters

Perhaps one of the most important experiment was to observe how the different parameters affect the complexity and subsequently the computation time of the block-matching algorithm. To this end, we conducted a series of experiments where we varied three critical parameters: Block Size, Scan Steps, and Resolution. These parameters were systematically adjusted to investigate their impact on computation time and to test the hypothesized linearity of their effects.

5.10.1 Graphical Analysis

The parameters were varied one at a time while keeping others constant to isolate the effects of each on the computation time. As we varied each parameter, we wanted to observe if there was linear behaviour, which was our expectation.

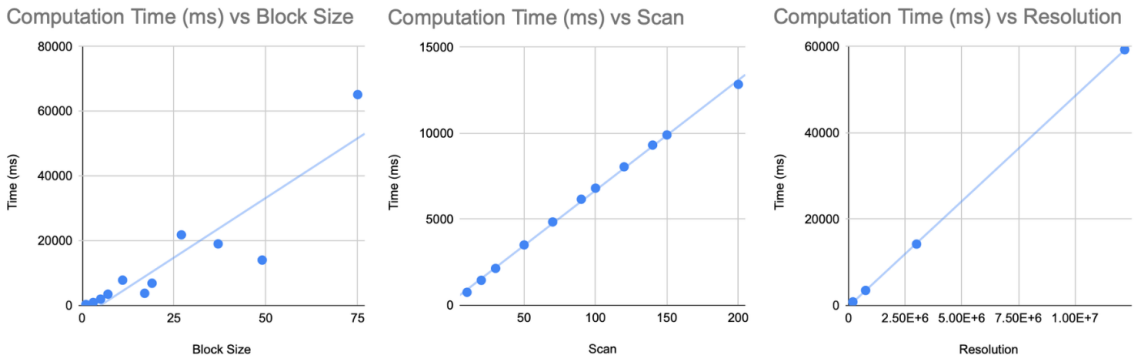


Figure 9: Linearity test for Block Size, Scan, Resolution

1. **Block Size**: As depicted in the first graph of in **Figure 9**, computation time increases almost linearly with the increase in block size. This is expected as larger blocks require more comparisons per pixel, significantly increasing the number of operations needed to compute the disparity map.

2. **Scan**: The middle graph in **Figure 9** shows a clear linear relationship between the scan range and computation time. This linear increase aligns with theoretical expectations because expanding the scan range enlarges the search area for matching pixels, thus linearly increasing the computational workload.

3. **Resolution**: The third graph illustrates a linear increase in computation time as image resolution increases. Higher resolutions result in a greater number of pixels to process, which escalates the computational demands proportionally.

4. **Implications**: These results suggest that optimizing parallelization strategies must consider the linear increase in computational load as these parameters expand. Moreover, the scalability of different parallel computing architectures, like OpenMP and CUDA, can be benchmarked against these parameters to optimize performance in real-world scenarios.

5.11 Video Specifications and Results

Video Specifications: Image Resolution: 1920x1080, 114 frames in total. Our final part of experiments was to analyze the performance of computing the disparity map of a video by splitting it into multiple frames and noting the **Total time per frame**, **Total time taken for the video**, and **Frames per Second (FPS)** achieved. We observed comparable results for OpenMP and CUDA (in their best runtime configuration).

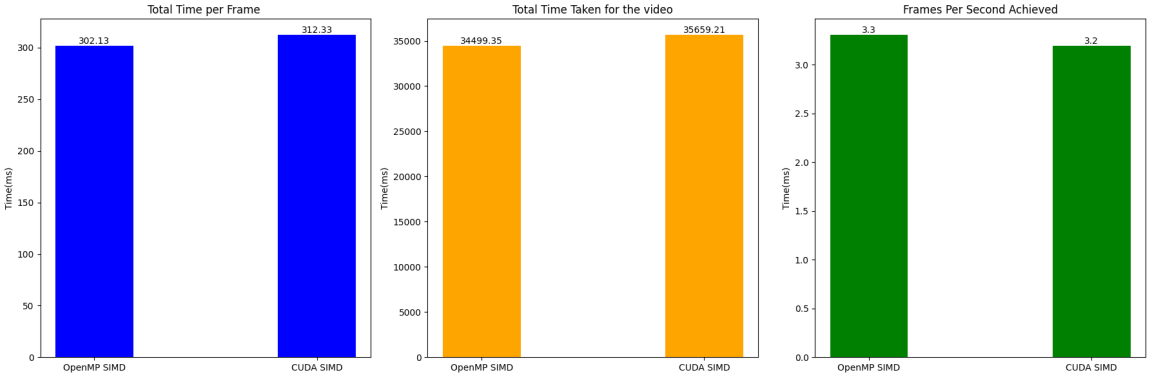


Figure 10: Video Performance Results

6 Results and Dicussion

Throughout the different experiments conducted, we came to many interesting insights. **Figure 2** allowed us to show the results of our base case for computing the disparity map of a pair of images, the single-threaded case. The figure compares the computation time taken for a high resolution (59197 ms) vs. a low resolution (3504 ms). More on this will be discussed later. It is important to note that we only care about the computation time as the read + setup and the write times are conducted by the operating system using a single thread and are not part of this study.

Once we introduce Single Instruction, Multiple Data (SIMD), the single-threaded performance already sees a tremendous improvement as the computation time for the high resolution decreases to 21373 ms (2.7x faster) and for the low resolution decreases to 1284 ms (also 2.7x). This is expected as SIMD usually improves performance by allowing the execution of the same operation on multiple parts of the data simultaneously, reducing the computation time for tasks that can be parallelized, as was demonstrated by our experiments.

Tables 4,5: OpenMP Performance for High-Resolution Image in the Appendix section show the data once we employ OpenMP instead. We can see that as we multiply the number of threads by two, the computation time gets divided by two in a general way. This is true for both high-resolution and low-resolution images. For instance, for the high-resolution table, when we go from 4 threads to 8 threads, computation time decreases from 14076 ms to 7105 ms. This is an example of strong scaling because the computation time is halved as the number of threads doubles, revealing that the workload is evenly distributed among the increased number of threads. This happens because the algorithm parallelized well without significant overheads, allowing for a linear speedup with the addition of more processing resources. We can come to the same conclusions from tables **Tables 6 and 7** as they also represent the same data but for an algorithm where we employ OpenMP + SIMD. The result is an even faster model, where the computation time for the high-resolution image goes from 568 ms using OpenMP alone for 128 threads to 254 ms using OpenMP + SIMD for 128 threads (2.2x speedup). Hence, combining OpenMP with CUDA results in faster computation times.

Table 8 and **Figure 5** revealed the results of using CUDA to compute the disparity map, where we can see the computation time for the high-resolution image takes 1207 ms while the low-resolution image takes 1168 ms. **Table 9** and **Figure 6** also reveal the same information for CUDA + SIMD, where the high-resolution image takes 1267 ms and the low-resolution image takes 1134

ms. Interestingly, we do not see significant differences between low-resolution and high-resolution photos and between CUDA and CUDA + SIMD. This is because CUDA introduces computational overhead, which is not being maximized in terms of throughput. In addition, the combination of CUDA + SIMD does not improve the performance as CUDA is already optimized alone, leaving little room for additional improvements.

Figure 7 compares the computation time of all algorithms for the high-resolution image. OpenMP SIMD (128 threads) displays the fastest time at 568 ms compared to the naive single-threaded time, which sits at 59197 ms, presenting a 233x speedup.

Figure 8 compares OpenMP and CUDA performances on multiple simulations of the same image, hence comparing the compute time without having to preprocess an image multiple times (we only have to do it once). From only 100 simulations, we can see that CUDA and CUDA + SIMD outperform OpenMP and OpenMP + SIMD. For 1000 simulations, this difference becomes even more significant, where CUDA outperforms OpenMP by 12.5x. For 10,000 simulations, we do not compute the times for OpenMP and OpenMP + SIMD as they become too slow, but we can now see that CUDA + SIMD outperforms CUDA. These results show us that CUDA and CUDA + SIMD are only great when we can maximize the throughput.

Figure 9 contains three sub-graphs where we plot the computation time against the single-threaded algorithm' block size, scan size, and resolution. We notice that they display linearity and confirm our hypothesis that the disparity map calculations will run with an order of complexity of

$$O(\text{Block Size} \times \text{Scan Steps} \times \text{Image Resolution}) \tag{6}$$

Lastly, **Figure 10** analyzes the performance of computing the disparity map on a video (sequence of images). By including the whole time here (including reading and writing), we notice that we can only achieve around 3.2 frames/second, and the results for OpenMP SIMD and CUDA SIMD are comparable. Hence, it would be interesting to explore in future studies how we can achieve fast preprocessing, reading, and writing to reach the 30 frames/second that can be implemented into self-driving cars.

7 Conclusion

In this project, we presented different techniques and evaluation methods to assess image processing using parallel computing techniques. We focus on calculating disparity maps crucial for stereo vision applications, employing parallelization strategies such as SIMD, OpenMP, and CUDA to improve computational efficiency. Our experiments demonstrated significant improvements in processing times, validating the potential of parallel computing to overcome the computational bottleneck in stereo vision applications.

8 Contribution

Everyone has equal contribution to the code and report.

A Appendix

Table 4: OpenMP Performance for High Resolution Image

Number of Threads	Read + Setup	Computation Time	Write	Total Time	Time (ms)
1	216	55887	889	56993	57430
2	212	27637	1092	28942	28810
4	217	14076	1540	15834	15049
8	212	7105	1260	8578	7949
16	220	3648	928	4796	4601
32	223	2035	1281	3541	2895
64	225	1030	964	2220	1954
128	215	568	1204	1987	1859

Table 5: OpenMP Performance for Low Resolution Image

Number of Threads	Read + Setup	Computation Time	Write	Total Time	Time
1	33	3294	65	3393	3406
2	107	1676	65	1849	1734
4	22	841	104	968	896
8	104	414	63	582	490
16	31	215	62	309	277
32	28	117	62	208	181
64	24	61	76	162	124
128	27	38	61	127	97

Table 6: OpenMP SIMD Performance for High Resolution Image

Number of Threads	Read + Setup	Computation Time	Write	Total Time	Time (ms)
1	258	24306	948	25513	25166
2	244	11991	1051	13286	12990
4	228	6096	968	7292	6944
8	214	3005	967	4187	3902
16	277	1538	1010	2826	2425
32	240	845	1250	2336	1810
64	230	427	984	1642	1330
128	233	254	941	1429	1117

Table 7: OpenMP SIMD Performance for Low Resolution Image

Number of Threads	Read + Setup	Computation Time	Write	Total Time	Time (ms)
1	51	1464	71	1587	1533
2	70	720	64	855	790
4	67	359	101	528	429
8	26	182	64	273	245
16	61	99	62	222	156
32	43	54	94	192	113
64	49	30	62	142	89
128	23	19	59	102	79

Table 8: CUDA Performance

Name	Threads	Read + Setup	Computation Time	Write	Total Time(ms)
High-Resolution	1	206	1207	931	2344
Low-Resolution	1	38	1168	64	1271

Table 9: CUDA SIMD Performance

Name	Threads	Read + Setup	Computation Time	Write	Total Time(ms)
High-Resolution	1	207	1267	895	2370
Low-Resolution	1	22	1134	76	1232

References

[1] M. Spryn, "Improving the Performance of Disparity Map Computation via Parallelization," *Mitchell Spryn's Blog*, Oct. 2020. [Online]. Available: `https://www.mitchellspryn.com/2020/10/18/Improving-The-Performance-Of-Disparity-Map-Computation-Via-Parallelization.html`.

[2] A. Author et al., "Parallel Implementation in a GPU of the Calculation of Disparity Maps for Computer Vision," *IEEE Xplore*, 2018. [Online]. Available: `https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8464217`.

[3] A. Puri, "Disparity Map Stereo Vision GitHub Project," GitHub, 2022. [Online]. Available: `https://github.com/AbhyudayPuri/Disparity_Map_Stereo_Vision`.