Université Libre de Bruxelles

ECOLE
POLYTECHNIQUE
DE BRUXELLES

GPU Computing INFO-H503

Depth estimation
Plane sweep algorithm

Student :
Achten Alexandre 494484

Professor :
Bonatto Daniele
Teaching assistant :
Soetens Eline

May 2025

Academic year 2024-2025

Contents

1 Introduction

Depth estimation is a fundamental problem in computer vision, with applications ranging from robotics and autonomous vehicles to 3D reconstruction and augmented reality. It involves calculating the distance of objects in a scene relative to a camera, often using multiple images taken from different viewpoints.
One effective technique for depth estimation is the plane sweep algorithm, which computes a depth map by projecting input images onto a series of hypothetical planes in 3D space and measuring photoconsistency across them.
The aim of this report is to accelerate the plane sweep algorithm using the parallel architecture of graphics processing units (GPUs). Through a reimplementation of the algorithm for parallel execution, we seek to achieve at least an order of magnitude improvement in speed, and preferably two orders of magnitude, compared to a baseline CPU implementation.
This work focuses on the implementation strategy and design choices guided by iteratively designing and profiling the implementations.

2 Depth estimation

Depth estimation is the process of determining the distance of objects from a camera in a scene, typically using two or more images taken from different viewpoints (stereo matching). By identifying corresponding pixels in these images and measuring their disparity (the difference in position between the images), a depth or disparity map can be created, translating 2D image coordinates into 3D structure (see Figure 1). This technique is used in a variety of fields, including robotics, autonomous vehicles, 3D reconstruction, and augmented reality. While traditionally based on stereo image pairs, recent advances also enable depth estimation from multiple views or even a single image using deep learning methods [2].

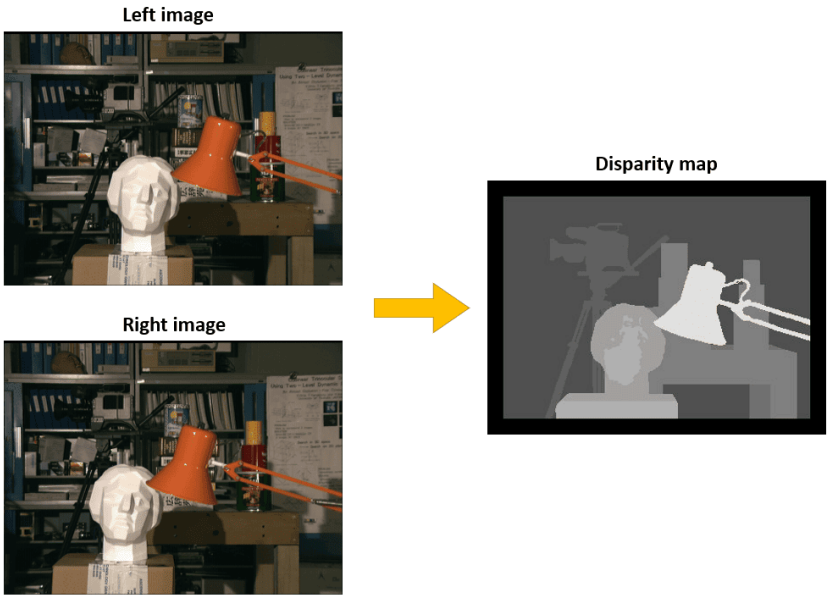


Figure 1: Stereo matching to create a disparity map (depth map)

2.1 Plane sweep algorithm

The plane sweep algorithm is a depth estimation technique that avoids pre-rectifying stereo images by virtually "sweeping" a series of hypothetical planes through the 3D scene. At each depth hypothesis (disparity level), images from different viewpoints are reprojected onto the plane using homographies (mathematical transformation that maps points from one plane to another in projective space). These warped images are compared for photoconsistency, typically using a metric such as SAD (Sum of Absolute Differences), which measures pixel-wise intensity differences between the reference image and reprojected images. This information is compiled into a Disparity Space Image (DSI), which stores consistency scores across multiple depth levels. In this report we will call that the `cost_cube`. The final depth map is computed by selecting the depth with the lowest SAD score at each pixel, indicating the best match. This method is flexible, works with arbitrary camera configurations, and is widely used in multi-view stereo and image-based rendering [2]. A representation of this algorithm is shown at the Figure 2.

The Sum of Absolute Difference (SAD) between two windows is defined as :

$$SAD(w_1, w_2) = \sum_{x,y} \frac{|w_1(x,y) - w_2(x,y)|}{\text{area of window}}$$

Where w_1 is the window from the reference image and w_2 is the window from another view.

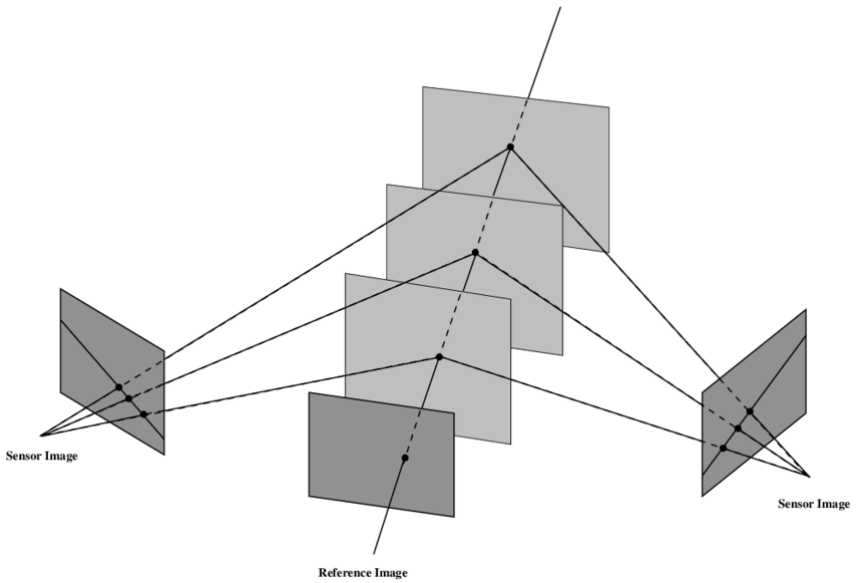


Figure 2: The Plane-Sweep algorithm. A plane is swept along the optical axis of the reference image. For every plane position the points in the reference image are transformed to the sensor views. For every pixel the depth that provides the lowest dissimilarity measure is retained [1]

Algorithm 1 Plane Sweep Stereo via Cost Volume Accumulation

Require: Reference camera C_{ref}, camera set $\{C_i\}_{i=1}^{N}$, window size w
Ensure: Cost volume $\mathcal{C}(x, y, d)$ of size $H \times W \times D$

Initialize cost volume $\mathcal{C}$ to large values
for all camera C_i in camera set, excluding C_{ref} **do**
  for $d = 0$ to $D - 1$ **do**
    Compute plane depth $z_d = \frac{z_{\text{near}} z_{\text{far}}}{z_{\text{near}} + \frac{d}{D}(z_{\text{far}} - z_{\text{near}})}$
    for $y = 0$ to $H - 1$ **do**
      for $x = 0$ to $W - 1$ **do**
        Back-project pixel (x, y) to 3D point $\mathbf{P}_{\text{world}}$ using C_{ref} intrinsics and extrinsics
        Project $\mathbf{P}_{\text{world}}$ to image coordinates (x', y') in C_i
        Initialize cost accumulator: $c \leftarrow 0$, count $n \leftarrow 0$
        for $\Delta x, \Delta y \in \{-\lfloor w/2 \rfloor, \ldots, \lfloor w/2 \rfloor\}$ **do**
          if $(x + \Delta x, y + \Delta y)$ and $(x' + \Delta x, y' + \Delta y)$ are within image bounds **then**
            $c \mathrel{+}= \sum_{ch \in \{Y,U,V\}} \left| I_{\text{ref}}^{ch}(x + \Delta x, y + \Delta y) - I_i^{ch}(x' + \Delta x, y' + \Delta y) \right|$
            $n \mathrel{+}= 1$
          end if
        end for
        if $n > 0$ **then**
          $c \leftarrow c/n$ {Divide by area of window}
          $\mathcal{C}(x, y, d) \leftarrow \min(\mathcal{C}(x, y, d), c)$ {Minimum of each camera}
        end if
      end for
    end for
  end for
end for
return $\mathcal{C}$

The pseudocode is presented at the Algorithm 1. Several parts can be parallelized and they will be explored in the following sections.

2.2 Graph-cut algorithm

The Graph-cut algorithm is an optimization technique used to solve discrete energy minimization problems. Given a set of pixels $\mathcal{P}$ and a finite label space $\mathcal{L}$, the objective is to assign a label $l_p \in \mathcal{L}$ to each pixel $p \in \mathcal{P}$ such that the following energy function is minimized:

$$E(\mathbf{l}) = \sum_{p \in \mathcal{P}} D_p(l_p) + \sum_{(p,q) \in \mathcal{N}} V_{p,q}(l_p, l_q)$$

Where $D_p(l_p)$ is the data term measuring how well the label fits the observation at pixel p, and $V_{p,q}(l_p, l_q)$ is the smoothness term enforcing spatial consistency between neighboring pixels p and $q \in \mathcal{N}$.

The problem is reformulated as a flow network $G = (V, E)$, with nodes corresponding to pixels and edges encoding the energy terms. A minimum cut on this graph

partitions the nodes into two disjoint sets such that the total edge weight between the sets is minimized. This cut corresponds to a label assignment that minimizes the total energy.

3 Experimental setup

3.1 Computational resources

Name	Value
CPU Name	Intel(R) Core(TM) i7-1065G7
CPU Clock Rate	1.50 GHz
#Cores	4
#Logical Cores	8
Cache Size L1	320 KB
Cache Size L2	2 MB
Cache Size L3	8 MB
Operating System	Windows 11 Family 24H2
Main Memory Size	15.8 GB
Main Memory Clock Rate	3200 MHz

Table 1: CPU specifications

Name	Value
Device Name	NVIDIA GeForce MX330
Compute Capability	6.1
Max Threads per block	1024
Max Threads per Multiprocessor	2048
Shared Memory per block	48 KB
Shared Memory per Multiprocessor	96 KB
Registers per Block/Multiprocessor	65536
Max Grid Size	[2147483647 65535 65535]
Max Block Dimensions	[1024 1024 64]
Half Precision FLOP/s (FP16)	9.564 GFLOP/s
Single Precision FLOP/s (FP32)	1.224 TFLOP/s
Double Precision FLOP/s (FP64)	38.256 GFLOP/s
#SSM	4
#Multiprocessor (SM)	3
Threads per Warp	32
#Cores	384
Processor Clock Rate	1.594 GHz
Global Memory Bandwidth	56 GB/s
Global Memory Size	2 GB
Constant Memory Size	64 KB
L2 Cache Size	512 KB
Memory Clock Rate	3.5 GHz

Table 2: GPU specifications

3.2 Experiment

Different implementations of the algorithm will be compared using the same four images (using one of them as the reference). The quality of the depth estimation is evaluated manually by comparing the results with those of the CPU implementation. The implementations are compared based on their execution time (in seconds), total execution timeand the number of floating-point operations (in GFLOPs). This is presented in Section 5. Figure 3 shows an example of the image used.

Figure 3: Image from the reference camera (1920 × 1080 pixels)

3.3 Metrics

The metrics used to compare the algorithms are the execution time of the kernel and the total execution time, including the pre-processing steps. When profiling the kernels, severa metrics are used, they are defined as follows:

$$\text{Occupancy} = \frac{N_{\text{active warps/SM}}}{N_{\text{maximum active warps/SM}}}$$

$$\text{Memory Throughput/Bandwidth} = \frac{\#\text{bytes read/written per kernel}}{\text{runtime for the kernel}}[GB/s]$$

$$\text{Computation Throughput/Operation Per Second} = \frac{\#\text{operation done per kernel}}{\text{runtime for the kernel}}[GOPS]$$

With the definition of memory and computation throughput, we can define the compute intensity as :

$$\text{Computational Intensity} = \frac{\text{Computation Throughput}}{\text{Memory Throughput}}[OPS/byte]$$

The kernel performance can be plotted on the roofline model using the computational intensity and computation throughput.

4 Implementations

4.1 Naive CPU

The CPU version of the algorithm strictly adheres to the pseudocode presented in Algorithm 1. It is important to note that the algorithm only uses one channel of

the coloured image: the intensity channel. Its performance is shown and compared in Table 3. The resulting depth map from the algorithm is shown in Figure 4.

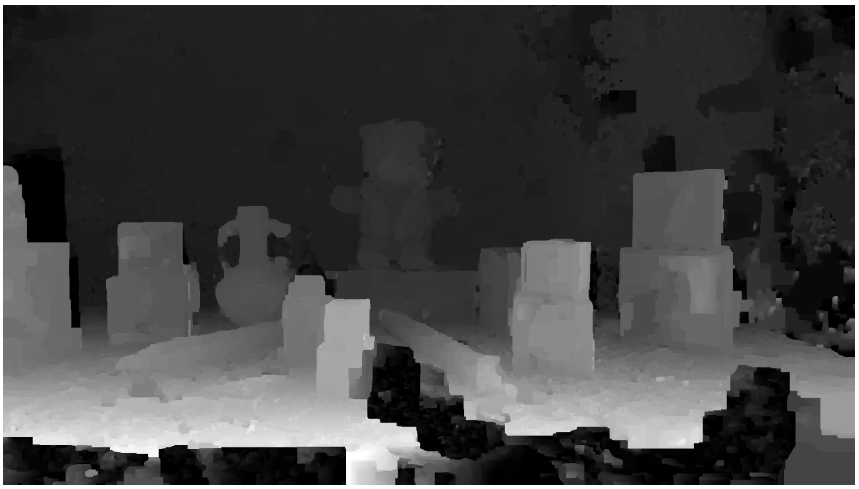


Figure 4: Depth map from the CPU implementation using graphcut optimization

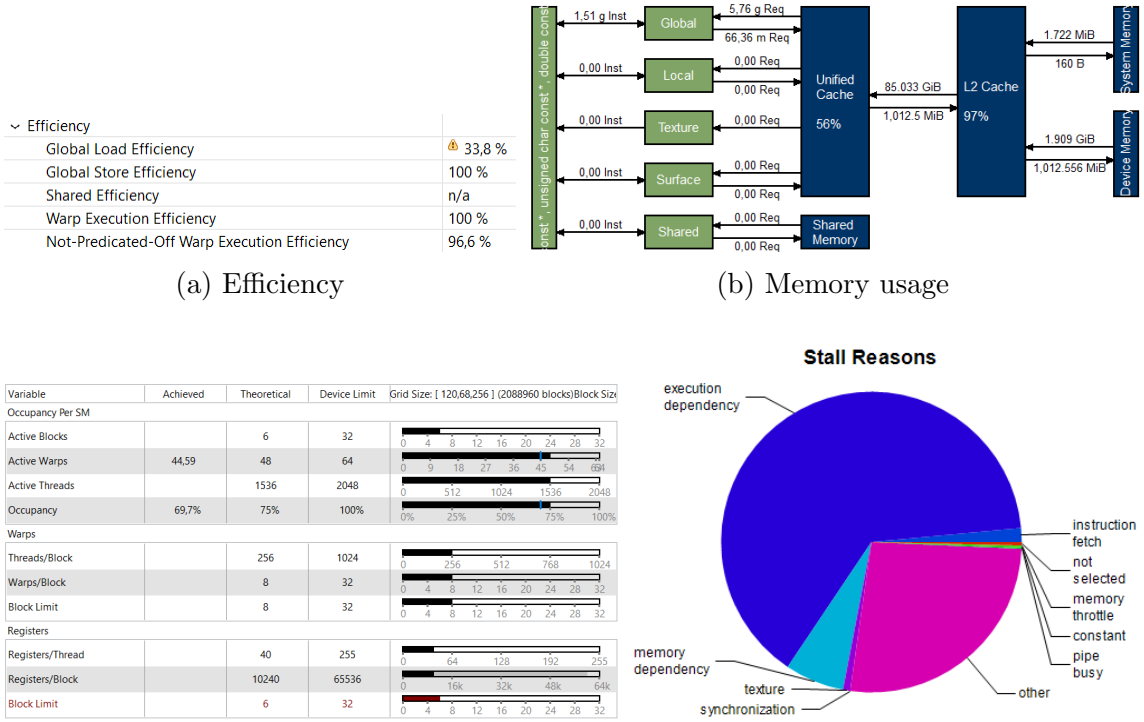
4.2 Naive GPU

This implementation uses a kernel with a grid size of [120,68,256] (corresponding to the size of the cost cube and using the formula $N_{\text{blocks}} = \frac{N+\text{block size}-1}{\text{block size}}$) with block size of [16,16]. A block size of [32, 32] was also tested, but it yielded poorer performance, likely due to the limited number of registers available per thread when using larger blocks. Each thread in this kernel computes the window associated with a single pixel (one thread per pixel of the output). The kernel is launched once for each camera image that is different from the reference image. The GPU implementation involves more pre-processing steps compared to the CPU version, therefore the total execution time is reported in Table 3, in addition to the kernel performance alone.

At this stage, the total amount of memory used on the device is given by images + cost cube + camera parameters = 8 294 400 B + 2 123 366 400 B + 1 248 B = 2.1 GB. From Table 2, we know that the global memory size is only 2 GB, making kernel profiling really bad due to the reliance on main memory (RAM).
To address this issue, the cost cube was encoded using `__half` types instead of `float`. This data type uses only 2 bytes and has a representable range of $\pm 6.1 \times 10^{-5}$ to $\pm 65\,504$. In our case, the values do not exceed 255, and we do not require more than four decimal digits of precision.
The first implementation was slow because all calculations were performed using FP16 operations, whereas the GPU is optimized for FP32 operations. As a result, all computations were performed in `float` and only converted to `__half` for storage.

In any case, the naive version still showed better performances than the CPU version. The profiling metrics[1] are shown at Figures 5.

[1]The Visual Profiler program could not give the performance limiter, memory and some compute metrics on this kernel.



(a) Efficiency (b) Memory usage

(c) Latency (d) Stall reasons

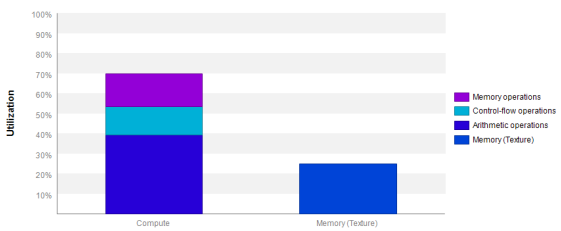
Figure 5: Charts from the profiler on GPU naive kernel

Figure 5 shows that there are several areas for improvement: poor global load efficiency due to uncoalesced access (use of windows); 85 GB of load between L1 and L2 indicating poor cache hit and poor data locality; occupancy at 69% of the theoretical maximum; and no use of shared memory. The profiler could not identify the performance limiter, but we are certainly memory bound. As can be seen from the roofline model in Figure 11, we are in the memory-bound region.

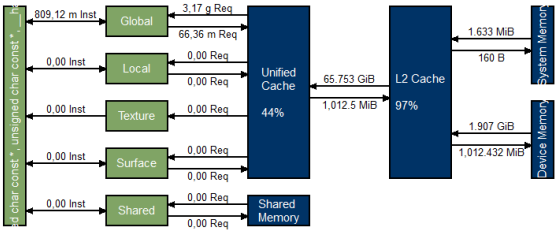
The poor load efficiency is undoubtedly due to access to camera images. Similarly, the high traffic between L1 and L2 is most likely due to the images and parameters of the cameras.

4.3 GPU parameters optimized

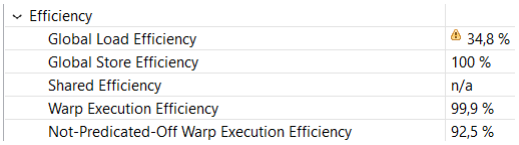
In this implementation, performance was improved by storing the cameras parameters in `constant` memory and switching from the `double` type to `float`. Since the camera parameters are accessed frequently, these changes had a significant impact. As shown in Table 2, computations using `float` are faster than those using `double`, which further contributed to the speedup. Additionally, the use of `constant` memory improved memory access efficiency between the L1 and L2 caches. Several minor code optimizations were also applied to enhance overall performance.



(a) Kernel performance limiter

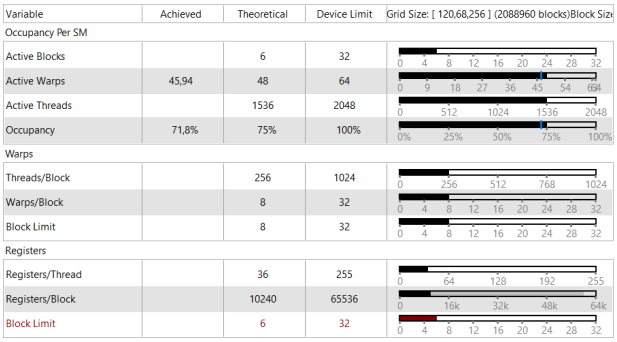


(b) Memory usage



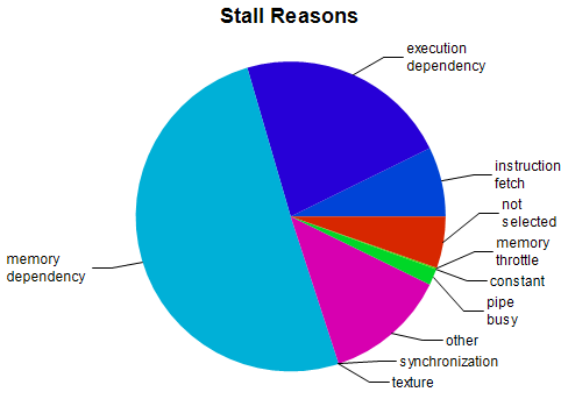
Efficiency	
Global Load Efficiency	34,8 %
Global Store Efficiency	100 %
Shared Efficiency	n/a
Warp Execution Efficiency	99,9 %
Not-Predicated-Off Warp Execution Efficiency	92,5 %

(c) Efficiency



Variable	Achieved	Theoretical	Device Limit	Grid Size: [120,68,256] (2088960 blocks)Block Size
Occupancy Per SM				
Active Blocks		6	32	
Active Warps	45,94	48	64	
Active Threads		1536	2048	
Occupancy	71,8%	75%	100%	
Warps				
Threads/Block		256	1024	
Warps/Block		8	32	
Block Limit		8	32	
Registers				
Registers/Thread		36	255	
Registers/Block		10240	65536	
Block Limit		6	32	

(d) Latency



(e) Stall reasons

Figure 6: Charts from the profiler on GPU parameters optimized kernel

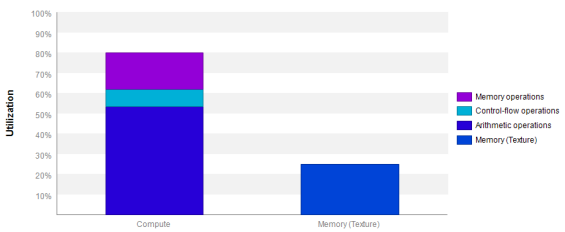
Figure 6 illustrates several minor enhancements in comparison to Figure 5. The occupancy is slightly higher, and the amount of data transferred between the L1 and L2 caches is reduced (from 85GB to 65GB). The primary stall reason is no longer execution dependency, but memory dependency. It indicates that there is room for improvement for the memory. The global load efficiency has increased slightly (33% to 34%). From Figure 6a we are compute bound, but it is difficult to reduce the computations. Overall, the performance of this implementation is better than the naive implementation as shown in the Table 3.

There are still some problems to solve, such as improving the global load efficiency. We could try to improve the hit rate and solve the poor data locality by using shared memory.

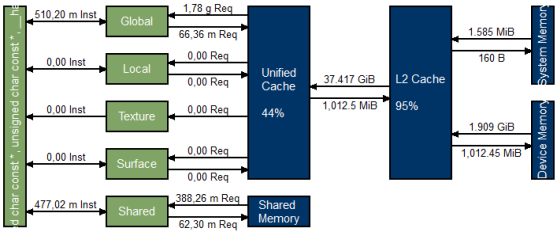
4.4 GPU shared

The first thing noticeable in the plane sweep algorithm, is the use of windows to compute the absolute differences. This is similar to the 2D convolution algorithm already seen in class. The use of shared memory could solve the load issues and the cache hit rates.

The shared memory is loaded by each block, including the borders too. The way of loading the shared memory is detailed at Figure 8. Although this method appears to have many conditions (potential divergence), it still outperforms the previous implementation.



(a) Kernel performance limiter



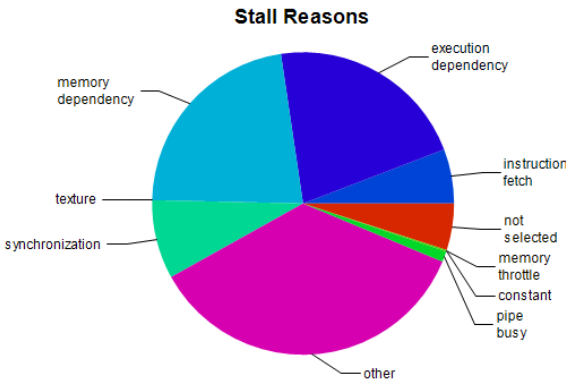
(b) Memory usage

Efficiency	
Global Load Efficiency	32,4 %
Global Store Efficiency	100 %
Shared Efficiency	22,9 %
Warp Execution Efficiency	96,9 %
Not-Predicated-Off Warp Execution Efficiency	92,2 %

(c) Efficiency

Variable	Achieved	Theoretical	Device Limit	Grid Size: [120,68,256] (2088960 blocks)Block Size
Occupancy Per SM				
Active Blocks		8	32	
Active Warps	61,18	64	64	
Active Threads		2048	2048	
Occupancy	95,6%	100%	100%	
Warps				
Threads/Block		256	1024	
Warps/Block		8	32	
Block Limit		8	32	
Registers				
Registers/Thread		32	255	
Registers/Block		8192	65536	
Block Limit		8	32	
Shared Memory				
Shared Memory/Block		400	98304	
Block Limit		192	32	

(d) Latency



(e) Stall reasons

Figure 7: Charts from the profiler on GPU parameters optimized kernel

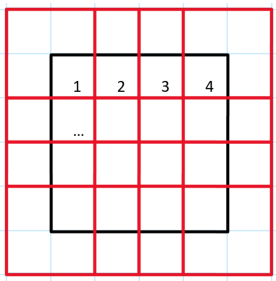


Figure 8: Shared memory is loaded for a block (black) by each thread associated to a pixel (each square). Depending on its position in the block, each thread loads a different number of pixels (red area). In the real implementation blocks are [16, 16]

The sweep plane algorithm uses a window through the reference image to try to project pixels from the other cameras. When trying to implement shared memory, the problem encountered is that, due to projection, it is impossible to know which pixels of the camera image will be used to compare. This pseudo-random access to camera images makes it difficult to implement shared memory over them. Therefore, this implementation only uses shared memory for the reference image.

Even with only the reference image in shared memory, significant performance improvements are still possible, as shown in Figure 7. Significantly better performance could be achieved if a solution was found to put all the camera images in shared memory.

Figure 7d indicates a better occupancy of 95%, and in the Figure 7b the load throughput between L2 and L1 has been halved (65GB to 37GB) thanks to the reference image that was placed in shared memory. Figure 7a shows that we are compute-bounded, as utilisation peaks at 80%. We can also see that control-flow operations have improved due to other optimisations in the code. Execution times are presented in Table 3.

Improvements involving the camera images include putting them in the texture memory, but the compiler already does this automatically. Other ways of loading the shared memory will be explored.

4.5 GPU shared 9 blocks

This implementation involved loading more blocks into the shared memory. This implementation still uses shared memory only for reference image. As shown in Figure 9, each thread will fill 9 pixels in the shared memory.

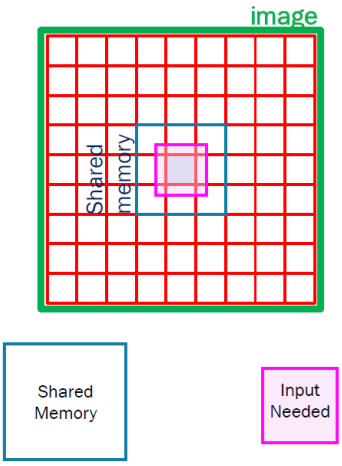


Figure 9: Loading 9 blocks in shared memory. Blocks (red) over an image (green), input needed for the window (rose), shared memory (blue).

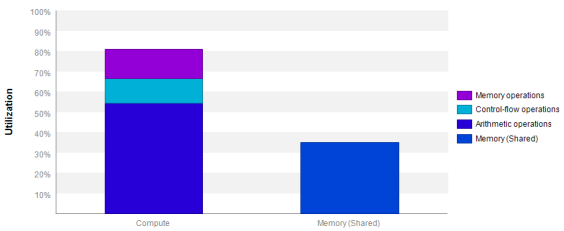
As this implementation showed no improvement on the previous one, the profiler charts will not be displayed. This lack of improvement is probably due to the fact that the borders of the adjacent blocks still needed to be loaded from global memory.

4.6 Pre-processing optimization

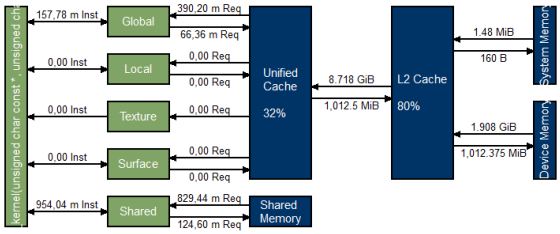
In this implementation, the kernel GPU shared was used and remained unchanged, but the pre-processing steps were optimised. This led to improvements in total time, rather than in the kernel itself. These improvements were achieved simply by copying the `cv::Mat` data directly into the global memory, rather than flattening all the images beforehand. It's performance is showed at Table 3.

4.7 GPU shared all

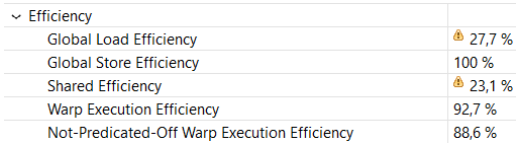
To maximise the use of shared memory, the camera images were included alongside the reference image. Although it seemed difficult, it simply relies on a hypothesis and calculations based on the projection indices, which were calculated prior to loading the shared memory. Proceeding in this way enabled to determine which part of the image to load into the shared memory. This method relies on the assumption that the projected pixels are not 'too far' apart. This assumption is, in fact, close to reality, so this implementation significantly improved the kernel's speed. The shared memory was loaded using the same technique used for the GPU shared implementation (see Figure 8). The performance results are shown in the Table 3.



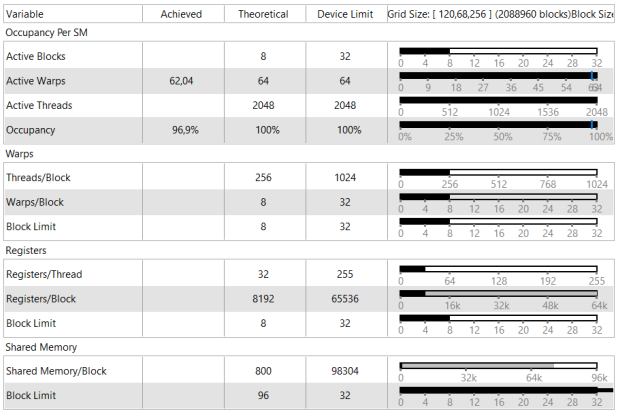
(a) Kernel performance limiter



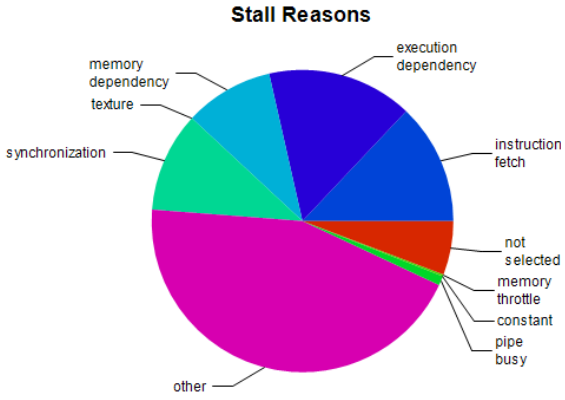
(b) Memory usage



(c) Efficiency



(d) Latency



(e) Stall reasons

Figure 10: Charts from the profiler on GPU parameters optimized kernel

Figure 10 shows the significant improvements of this implementation compared to the previous best implementation, GPU Shared (Figure 7). The load between L1 and L2 has decreased significantly (from 37 GB to 8 GB) and the occupancy has increased (from 95% to 96%). Although the shared efficiency remains low, this is not due to bank conflicts. The speed of the shared memory masks this low efficiency. The remaining 8GB must be due to the load of the images and the load of the `cost_cube` to compare and store the minimum value.

5 Comparison

All the implementations reached the same resulting depth map than the CPU implementation presented at Figure 4. The kernel performances from the previous sections are presented in the Table 3.

Implementation	Kernel Time (s)	Total Time (s)	GFLOP/s	Speedup
CPU naive	323.603	323.603	1.019	1
GPU naive	9.591	10.139	32.529	31.91
GPU param opt	3.250	4.283	75.485	66.67
GPU shared	2.213	3.149	108.928	102.76
GPU shared 9 blocks	3.115	4.107	80.096	78.79
GPU processing opt	2.213	2.948	108.928	109.77
GPU shared all	**1.760**	**2.680**	**122.111**	**120.75**

Table 3: Comparison of implementations

The speedup in the Table 3 is speedup $= \frac{\text{Total time CPU}}{\text{Total time kernel}}$. The total time represents the kernel execution time and the pre-processing steps. The GFLOP/s are the addition of every floating point operation (FP16, FP32, FP64) divided by the execution time.

The different implementations have been plotted on the theoretical roofline model (see Figure 11) to determine whether they are compute- or memory-bound. Unlike the other charts presented, it appears that all the implementations are memory bound. The roofline model was established by taking into account only the single precision floating-point operation (FP32) and the total global throughput (load + store).

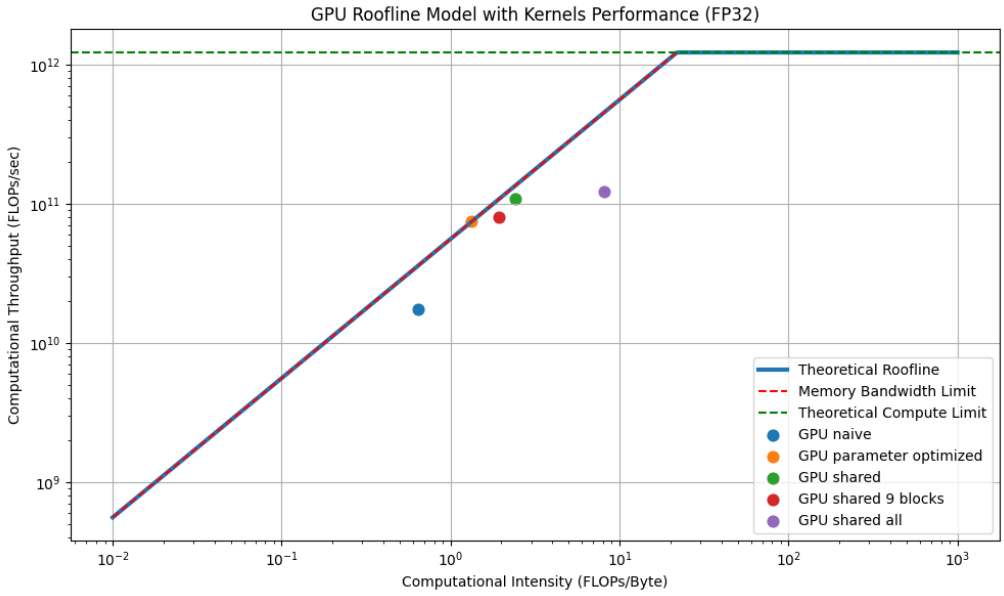


Figure 11: Roofline model

The all shared GPU implementation is the best so far, but there is still room for improvements. These ideas will not be explored in this work, but it could be : processing multiple camera images at once by changing the block size to [16 16 3], where 3 indicates the number of other cameras; Trying to use different streams and asynchronous `cudaMalloc` to launch kernels in parallel while still allocating memory (not possible on the GPU presented at Table 2); Padding the image in the height to make 1080 a multiple of 32 for better cache line loads; Use simpler functions than `fminf`, `fabsf`, `roundf`; Use of shared memory for the `cost_cube`; Padding the images to avoid flow control operations, ...

6 Conclusion

This report presented a thorough exploration and optimization of the plane sweep algorithm for depth estimation using GPU acceleration.
The best implementation achieved a speedup of over 120× compared to the CPU baseline, with significant gains execution time. Profiling further revealed that while compute performance improved substantially, memory bandwidth remains a limiting factor in current implementations.
Despite these advancements, there is still room for further enhancement. Further works could explore multi-stream execution, or alternative memory layouts to improve data locality. Overall, this project successfully validates the potential of GPU acceleration in depth estimation tasks.

References

[1] Theo Moons, Maarten Vergauwen, and Luc Van Gool. “3D reconstruction from multiple images”. In: (2008).

[2] Richard Szeliski. *Computer Vision - Algorithms and Application Second Edition*. Springer Nature Switzerland, 2022.