PaSSPI: Parallel Semantic Segmentation with Pipeline and Image Slicing

Mao-Siang Chen[*]
siang.cs09@nycu.edu.tw
National Yang-Ming Chiao-Tung University
Hsinchu, Taiwan

Min-Jung Li[†]
a0988282303@gmail.com
National Yang-Ming Chiao-Tung University
Hsinchu, Taiwan

Chu-Chun Chi[‡]
chuchun1231@gmail.com
National Yang-Ming Chiao-Tung University
Hsinchu, Taiwan

ABSTRACT

This study delves into the efficacy of various parallelization techniques in model inference, specifically examining their application to the SegFormer model. Focusing on four key methods—Data Parallelism, Pipeline Parallelism, Multiple GPU Parallelism, and Image Slicing. Our findings indicate that Data and Pipeline Parallelism significantly enhance computational efficiency on CPU platforms without compromising the model's mean Intersection over Union (mIoU) value. However, these techniques showed limited effectiveness when implemented on single GPU setups. To address this, we extended the experiment to multiple GPU configurations, where, following strategic adjustments, both Data and Pipeline Parallelism demonstrated improved performance. In contrast, Image Slicing proved less effective due to high overhead costs associated with thread creation and multi-process parallelism. This study highlights the importance of tailoring parallelization strategies, offering valuable insights for optimizing model inference in the field of artificial intelligence and parallel programming.

1 INTRODUCTION

In recent years, self-driving cars have surged in popularity. However, the ongoing discussions surrounding the security of these systems have raised crucial concerns. In the quest to bolster the safety of self-driving technology, two key elements stand out: ensuring high accuracy and minimizing response time when encountering diverse road scenarios. To address these challenges, one approach we are considering is to focus on a critical component of the self-driving system — semantic segmentation.

2 RELATED WORK

2.1 Semantic segmentation

Semantic segmentation is a computer vision task that assigns a class label to every pixel of an image, thus creating detailed maps of different objects and their boundaries within the input image. There have been a lot of projects conducted regarding semantic segmentation tasks. In their research [1], a Flow Alignment Module (FAM) is introduced to effectively and efficiently learn the Semantic Flow between feature maps from adjacent levels. This module facilitates the broadcasting of high-level features to high-resolution features. Finally, achieve 80.4% mIoU on Cityscapes with a frame rate of 26 FPS.

[*]Both authors contributed equally to this research.
[†]Both authors contributed equally to this research.
[‡]Both authors contributed equally to this research.

2.2 SegFormer

In the realm of semantic segmentation, SegFormer [2] stands out as an innovative approach that overcomes the limitations of handcrafted and computationally intensive components. Its hierarchical Transformer encoder, featuring Mix Transformer encoders and an efficient self-attention mechanism, captures both local and non-local attentions while producing multi-level features. SegFormer's architecture represents a significant advancement in semantic segmentation by demonstrating the effectiveness of a streamlined approach, offering a promising direction for the field. In our project, we leverage the efficiency and robustness of SegFormer and complement its power with parallelization techniques to expedite the process, making it an ideal solution for accelerating segmentation tasks.

2.3 PiPPy

PiPPy[3], short for Pipeline Parallelism in PyTorch, is an open-source project offering a comprehensive toolkit designed to automate parallelism and scaling for PyTorch models. In our project, we are committed to leveraging PiPPy to implement pipeline parallelism due to the automated model parallelism feature it provides. It empowers us to streamline the implementation of parallelism within our models without requiring extensive modifications.

3 PROPOSED SOLUTION

In the realm of model inference optimization, numerous methodologies have emerged, significantly enhancing computational efficiency. Among these, TensorRT and various ONNX optimization formats are prominent. However, for our experiment, we are focused on investigating three intuitive parallelism approaches, namely Data Parallelism, Pipeline Parallelism, and Multiple GPU Parallelism. The target of our application is the SegFormer model, a modern architecture known for its robust performance in segmentation tasks.

3.1 Data Parallelism

This technique involves distributing data across multiple processing units. In our context, it means partitioning the input data and feeding these subsets to different CPUs/GPUs. Each CPU/GPU computes its segment of the data independently, and the results are then aggregated. This approach is highly effective in handling large datasets, as it significantly reduces the time required for model inference by parallel processing. For implementing Data Parallelism, we have chosen to utilize the built-in torchrun function in PyTorch.

This choice is primarily due to its capability to effectively manage inter-process communication (IPC).

3.2 Pipeline Parallelism

Our approach to Pipeline Parallelism involves leveraging PyTorch's open-source tool, PiPPy. PiPPy utilizes its special compiler, which prepares the model for the next phase of pipeline parallelism. Once compiled, the model undergoes automatic segmentation into different pipelines through its runtime-stack and other mechanisms. This automated segmentation is pivotal in dividing the model into distinct segments, each of which can be processed on different CPUs/GPUs. By doing so, the model can be processed in stages, with the output of one stage seamlessly feeding into the next.

3.3 Multiple GPU Parallel Inference

Another way to improve the inference latency is to utilize powerful computing resources on cloud. If network latency is shorter, inference on cloud machines can be a possible solution to reduce latency when performing a huge workload of semantic segmentation tasks on self-driving vehicles. We will build a demo system using Huggingface Accelerate API to communicate with multiple GPUs, aiming to parallelize the inference process and simulate the situation. After waiting for all GPUs to finish their work, we will gather the results and compute the mIoU metrics. Finally, we will compare the elapse time between different datasets and numbers of GPUs for further discussions and improvement.

3.4 Multi-Threaded Image Slicing

The proposed method involves partitioning a single image into four smaller sub-images, which are then individually processed by the model. This approach aims to achieve faster processing speeds by utilizing smaller inputs within the model. The concept behind this technique is to enhance computational efficiency and reduce memory requirements, a crucial consideration in image processing tasks.

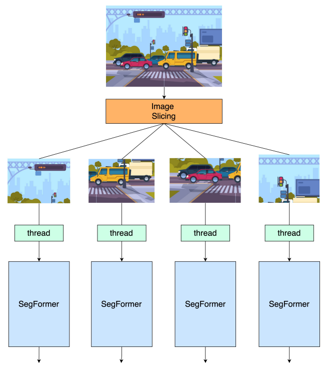

Figure 1: Parallel Image Slicing with Multi-Threading

4 EXPERIMENTAL METHODOLOGY

4.1 Datasets

In our research, we employed two datasets: the Cityscapes[4] and ADE20K[5].

The Cityscapes Dataset predominantly consists of urban street scenes captured from various countries, offering a wide array of weather conditions and times of day, thus providing a multifaceted view of urban landscapes.

ADE20K encompasses an extensive collection of scene-centric images that range from intricate indoor settings to expansive outdoor environments. Together, these datasets contribute a rich tapestry of visual data that challenge and validate our parallel processing methodologies across a spectrum of real-world scenarios, thereby ensuring that our findings are robust and applicable to a variety of image recognition and processing applications.

4.2 Correction

To ensure the accuracy and reliability of our experimental outcomes, we will be implementing a rigorous evaluation of the inference results using the mean Intersection over Union (mIoU) metric, which is a standard measure of accuracy in semantic segmentation tasks. This metric will allow us to quantitatively assess the precision of our parallel processing methods by calculating the average overlap between the predicted segmentation and the ground truth across all classes.

4.3 Environment

In our experiments, we utilized two environments for testing: the PP Server T.A. provided and a Cloud Computing Resource. Each of these environments offers unique attributes and capacities, allowing us to assess the effectiveness of parallelism under different conditions.

The PP Server was equipped with an Intel® Core™ i5-7500 CPU clocking in at 3.40GHz and an NVIDIA GeForce GTX 1060 6GB GPU. In contrast, the Cloud Computing Resource was powered by an Intel® Xeon® Gold 6154 CPU at 3.00GHz, paired with an NVIDIA Tesla V100 GPU, and ran on the Ubuntu 22.04 LTS operating system.

5 EXPERIMENTAL RESULTS

5.1 Data and Pipeline parallelism with CPU

Table 1: Performance on CPU Using Torch Random Input

Version	Time	Speedup
serial	3.6424	1X
dp	0.9560	3.788X
pp	0.9901	3.66X

The first table presents the test results conducted on a CPU, where we utilized torch random as the input with a size of (1, 3, 224, 224). In this setup, our process count was set to 4. It is evident from the results that both pipeline parallelism (pp) and data parallelism (dp) achieved significant optimization effects.

Table 2: Performance on CPU Using Real Dataset Input

Version	Time	Speedup	mIoU
serial	0.75135	1X	0.2494
dp	0.449	1.67X	0.2494
pp	0.6903	1.08X	0.2494

The previous table displayed results obtained using Torch random as the input. In contrast, this table showcases the outcomes measured using real dataset. It's noteworthy that both pipeline parallelism (pp) and data parallelism (dp) continue to achieve commendable optimization results even when applied to real images.

5.2 Data and Pipeline parallelism with GPU

Table 3: Performance on cuda Using Torch Random Input

Version	Time	Speedup	Memory (MB)
serial	0.0392	1X	316.4
dp	0.0387	1.012X	218.4
pp	0.0590	0.655X	139.2

Table 4: Performance on mps Using Torch Random Input

Version	Time	Speedup	Memory (MB)
serial	0.14198	1X	695.25
dp	0.144	0.985X	656.88

Table 5: Performance on GPU Using Real Dataset Input

Version	Time	Speedup	mIoU
serial	0.1325	1X	0.24942
dp	0.1685	0.786X	0.24941
pp	0.149	0.88X	0.17

The first two tables above represent results obtained using Torch random as the input, run on cuda device and mps device, respectively, while table 5 represent result obtained using real dataset as input on cuda device. It was observed that the results showed a significant acceleration compared to CPU processing – approximately 90 times faster! However, the speedup achieved by pipeline parallelism (pp) and data parallelism (dp) relative to serial processing was not as substantial as seen with the CPU. In some cases, these methods were even slower.

We hypothesize that this might be due to a couple of factors. Firstly, the computational workload might not be substantial enough, while the cost of communication is too high, leading to communication time creating an overhead. This is a common issue in parallel computing, where the overhead of managing parallel tasks can sometimes outweigh the benefits, especially for smaller workloads. Secondly, as our workstation is equipped with only one GPU, this hardware limitation might not showcase the expected performance enhancements.

Based on these observations, we have designed subsequent experiments to further investigate and address these issues. These experiments aim to delve deeper into the nuances of GPU-based parallel processing, exploring ways to optimize the balance between computation and communication, and considering the impact of hardware configurations on performance.

5.3 Multiple GPU parallelism

Table 6: Parallel inference and evaluation (Cityscapes)

# of GPU	Time	Speedup	mIoU
1	74.0	1X	0.24942
2	338.24	0.218X	0.24942
4	281.63	0.263X	0.24942

In this section, we initially parallelize both the inference and evaluation processes using varying numbers of GPUs on the Cityscapes dataset. Upon observation, it becomes evident that parallelizing inference and evaluation with 2 and 4 GPUs results in significantly longer elapsed times compared to the serial version. Consequently, we conduct a profiling analysis for configurations employing 2 and 4 GPUs.

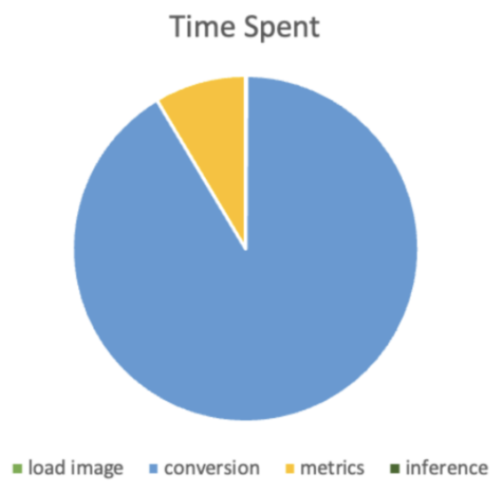

Figure 2: Profiling analysis of parallel inference and evaluation on cityscapes dataset

The results of the profiling analysis reveal a noteworthy finding: a substantial portion of the processing time is dedicated to the label conversion process, specifically from trainId to label. This conversion is indispensable for obtaining accurate evaluation results on the Cityscapes dataset. However, the conversion operation involves extensive searches within the trainId2Label dictionary and requires frequent data transfers between the CPU and GPU, contributing significantly to the overall time consumption.

Since the dictionary can not moved to GPU, we found out other two possible solutions: First, do parallel inference on GPUs and evaluate the results on CPU serially. Second, select a different dataset that doesn't require trainId conversion after inference. Applying either one of them can better find out the speedup ratio and make sure the correction at the same time. In the next section, we will show the results of both two methods.

Table 7: Parallel inference on GPU (Evaluate on CPU)

# of GPUs	inference Time	Speedup	mIoU
1	27.26	1X	0.24942
2	13.01	2.09X	0.24942
4	8.19	3.328X	0.24942

While the evaluation is done on CPU serially instead of on each GPU worker, we focus on the speed up while inferencing on different numbers of GPUs. In the graph above, we can observe that there is a speedup of 2.09 with 2 GPUs and 3.328 with 4 GPUs while inferencing 100 images. Although the speedup of 2 GPUs is larger than 2, the error is still in an acceptable range.

Table 8: Parallel inference and evaluation (ADE20K)

# of GPUs	Time	Speedup
1	64.21	1X
2	37.69	1.703X
4	22.95	2.798X

Furthermore, when conducting tests on ADE20K datasets, both the inference and evaluation processes exhibit the potential for parallelization without substantial communication overhead between the CPU and GPU on each worker. While label conversion remains a necessary step, it can be executed in parallel as it involves simple addition operations. In comparison to the serial version, the results demonstrate a commendable speedup—1.703 times faster with 2 GPUs and 2.798 times faster with 4 GPUs, which is deemed reasonable.

The result of multiple GPU parallelism suggests that once the model grows bigger and the network latency drops, inference images on the cloud will be a possible solution for a heavy workload on self-driving vehicles. As the previous section shows, without additional label conversion, parallel GPU inference has a great speedup with multiple GPUs. For further experiments, if we test on a heavier workload, the speedup ratio might be more significant.

5.4 Multi-Threaded Image Slicing

Table 9: Processing Image with Different Segment Sizes

Image Size	Time
full-sized	5.75
quarter-sized	5.69
sixteenth-sized	5.66

Table 9 presents the results of running different sizes of images on the model: a full-sized image, a quarter-sized image (1/4), and a sixteenth-sized image (1/16). Surprisingly, the time taken for processing these varied sizes is almost similar. This outcome suggests that the overhead of creating threads is larger than the overhead of processing larger images. Consequently, even though multi-threading is employed for parallelization, it does not lead to faster processing times. Instead, more time is spent on creating threads.

6 CONCLUSIONS

In this comprehensive experiment, we explored four parallelization methods: Data Parallelism, Pipeline Parallelism, Multiple GPU Parallelism, and Image Slicing, applied to the SegFormer model. We found that Data and Pipeline Parallelism were highly effective on CPUs, maintaining accuracy while improving parallel processing. However, their performance on single GPU setups was less impressive, leading us to extend the experimentation to multiple GPUs. After profiling techniques and adjustments, these methods showed improved results on two and four GPU setups.

In contrast, Image Slicing did not yield favorable outcomes due to the high overhead associated with thread creation and multi-process parallelism.

In conclusion, while Data and Pipeline Parallelism demonstrated success in CPU and multiple GPU environments, the limitations on single GPU setups and the ineffectiveness of Image Slicing underline the importance of adapting strategies to specific hardware configurations. These insights are valuable for future research in computational efficiency and model optimization.

REFERENCES

1. Xiangtai Li, Ansheng You, Zhen Zhu, Houlong Zhao, Maoke Yang, Kuiyuan Yang, Yunhai Tong: Semantic Flow for Fast and Accurate Scene Parsing. ECCV (2020)
2. Enze Xie, Wenhai Wang, Zhiding Yu, Anima Anandkumar, Jose M. Alvarez, Ping Luo1SegFormer: Simple and Efficient Design for Semantic Segmentation with Transformers
3. James Reed, Pavel Belevich, Ke Wen: PiPPy: Pipeline Parallelism for PyTorch. https://github.com/pytorch/PiPPy
4. Cityscape Dataset: https://www.cityscapes-dataset.com/dataset-overview/
5. ADE20K Dataset: https://groups.csail.mit.edu/vision/datasets/ADE20K/