Parallelization of Mean Shift for Image Segmentation Term Project, CSC 746, Fall 2023

Duccio Rocca[*]
SFSU

Abstract

This paper presents an in-depth investigation into the parallel implementation of the Mean Shift algorithm using both OpenMP and CUDA, with a focus on enhancing performance and addressing the computational intensity inherent in image segmentation. As the algorithm's real-time applicability faces challenges, our study aims to scrutinize and optimize the performance of both implementations.

1 Introduction

The Mean Shift algorithm, renowned for its efficacy in image segmentation, is hampered by computational intensity, restricting its applicability. Current implementations struggle to strike the right balance between performance and computational demand. In this paper, we tackle the challenge of optimizing the Mean Shift algorithm's computational intensity through a parallel implementation using OpenMP and CUDA.

The intrinsic computational intensity of the Mean Shift algorithm poses a significant hurdle, limiting its segmentation capabilities. Existing solutions grapple with the trade-off between accuracy and speed.

We aim to measure the performance of our parallel implementations of the Mean Shift algorithm, leveraging OpenMP and CUDA. We hypothesize that our implementation will achieve a significant speedup over existing solutions, while maintaining accuracy.

To provide context, Section 2 explores related work, highlighting the attempts and challenges in parallelizing the Mean Shift algorithm. In Section 3, we present the intricacies of our parallel implementations, employing OpenMP and CUDA. The results of our performance evaluation are detailed in Section 4, showcasing the advancements achieved in computational intensity. Finally, Section 5 encapsulates our conclusions and outlines potential avenues for future research in optimizing computationally intense image segmentation algorithms.

2 Related Work

Image segmentation entails the identification of pixel groups exhibiting cohesive characteristics [12]. This challenge finds its statistical counterpart in cluster analysis, a extensively researched domain boasting a myriad of algorithms.

Within computer vision, image segmentation emerges as a longstanding and extensively explored problem. Initial methods delved into strategies such as region splitting or merging, aligning with the divisive and agglomerative algorithms prevalent in clustering literature. Contemporary algorithms, in contrast, frequently prioritize the optimization of global criteria, including intra-region consistency, inter-region boundary lengths, and dissimilarity.

Segmentation involves dividing a digital image into multiple segments, intending to simplify and/or alter the image's representation for enhanced meaning and analysis [7].

[*]email:drocca@sfsu.edu

Typically employed for object and boundary localization in images, image segmentation yields a collection of segments encompassing the entire image or a set of extracted contours. Various techniques have been devised to achieve image segmentation, including thresholding, clustering, edge detection, and region growing, among others.

A well known clustering algorithm used for image segmentation is the Mean Shift (MS) algorithm. MS is a characteristic vector-based clustering algorithm that can be described as an unsupervised classification and global/local optimization algorithm without a priori knowledge [6].

Mean shift was introduced in Fukunaga and Hostetler [4]. It was originally developed for the problem of mode finding in probability density estimation but its application has been extended to be applicable in other fields like Computer Vision [3].

However, the mean shift algorithm is computationally intensive, exhibiting a time complexity of $O(Tn^2)$, where T represents the number of iterations for processing each data point, and n is the total number of data points in the dataset—such as the number of pixels in an image. This results in prohibitively long processing times, particularly for large images.

To mitigate this challenge, various parallel implementations of the mean shift algorithm have been proposed.

To overcome the efficiency problem, early examples of parallelization of serial version of mode detection algorithm have made use of high-performance computing (HPC) techniques [1]. In the field of cell segmentation in biocial imagining a parallel version of the mean shift agorithm was proposed. [9] implemented a version that make use of graphic processing units (GPU), which resulted in significant speedup over the serial implementation.

To parallelize the serial algorithm, [5] implemented a parallel mean shift image segmentation algorithm using the MPI library on a four-node computer cluster. In that study, they focused mainly on the decomposition method with a single RS image.

[10] introduced the first parallel implementation of MS accelerated using Improved Fast Gauss Transformation (IFGT). They based their implementation on a many core GPU platform and showed that they could achieve a considerably faster segmentation execution compared with alternative CPU. [7] proposed a parallel mean shift image segmentation algorithm using CUDA. For each pixel, one CUDA thread was created without exploring distribution strategies for threads. This resulted in a relatively modest speedup (up to 21.46).

Alternative CUDA implementations [11] achieved a higher speedup, reaching up to 30, by employing parallel mean shift on RS images. In another work, Huang et al. [6] parallelized the mean shift algorithm on a heterogeneous platform comprising CPU + GPU using Thread Building Blocks and CUDA, achieving a remarkable speedup of 55.23.

Other implementation used hybbrid models, like [13] that used MPI + OpenMP + CUDA to accelerate high-resolution molecular dynamics simulations of proteins on an eight-node GPU cluster.

To overcome the limitations of portability other researcher focused on open source implementations. [8] proposed a programming framework for CPU/ GPU using MPI + OpenCL hybrid programming.

What emerges from this literature review is that enhancing the efficiency of the image segmentation algorithm based on serial mean shift through optimization is unlikely to yield significant improvements. Instead, a more promising approach would involve parallelization.

Although challanges persist in the development of parallel algorithms, the use of parallel computing is becoming increasingly popular in the field of image segmentation. With this work we aim to contribute to this trend by proposing two parallel implementations of the mean shift algorithm using the OpenMP library and CUDA.

The bbase of our work is the code developed by [2] that we parallelized using OpenMP and CUDA. We then compared the performance of the three implementations (serial, OpenMP and CUDA) on a single node with a GPU and on the multicore CPU offered by the Perlmutter supercomputer at NERSC.

3 Implementation

In this section, we detail the implementation aspects of our approach for parallelization of image segmentation, leveraging the Edge Detection and Image Segmentation (EDISON) system for discontinuity-preserving filtering and segmentation.

The segmentation component of EDISON was utilized within a custom implementation to achieve the desired results. Additionally, the implementation makes use of the CImg Library, a modern C++ toolkit for image processing, to facilitate efficient image manipulation.

The part that was parallelized is the Mean Shift algorithm, which is the core of the EDISON system. This is a crucial part of the whole process called filtering.

The main.cpp file orchestrates the entire process. It reads the input image, calls the filtering function, and writes the output image. The parameters for segmentation, including spatial bandwidth (sigmaS), range bandwidth (sigmaR), and minimum area (minArea), are set based on user input. The input image is loaded into the system using the CImg Library. The image is expected to be in grayscale.

The mean shift segmentation is executed on the loaded image using the specified parameters and implementation strategy. The implementation supports various strategies, such as serial, parallel (OpenMP), GPU (CUDA).

The segmentation process yields regions, which are encapsulated in a ResultWrapper object. This object contains information about the segmented regions, including the output image and region borders.

The pixel values in the original image are then substituted with the mean pixel value of the corresponding region, resulting in a segmented image. Borders of the segmented regions are drawn on the original image to enhance visual representation.

In addition to leveraging existing tools, a custom wrapper was implemented to extract relevant results and facilitate the integration of images into the workflow. This wrapper serves as a bridge between the EDISON system and our custom image processing pipeline.

The ResultWrapper class serves as a container for storing the results of the segmentation process. It contains the segmented pixels, the borders of the segmented regions, and the number of regions. The ResultWrapper object is then initialized and populated with relevant information, including region boundaries and the segmented image.

The execution times for filtering and region fusion are recorded to provide insights into the computational efficiency of our segmentation approach.

The core of the segmentation process is contained in the *msImageProcessor.cpp* which contains the Filter function. This function is responsible for the mean shift segmentation of the input image.

There are three different implementations of the Filter function, each of which is responsible for a different execution strategy. The serial implementation is the baseline, while the parallel and GPU implementations are the parallelized versions of the serial implementation.

3.1 Serial MS

The serial implementation of the mean shift algorithm serves as a reference. This implementation comes from the EDISON system and is used as a baseline for comparison with the parallel and GPU implementations. The serial implementation is contained in the $ms_filter_serial.cpp$ file.

The input data, representing the image to be segmented, is preprocessed for mean shift filtering. The image dimensions and bandwidth parameters (sigmaS and sigmaR) are initialized. The data is then scaled and organized into buckets for efficient search operations.

The input image data is scaled based on the specified bandwidths (sigmaS and sigmaR). For grayscale images, each pixel's coordinates and intensity values are scaled accordingly.

A bucketing mechanism is employed to efficiently index and locate neighboring pixels during the mean shift process. The image space is divided into discrete buckets, and each pixel is assigned to a specific bucket based on its scaled coordinates and intensity.

The mean shift algorithm is applied to each pixel in the image to iteratively shift its position until convergence.

The mean shift vector (Mh) is initialized for each pixel. A lattice structure is used to define the window center (yk), initially set to the pixel's coordinates and intensity.

A search window is defined around the current window center. Neighboring pixels within the search window contribute to the mean shift vector based on their distances and intensity differences.

Contributions from neighboring pixels are weighted based on their proximity and intensity dissimilarity. The weighted sum of pixel values is accumulated to compute the mean shift vector.

The magnitude of the mean shift vector is then evaluated to determine convergence. Iterative adjustments to the window center continue until convergence or a predefined limit (LIMIT) is reached.

3.2 OpenMp MS

For this implementation we used OpenMP to parallelize the serial implementation. The parallelized version of the mean shift algorithm (contained in the file $ms_filter_omp.cpp$) leverages parallelization to enhance performance. This section provides an overview of the key aspects of the OpenMP implementation.

The primary goal of parallelization is to exploit multicore architectures for concurrent execution, thereby accelerating the mean shift process. OpenMP directives and constructs are utilized to distribute the workload among multiple threads.

Data preparation remains consistent with the serial version, involving scaling, bucketing, and organization of the input data. However, OpenMP introduces parallelism in the subsequent pixel-wise processing.

The processing of pixels is parallelized to improve the overall efficiency of the mean shift algorithm.

The OpenMP parallel region is introduced to spawn multiple threads. Each thread independently processes a subset of pixels in parallel.

Memory allocations for variables such as yk and Mh are performed within the thread-specific scope to avoid data race conditions.

The main loop over pixels is parallelized using the OpenMP for directive. The workload is dynamically distributed among threads for load balancing.

The whole loop is mainly reading from the input image and there is only one write operation at the end of the loop.

A critical section is introduced to ensure that only one thread updates the shared msRawData array at a time as it is shown in Listing 2.

This avoids potential race conditions and ensures correct results.

The LIKWID Marker API is integrated into the OpenMP code to facilitate performance monitoring. Markers are registered, started, and stopped within the parallel region, providing insights into the execution time of specific code segments.

The parallelization strategy aims to maximize computational efficiency by efficiently distributing the workload among threads. Load balancing is achieved through dynamic scheduling of tasks, ensuring that each thread contributes effectively to the overall computation.

In summary, the OpenMP implementation of the mean shift algorithm capitalizes on parallel processing to accelerate the segmentation of images. The combination of efficient data organization, and parallel pixel processing contributes to the overall effectiveness of the parallelized algorithm.

3.3 CUDA MS

The GPU implementation of the mean shift algorithm (contained in the file $ms_{f}ilter_{c}uda.cu$) leverages the parallel processing capabilities of GPUs to accelerate the segmentation process. This section provides an overview of the key aspects of the CUDA implementation.

The core of the CUDA implementation is the msImageProcessorKernel kernel, which is responsible for processing individual pixels in parallel. Key features of the CUDA kernel include:

The kernel is launched with a grid of blocks, each containing multiple threads. Each thread processes one pixel, and the grid covers all pixels in the image. Threads calculate the mean shift vector for their assigned pixel. A predefined set of neighbors (*bucNeigh*) assists in accessing neighboring buckets. The mean shift vector is computed based on the neighboring pixels, considering their weights.

The host code handles memory allocations, data transfer between the host and device, and kernel invocation. Memory space on the GPU device is allocated for input data ($d_{s}data$), buckets ($d_{b}uckets$), weight map ($d_{w}eightMap$), neighbor list ($d_{s}list$), and output data ($d_{m}sRawData$). Input data and related parameters are copied from the host to the device memory using cudaMemcpy. The CUDA kernel is then launched with an appropriate grid and block configuration as shown in Listing 1

Execution time of the CUDA kernel is measured using high-resolution timers (std::chrono) to evaluate the performance of the GPU-accelerated mean shift algorithm.

The final segmented image (msRawData) is copied from the device memory to the host memory. Finally allocated device memory is freed to prevent memory leaks.

The implementation includes error-checking mechanisms (gpuErrchk) to identify and handle potential CUDA errors. Proper error handling ensures robustness and helps diagnose issues during development and execution.

The CUDA implementation of the mean shift algorithm demonstrates the potential for significant speedup in image segmentation tasks, particularly on GPUs with parallel processing capabilities. The efficient utilization of GPU resources and the parallel nature of the algorithm contribute to improved computational performance.

4 Evaluation

In this section we present the results of our experiments. We begin by describing the experimental setup, including the hardware and software environment, and the methodology used to evaluate our solution. We then present the results of our experiments, including the performance of our solutions and a comparison among the performance of all solutions. We conclude with a discussion of the results, including the context and limitations of our solution.

```
// Launch the CUDA kernel

const int threadsPerBlock = 1024;
const int numberBlocks = (L + threadsPerBlock
- 1) / threadsPerBlock;

msImageProcessorKernel<<<numberBlocks,
threadsPerBlock>>>
(d_sdata, d_buckets, d_weightMap, d_msRawData,
 d_slist, hiLTr, nBuck1, nBuck2, nBuck3, width
, height, d_bucNeigh, sMins, sigmaS, sigmaR);
```

Listing 1: The threads are computed to have one per pixel. The kernel is launched with all the device copies of the data structures.

```
#pragma omp critical
{
    // store result into msRawData...
    msRawData[i] = pixValue;
}
```

Listing 2: The critical region is set to avoid race conditions when writing to the msRawData array.

4.1 Computational platform and Software Environment

Our experimental evaluations were carried out on a computing system[1] equipped with dual AMD EPYC 7763 (Milan) CPUs, each featuring 64 cores. These CPUs support the AVX2 instruction set and incorporate specific cache configurations: 32 KB L1 cache and 512 KB L2 cache per core, accompanied by a shared 256MB L3 cache per Core Complex Die (CCD). The system features 512 GB of DDR4 memory, providing a memory bandwidth of 204.8 GB/s per CPU. Operating under SUSE Linux Enterprise Server 15 SP4 and utilizing g++ (GCC) 11.2.0 20210728 (Cray Inc.) compiler with full optimization, this environment serves as the foundation for our performance assessments.

Additionally, the computing system on which we ran the experiments extends its capabilities to incorporate GPU nodes, offering two different configurations. The first configuration consists of 1 AMD EPYC 7763 CPU coupled with 4 NVIDIA A100 (40GB)

[1]Documentation available at URL: https://docs.nersc.gov/systems/perlmutter/architecture

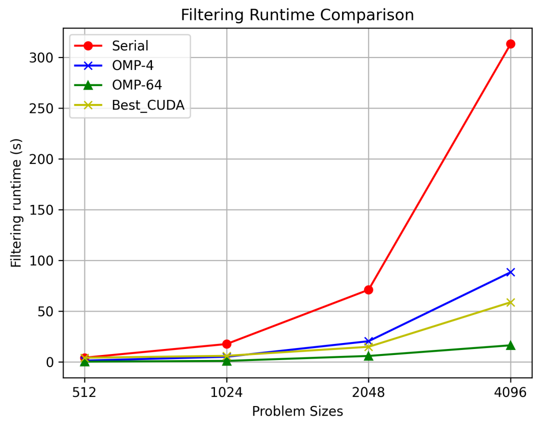

Figure 1: The runtime of the filtering step for different implementations.

GPUs. These GPUs feature PCIe 4.0 GPU-CPU and NIC-CPU connections, ensuring efficient data transfer and computation. Each GPU node in this configuration comes equipped with 256 GB of DDR4 DRAM and 40 GB of High Bandwidth Memory (HBM) per GPU, operating at an impressive GPU memory bandwidth of 1555.2 GB/s, while maintaining a CPU memory bandwidth of 204.8 GB/s. Furthermore, 12 third-generation NVLink links are established between each pair of GPUs, offering a substantial data transfer rate of 25 GB/s in each direction for every link.

The second GPU node configuration aligns with the previous setup, featuring 1 AMD EPYC 7763 CPU and 4 NVIDIA A100 GPUs. However, these GPUs offer larger memory capacity, providing 80GB of HBM per GPU. This expanded memory capacity is paired with the same robust PCIe 4.0 connections and network interface cards (NICs), ensuring seamless integration with the CPU and exceptional data throughput.

4.2 Methodology

In our performance evaluation of the parallelized Mean Shift algorithm, we utilize various performance metrics to assess the efficiency of our implementations. These metrics include runtime, speedup, achieved occupancy, and percentage of peak sustained memory bandwidth.

To measure the runtime of our implementations, we employ the chrono_timer() code for CPU executions. We utilized the ncu command for GPU executions and LIKWID marker to measure the runtime of the OpenMP threads. This metric provides insights into the processing time required for different concurrency levels.

We also computed the speedup of our implementations to assess the performance improvement achieved by parallelization. We calculated the speedup by dividing the execution time of the serial implementation by the execution time of the parallel implementation.

Achieved occupancy [2] is a GPU-specific metric that quantifies the extent to which the GPU's hardware resources are effectively utilized during computation. This GPU metric is vital in assessing the degree of parallelism and efficiency achieved during GPU computation, and we vary the number of blocks and threads per block to assess its impact.

Percentage of Peak Sustained Memory Bandwidth is a crucial factor in GPU performance, particularly for memory-bound applications like image processing. We calculate the percentage of peak sustained memory bandwidth by analyzing GPU metrics reported by the ncu command. This metric provides insights into the efficiency of memory access patterns in GPU executions and is represented in the form of the percentage of peak sustained number of sectors. Higher values suggest a higher utilization of the unit and may reveal potential bottlenecks, although they do not necessarily indicate efficient usage.

Our experiments are carried out on the specified computing system, which includes both CPU and GPU nodes. Achieved occupancy and memory bandwidth metrics are computed specifically for GPU test cases.

For CUDA implementation, we vary the number threads per block to assess their impact on performance, while for OpenMP implementation, we vary the number of threads.

Additionally we measured the L2 and L3 cache accessess using Likwid to assess the efficiency of data access patterns of our implementations.

Lastly we varied the size of the input image to assess how performance is impacted by the size of the input image. Since we wanted to mantain a proportion between the number of pixels and the number of segmeneted regions, we just incremented the size by tiling the original image in multiple copies of itself. The prblem sizes are 512, 1024, 2048 and 4096.

[2]Documentation available at URL: `https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html`

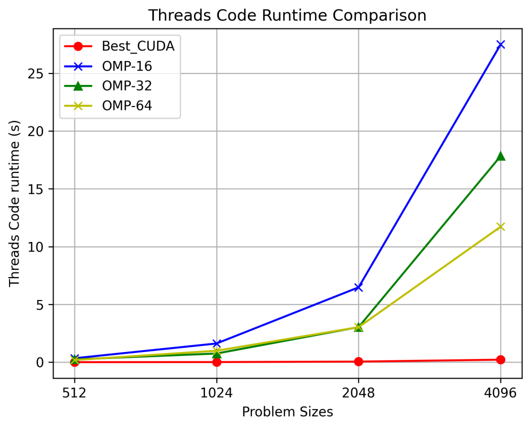

Figure 2: The runtime of the code executed by threads for different implementations.

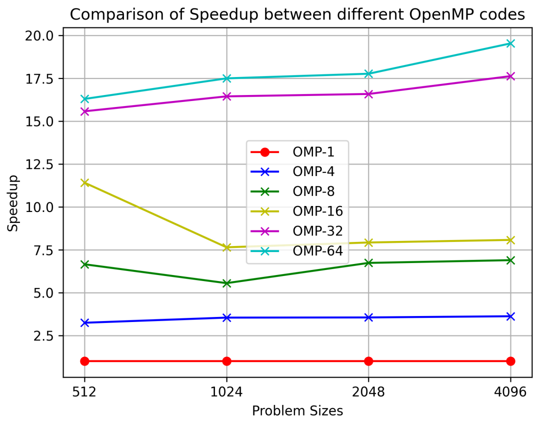

Figure 3: A comparison of the speedup achieved by different concurrency levels.

4.3 Findings

4.3.1 Runtime

The performance evaluation of the parallelized Mean Shift algorithm reveals insightful findings across different parallelization strategies. The runtime results showcase nuances in the efficiency of each approach under varying problem sizes.

When considering the overall runtime, it becomes evident that the serial code exhibits the slowest performance, while the OpenMP implementation with 64 threads achieves the best results, as shown in Fig 1. Surprisingly, the CUDA implementation, even in its optimal configuration, demonstrates only a marginal improvement over the OpenMP version with four threads.

The relatively modest improvement in CUDA performance can be attributed to the inherent challenges associated with data movement. The process of copying the entire image to the GPU incurs a notable time overhead. As a result, the benefits gained from parallelizing the computational workload are partially offset by the time spent on data transfer.

To delve deeper into the performance dynamics, a specific focus on the thread execution portion of the code unveils a different scenario. When exclusively considering the part of the code executed by threads, CUDA emerges as the top performer across all problem sizes, as presented in Fig 2. This signifies that the computational capabilities of CUDA-enabled GPUs, particularly when dealing with concurrent threads, outshine other implementations.

The findings emphasize the importance of considering the balance between computation and data movement when evaluating the overall performance of parallel algorithms. While CUDA threads exhibit superior computational efficiency, the overhead of transferring data to and from the GPU influences the comprehensive runtime results.

Varying the size of the input image provides valuable insights into the scalability of each parallelization strategy. The experiments conducted with problem sizes ranging from 512 to 4096 pixels demonstrate how performance is impacted by the scale of the input data. As the problem size increases, the influence of data movement becomes more pronounced, and strategies that effectively manage this overhead are crucial for optimal performance.

4.3.2 Speedup OpenMP

The speedup of the mean shift algorithm, particularly during the filtering phase, is influenced by the degree of parallelization achieved and the problem size. We analyzed the speedup results for different OpenMP configurations (OMP-1, OMP-4, OMP-8, OMP-16, OMP-32, OMP-64) across varying problem sizes.

Amdahl's Law describes the theoretical limit on the speedup of a parallelized algorithm based on the fraction of the code that cannot be parallelized. In the case of the mean shift algorithm, there may be serial portions that limit the overall speedup.

The chart in 3 table presents the speedup values for different OpenMP configurations across various problem sizes. OpenMP with four threads exhibits relatively consistent speedup across different problem sizes. This suggests that the parallelization is effective and maintains a reasonable level of efficiency even as the problem size increases.

The highest speedup is achieved when utilizing 64 threads (OMP-64) for the largest problem sizes. This indicates that for sufficiently large problems, increasing the number of threads can lead to improved parallel processing efficiency.

4.3.3 Data Access Patterns

The evaluation of L2 and L3 cache access patterns provides valuable insights into the efficiency of data access, revealing interesting trends that vary with concurrency levels and problem sizes.

The L2 cache requests exhibit an expected trend of increasing with the problem sizes (512, 1024, 2048, and 4096 pixels) and aligns with the typical expectation that higher problem sizes require more data access.

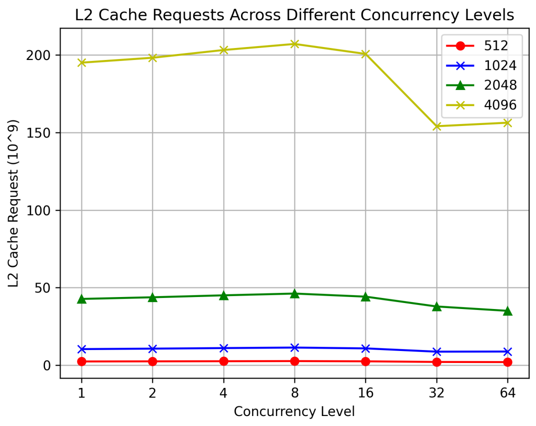

Figure 4: Number of requests to the L2 cache for for different problem sizes across different concurrency levels.

However, an intriguing observation emerges when reaching a certain level of concurrency. Beyond this point, the trend inverts, and L2 cache requests either plateau or experience a slight decrease, as presented in Fig 4. This phenomenon could be attributed to an optimal concurrency level where the L1 cache efficiently absorbs requests and utilizes loaded data. It suggests that, in the context of the Zen3 core, an appropriate number of threads may allow L1 to better take of spatial locality and reduce the need for extensive L2 cache access.

Similar to L2 cache requests, L3 cache requests initially increase with concurrency, as shown in Fig 5. However, the trend inverts at higher concurrency levels, and the number of requests drop significantly. Particularly with the problem size of 4096 pixels. Here, a drastic drop in L3 cache requests occurs after a concurrency level of eight. This phenomenon may be attributed to the size of the chunks processed, which might fit more efficiently within the L2 cache. As a result, at lower concurrency levels, the L2 cache satisfactorily handles requests without the need for extensive access to the larger L3 cache. This finding underscores the importance of considering cache hierarchies and data chunk sizes when optimizing for specific architectures.

The observed trends in L2 and L3 cache access patterns provide valuable guidance for optimizing the Mean Shift algorithm. Optimizing chunk sizes based on cache considerations could further enhance performance.

4.3.4 Achieved Occupancy and Memory Bandwidth

Achieved Occupancy is a crucial metric in evaluating the effectiveness of GPU utilization. In our performance evaluation across different problem sizes (512, 1024, 2048, and 4096 pixels), we observed that the optimal level of Achieved Occupancy is consistently achieved with a configuration utilizing 64 threads per block. This specific thread count consistently demonstrates the highest Achieved Occupancy across varying problem sizes.

The relationship between thread count and block count is crucial in achieving the best level of occupancy. Each thread in the configuration is responsible for processing a pixel. An excessively high number of threads per block may result in some threads remaining idle, even though each thread is assigned the task of handling a pixel. This observation underscores the importance of balancing thread and block counts to maximize GPU occupancy effectively.

Memory Bandwidth is a critical factor in GPU performance, par-

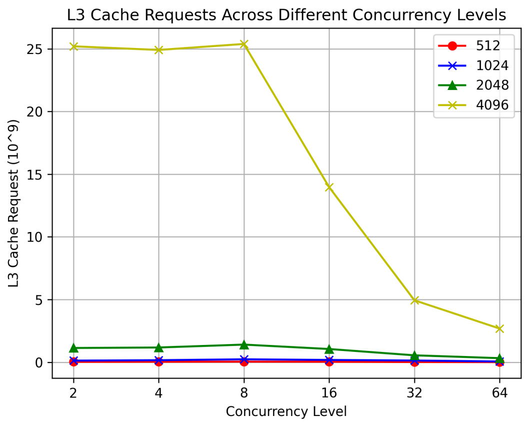

Figure 5: Number of requests to the L3 cache for different problem sizes across different concurrency levels.

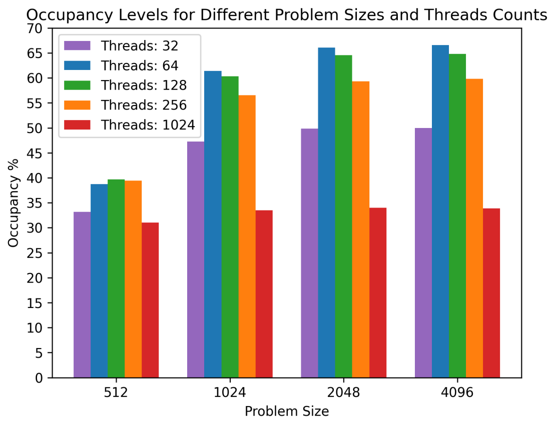

Figure 6: Achieved occupancy for different problem sizes across different threads per block.

ticularly for memory-bound applications like image processing. Our evaluation of Memory Bandwidth across different problem sizes indicates that the results are not optimal, ranging from a minimum of 7% with problem size 512 and 64 thereads configuration, to a maximum of 15% at problem size 4096 with 64 threads per block. This range suggests that the GPU's parallel power is not fully utilized for memory access, and there is room for improvement.

As expected, the Memory Bandwidth does show an increasing trend as the problem size grows. This behavior aligns with the anticipated outcome, where larger problem sizes benefit from increased parallelism and better leverage the GPU's processing capabilities. Despite the observed increase, the achieved Memory Bandwidth values indicate that there is potential for optimization to more efficiently utilize the GPU's memory subsystem.

4.3.5 Threads configurations with CUDA

In our exploration of the impact of different threads per block configurations on the runtime performance of the Mean Shift algorithm, we focused on configurations employing 32, 64, 128, 256, and 1024 threads per block. Our analysis, using the NVIDIA Command Line Profiler (ncu), revealed distinct patterns in the runtime performance across these configurations.

The 64, 128, and 256 configurations consistently stood out as the top-performing setups across all problem sizes (512, 1024, 2048, and 4096 pixels), as demonstrated by Fig 7. The performance improvement achieved showcases their efficiency in processing pixel data, leading to shorter execution times.

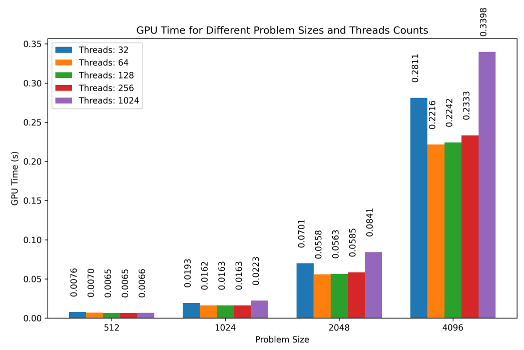

Figure 7: The runtime of the GPU executions for different problem sizes across different threads per block.

The preference for 64, 128, and 256 threads per block configurations in terms of runtime performance suggests that these settings strike a balance between parallelism and resource utilization. The optimal choice of threads per block depends on factors such as the nature of the algorithm, GPU architecture, and problem size.

Developers should consider experimenting with different thread per block configurations to find the most suitable balance for their specific implementation. Fine-tuning this parameter can significantly impact the overall performance of the parallelized Mean Shift algorithm.

5 Conclusions and Future Work

Our exploration of parallelized implementations for the Mean Shift algorithm has provided valuable insights into the strengths and challenges of GPU (CUDA) and CPU (OpenMP) parallelization. The following key conclusions and avenues for future work emerge from our study.

While CUDA demonstrated promising parallel performance, there is room for further improvement in data transfer efficiency. The need to minimize data copying between the host and device is crucial for enhancing overall GPU performance. Investigating advanced memory management techniques and data transfer optimization strategies could be a focus area for future CUDA code refinements.

Introducing early stopping and pruning techniques could offer significant gains in computational efficiency. By avoiding redundant work for similar pixels, the algorithm can converge more quickly, leading to improved runtime performance. Exploring algorithm-level optimizations, such as early termination conditions, could be a fruitful direction for enhancing both GPU and CPU implementations.

Our experiments revealed that OpenMP achieved commendable performance even without extensive optimization efforts. Further exploration into fine-tuning OpenMP configurations and leveraging advanced compiler directives may unlock additional performance gains. Additionally, considering hybrid parallelization strategies that combine GPU and CPU resources could be explored to maximize overall system efficiency.

While parallelization using CUDA and OpenMP demonstrated promising performance, a future exploration of distributed computing using MPI (Message Passing Interface) holds promise for scaling Mean Shift algorithm implementations across multiple nodes. Investigating how MPI can be integrated into the parallelization strategy could lead to significant performance improvements for large-scale image processing tasks. This avenue becomes particularly relevant as the dataset size and computational demands grow.

References

[1] D. Chen, L. Wang, G. Ouyang, and X. Li. Massively parallel neural signal processing on a many-core platform. *Computing in Science & Engineering*, 13(6):42–51, 2011.

[2] C. M. Christoudias, B. Georgescu, and P. Meer. Synergism in low level vision. In *2002 International Conference on Pattern Recognition*, vol. 4, pp. 150–155. IEEE, 2002.

[3] D. Comaniciu and P. Meer. Mean shift analysis and applications. In *Proceedings of the seventh IEEE international conference on computer vision*, vol. 2, pp. 1197–1203. IEEE, 1999.

[4] K. Fukunaga and L. Hostetler. The estimation of the gradient of a density function, with applications in pattern recognition. *IEEE Transactions on information theory*, 21(1):32–40, 1975.

[5] X. HU, J. LUO, Z. SHEN, W. WU, and Q. CHEN. Data sewing algorithm for parallel segmentation of high-resolution remotely sensed image. *Journal of Remote Sensing*, 14(5):917–927, 2010.

[6] F. Huang, Y. Chen, L. Li, J. Zhou, J. Tao, X. Tan, and G. Fan. Implementation of the parallel mean shift-based image segmentation algorithm on a gpu cluster. *International Journal of Digital Earth*, 12(3):328–353, 2019.

[7] M. Huang, L. Men, and C. Lai. Accelerating mean shift segmentation algorithm on hybrid cpu/gpu platforms. *Modern Accelerator Technologies for Geographic Information Science*, pp. 157–166, 2013.

[8] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee. Snucl: an opencl framework for heterogeneous cpu/gpu clusters. In *Proceedings of the 26th ACM international conference on Supercomputing*, pp. 341–352, 2012.

[9] X. Qi, F. Xing, D. J. Foran, and L. Yang. Robust segmentation of overlapping cells in histopathology specimens using parallel seed detection and repulsive level set. *IEEE Transactions on Biomedical Engineering*, 59(3):754–765, 2011.

[10] J. Sirotković, H. Dujmić, and V. Papić. Accelerating mean shift image segmentation with ifgt on massively parallel gpu. In *2013 36th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pp. 279–285. IEEE, 2013.

[11] X.-g. Sun, M.-c. Li, Y.-x. Liu, L. Tan, and W. Liu. Accelerated segmentation approach with cuda for high spatial resolution remotely sensed imagery based on improved mean shift. In *2009 Joint Urban Remote Sensing Event*, pp. 1–6. IEEE, 2009.

[12] R. Szeliski. *Computer vision: algorithms and applications*. Springer Nature, 2022.

[13] F. Zhou, Y. Zhao, and K.-L. Ma. Parallel mean shift for interactive volume segmentation. In *Machine Learning in Medical Imaging: First International Workshop, MLMI 2010, Held in Conjunction with MICCAI 2010, Beijing, China, September 20, 2010. Proceedings 1*, pp. 67–75. Springer, 2010.