Optimization on Text-Image Retrieval with CLIP Vision Transformers

110062211
謝東豫

110062212
洪梓翔

June 18, 2024

Contents

1 Abstract

In this project, we show the possibility of optimizing the inference of Vision Transformers (ViT). We use the CLIP model proposed by OpenAI [1] with text-image retrieval as our application scenario, and use several attempts to parallelize the inference process. **In our work, we gain about 26.57x speed-up for CPU only inference and 23.68x speed-up for GPU inference comparing to the baseline.**

2 Introduction

In this project, we aim to explore parallel acceleration for text-image retrieval which is a system that searches for images based on textual input. By utilizing the input text from the user, this system aims to find the most relevant images from a database.

Text-image Retrieval can be divided into two parts: the retrieval system and the image database, with the former being implemented using Vision Transformer. The Vision Transformer (ViT) [2] applies the Encoder from the transformer architecture to image classification tasks. In practice, it replaces the CNN layers with self-attention computation and has achieved impressive results in classification problems. The Fig 1 shows how Vision Transformer works.

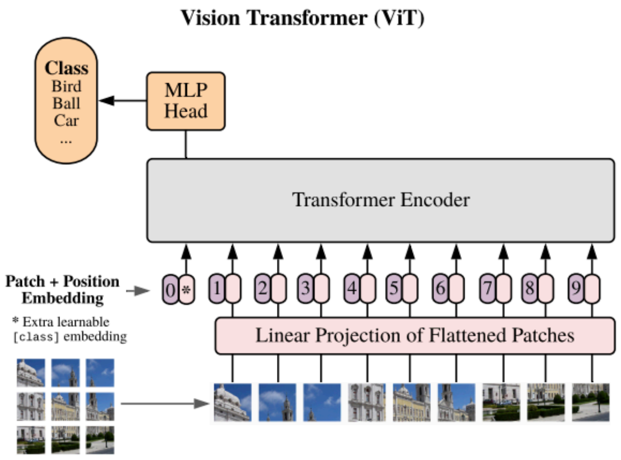
Vision Transformer (ViT)
Class
Bird
Ball
Car
...
MLP Head
Transformer Encoder
Patch + Position Embedding
* Extra learnable [class] embedding
0 * 1 2 3 4 5 6 7 8 9
Linear Projection of Flattened Patches

Figure 1: Vision Transformer[1]

We first introduce a Vision Transformer proposed by OpenAI—CLIP. Its core concept is to ”map text and images into the same space.” It uses two encoders to map text and images into the same vector space, determining the relevance between them by calculating the similarity between the text and images. Through the transformer architecture and extensive dataset pretraining, CLIP’s versatility far exceeds that of models trained on ImageNet alone, achieving excellent performance in zero-shot classification across different domain datasets. The Fig 2 shows how CLIP maps text and images into the same vector space.

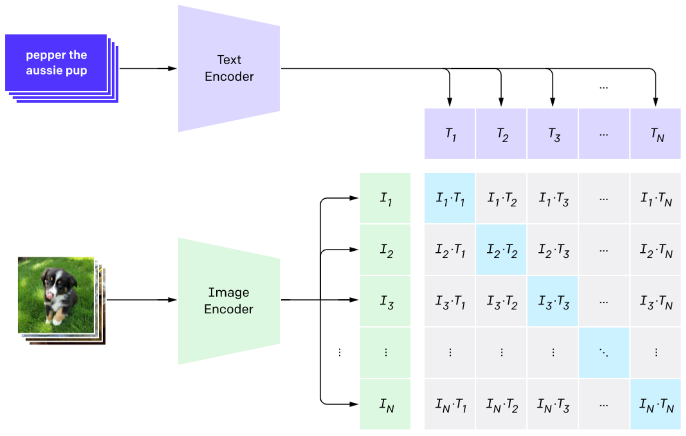
pepper the aussie pup
Text Encoder
Image Encoder
T_1 T_2 T_3 ... T_N
I_1 I_2 I_3 ⋮ I_N
$I_1 \cdot T_1$ $I_1 \cdot T_2$ $I_1 \cdot T_3$ ... $I_1 \cdot T_N$
$I_2 \cdot T_1$ $I_2 \cdot T_2$ $I_2 \cdot T_3$ ... $I_2 \cdot T_N$
$I_3 \cdot T_1$ $I_3 \cdot T_2$ $I_3 \cdot T_3$ ... $I_3 \cdot T_N$
$I_N \cdot T_1$ $I_N \cdot T_2$ $I_N \cdot T_3$ ... $I_N \cdot T_N$

Figure 2: CLIP Model[2]

Since CLIP can determine the degree of relevance between text and images, it is well-suited for application in text-image retrieval. After receiving the user's input text, CLIP compares it to all the images in image database. This involves mapping the text and images into the same vector space and calculating their similarity as mentioned earlier. Finally, after comparing the input text with all the images, we return the images with their similarity higher than a defined threshold as the result of the text-image retrieval. The Fig 3 shows the structure of text-image retrieval.

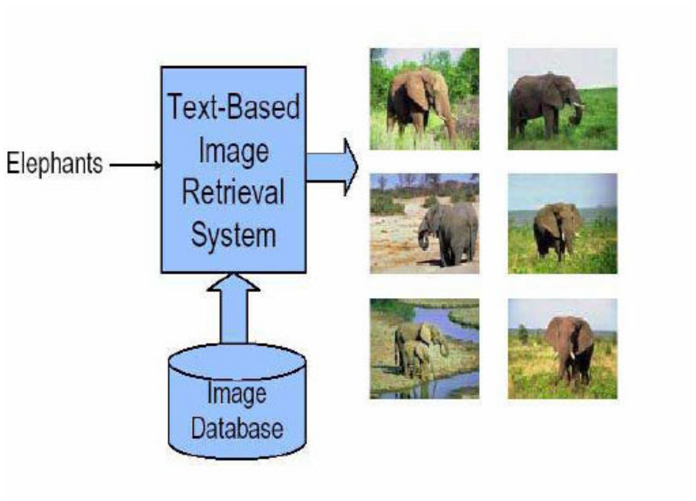

Figure 3: text-image retrieval[3]

3 Task Definition

In our work, we chose a fine-tuned OpenAI clip-vit-base-patch32 model as the retrieval system, and the Train-mini set of the iNaturalist Dataset 2021 (a total of 500,000 images) as the image database. Then, we define our user scenario as follows: The user inputs a name of a species included in our database, the retrieval system will use the input prompt to compare with all images to return similarity scores. To focus on acceleration of inference, we only choose top nine highest similarity score pictures as our final output.

Based on the previously introduced text-image retrieval, the main task of this project is to attempt to accelerate the entire search system using different parallelization methods. To achieve this, we explore and implement various

[1]From https://research.google/blog/transformers-for-image-recognition-at-scale/

[2]From https://openai.com/index/clip/

[3]From https://www.researchgate.net/figure/Text-Based-Image-Retrieval-these-search-engines-are-fast-and-robust-but-sometimes-they_fig1_264276256

parallelization techniques, each aimed at improving the efficiency and speed of the retrieval process. By leveraging these methods, we aim to handle the large-scale computations involved in text-image retrieval more effectively.

In the following sections, we will describe each parallelization method in detail, explaining how they contribute to speeding up the system and the specific implementation strategies used in this project.

4 Methodology

There are four different optimization attempts we've tried. We will explain all of them one by one in the following sections.

4.1 Evaluation Platform

We use two different platform listed in Table 1 to evaluate our experiment, which aims for CPU inference and GPU inference respectively. While our personal cluster is mainly for observing GPU inference optimization, we will still leverage it's CPU to get better observation.

	CPU	GPU
Taiwania 3	2 x Intel(R) Xeon(R) Platinum 8280	N/A
Personal Cluster	2 x Intel(R) Xeon(R) Gold 6230	1 x NVIDIA Tesla V100

Table 1: Hardware environment of evaluation platform

4.2 Optimization Attempts

4.2.1 Baseline

The baseline in our work uses the example code on the huggingface hub model card of `openai/clip-vit-base-patch32` [3] as template. We simply iterate through every image in our database and take them as input to the model one by one. All of the pictures will be processed with the `CLIPProcessor` first then passed into the model for evaluation. We store all of the similarity score in a single array and sort them by the `sorted()` function provided by python, then retrieve images we want by filtering the score.

4.2.2 Batched Inference

The first method we've tried is to simply use batched inference. Just like most of the other Transformers, with batched inference can have performance improvement due to more inputs can be processed concurrently. While this could consume lots of memory to store those inputs, CLIP itself is a relatively small model which only occupied for about 1.7 GB of memory so we are able to have for flexibility to raise batch size and increase memory utilization.

We batched our images in our database by iterating through all images in our database, and put those images to a list with it's size equals to assigned batch size. These lists will be forwarded to the model to achieve a simple batched inference.

4.2.3 Parallel `CLIPProcessor`

The entire inference stage can be separate into three major stages. The first stage is data pre-processing, which is tokenization and image processing inside `CLIPProcessor`. The next stage will be data transfer, processed tensors have to be moved to corresponding device for the next phase. The final stage is model forwarding. Data will go through the entire CLIP model and calculate the final outputs such as `logits_per_images`. While using CPU to inference, the torch backend will leverage OpenMP to accelerate the final phase. Based on this fact, we observe the entire inference process with different number of OpenMP threads. The result is shown in Fig 4.

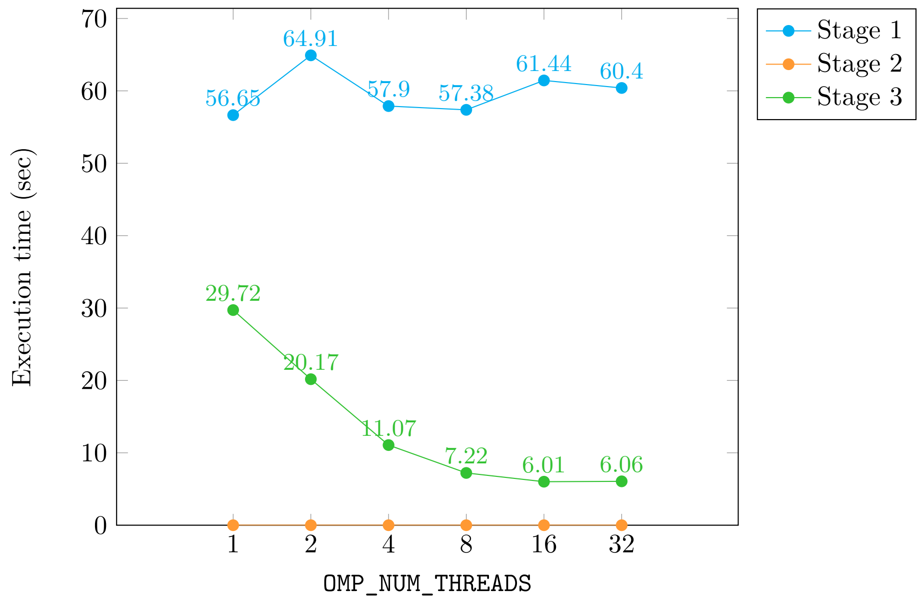

Figure 4: Execution time of each stage in CLIP inference

Stage 2 only occupies minimal amount of time which can be neglected. However, the trend of Stage 1 and Stage 3 shows that the `CLIPProcessor` will not be parallelized by torch with OpenMP in it's backend, which points out a possibility of further optimizations.

Hence, we decide to parallel this process with data parallelism. We create a producer-consumer framework which distributes some cores to perform stage one processing (`CLIPProcessor` preprocessing) to become a producer and push pre-processed batches into a queue, and other cores will be used by CLIP model to parallel execute model evaluation as a consumer. All input data is distributed to all producer evenly to balance the workload of each producer as much as possible.

To prevent intense contention, we have to control the number of producers and number of threads used by the consumer. The ideal situation is that the sum of number of producers and number of threads equals to the number of physical cores on the machine while the total producer output speed is identical to the consumer processing speed. In real cases, it's very hard to match identical processing speed under limited number of physical cores. We use manual tuning to find the best configuration under our working environment here.

4.2.4 Load Balancing and Data Parallelism

Furthermore, based on the producer-consumer framework mentioned in the last section, we utilize both CPU and GPU resources to get maximum performance in our platform. We design a simple static scheduling and dynamic scheduling rule to distribute the entire workload on both CPU and GPU and inference simultaneously.

In static scheduling, we use the following method to distribute the workload:

1. Use a few images to benchmark the inference time ratio on CPU and GPU.
2. Use this ratio to distribute the amount of data for CPU and GPU, and the number of producers to pre-process images for different devices.

Under this method, CPU consumer and GPU consumer will have independent task queue. As for dynamic scheduling, the distribution is done as follows:

1. All producers put processed batches into same task queue
2. All consumers process the task in first-come-first-serve order

With this kind of framework, we can also deploy more than one model on both CPU and GPU, thus achieving data parallelism in model level. Since the model forwarding process is very stable, the computation is highly related to the number of batches per process. In this way, we can have better utilization and average load balancing.

5 Implementation

In our implementation, we use Python interface to access and use CLIP model with `transformers` module with `torch`. We implement parallelization with `multiprocessing` module in Python, and use Huggingface `dataset` module to load our selected image database. Detail of our implementation can be accessed at `https://github.com/Catking14/IPC24-image-retrieval`.

6 Results

6.1 Baseline

First, we test out baseline performance in our working environment. The result is shown in Fig. 5. We are able to get about 1.72x speed-up by just porting the model and input tensors to GPU to leverage GPU inference by integrated `to("cuda")` method.

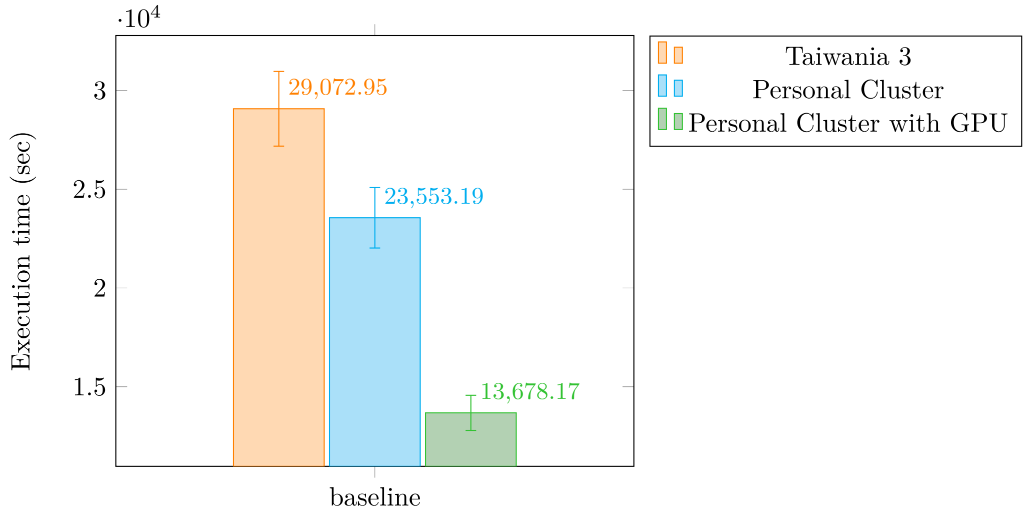

Figure 5: Execution time of baseline

6.2 Batched Inference

With batched inference, we test out the performance with different batch size to find the sweet spot of batch size on our working environment. The outcome is shown in Fig. 9. The overall trend is quite different on different machine. On Taiwania 3, we are able to have best performance with batch size 2048, which is the largest batch size we've tested. However, the best performance on our personal cluster appears at batch size 32 on both CPU and GPU side. This could be pure hardware problem due to memory bandwidth and computational differences. After batched inference is applied, we can get about 3.07x faster on our personal cluster and 6.77x faster on Taiwania 3 with CPU inference, and 4.39x faster with GPU inference on our cluster.

6.3 Parallel `CLIPProcessor`

After the parallelization of `CLIPProcessor`, we are able to further improve the CPU inference performance. We manually tune the number of producers and `OMP_NUM_THREADS` to get the best configuration. The result is shown in Fig.

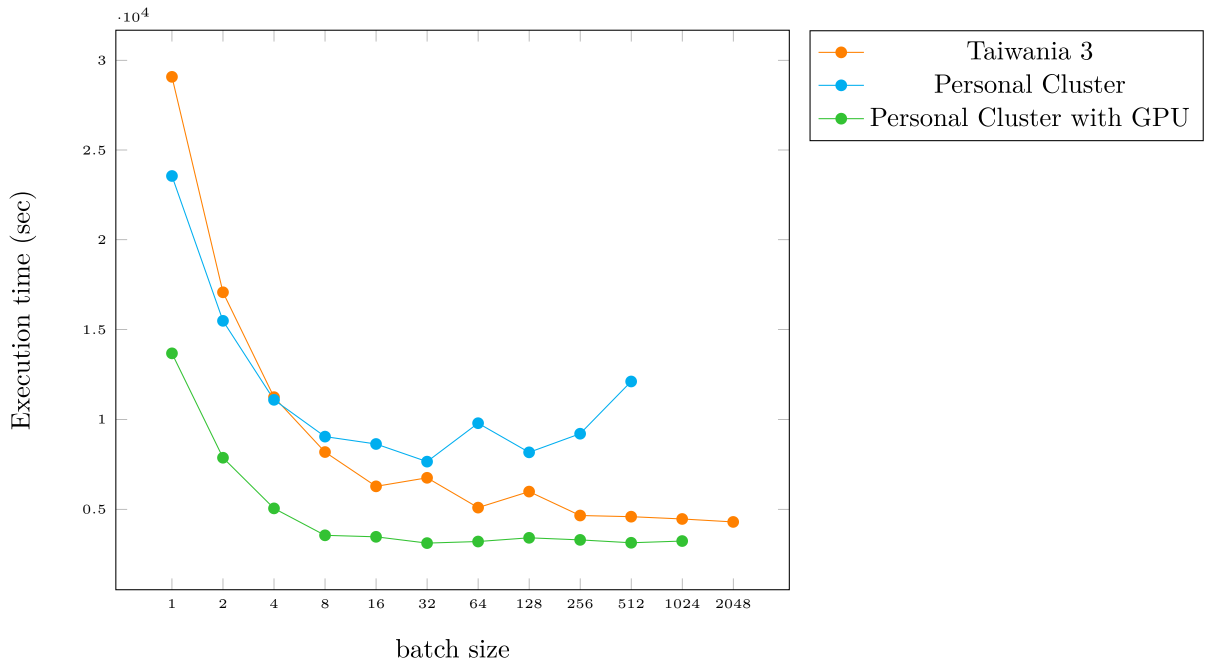

Figure 6: Execution time of different batch size

7. On our personal cluster, we can get maximum 1.66x faster with the configuration of 5 producer and `OMP_NUM_THREADS` equals to 30. As for Taiwania 3, there are maximum 2.3x faster with the configuration of 3 producers and `OMP_NUM_THREADS` equals to 27.

6.4 Load Balancing and Data Parallelism

Finally, with two different scheduling method and model level of data parallelism, we can further improve the inference speed. First, we compare static scheduling and dynamic scheduling with the same configuration. We set 10 producers and `OMP_NUM_THREADS` equals to 20 with a model on CPU and GPU on our own cluster to leverage heterogeneous computing. It turns out that dynamic scheduling outperforms static scheduling due to more balanced workload under all circumstances (Fig. 8). Although the characteristic of model forwarding make the estimation of workload and execution time easier, static scheduling still faces inaccurate distribution sometimes. While dynamic scheduling can effectively distribute workload evenly under both of our working environment.

To get the best performance, we further deploy different number of models on our devices to make use of all computing resources as much as possible. On Taiwania 3, since it only have CPU processors, we can only apply model level data parallelism based on parallelized `CLIPProcessor`. Under this circumstance, the best performance we can get is achieved by setting 3 producers,

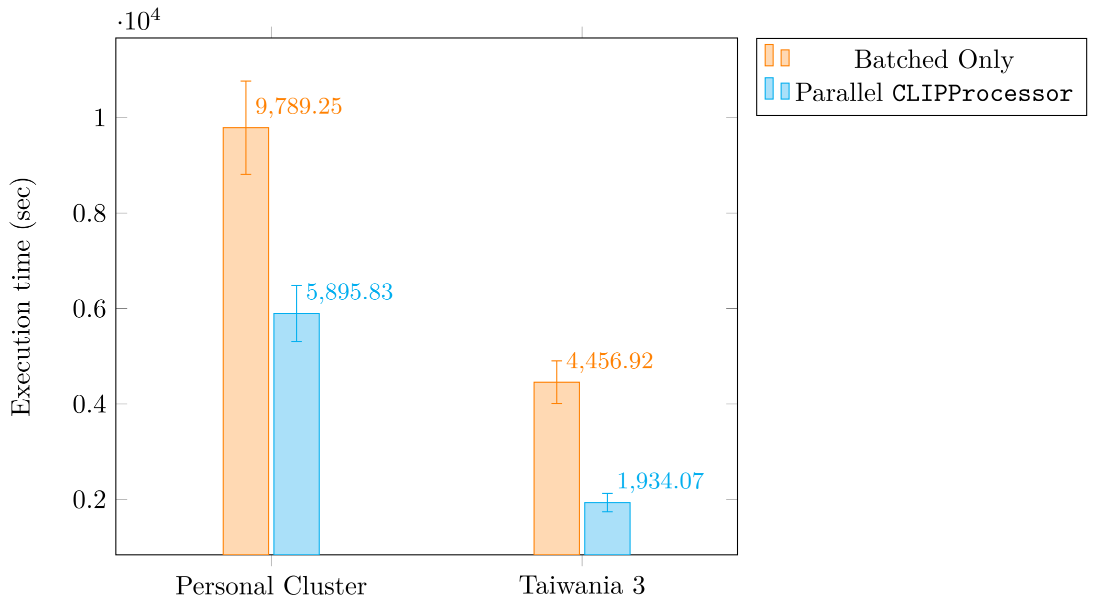

Figure 7: Execution time of Parallelized `CLIPProcessor`

`OMP_NUM_THREADS` equals to 7, and 4 models in total. For our personal cluster, with the aid of GPU, we can get significant improvement with 10 producer, `OMP_NUM_THREADS` equals to 20, 1 model on CPU, and 4 model on GPU. If we limited to use CPU only, we can still get huge performance improvement with 5 models in parallel.

After all optimizations, we are able to get about 8.43x faster on our cluster with CPU inference, 26.57x faster on Taiwania 3, and 23.68x faster our our cluster with GPU inference as shown in Fig. 10.

References

[1] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, G. Krueger, and I. Sutskever, "Learning transferable visual models from natural language supervision," 2021.

[2] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," 2021.

[3] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, G. Krueger, and I. Sutskever, "Huggingface hub of openai/clip-vit-base-patch32." `https://huggingface.co/openai/clip-vit-base-patch32`, 2021.

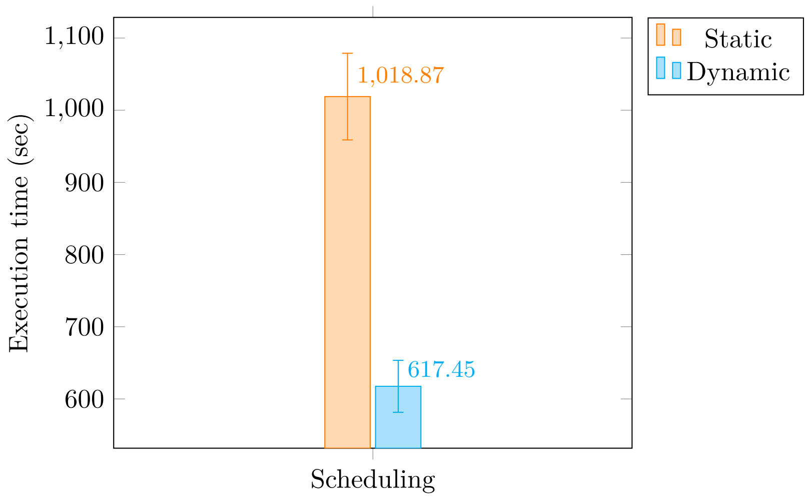

Figure 8: Execution time of different scheduling strategy

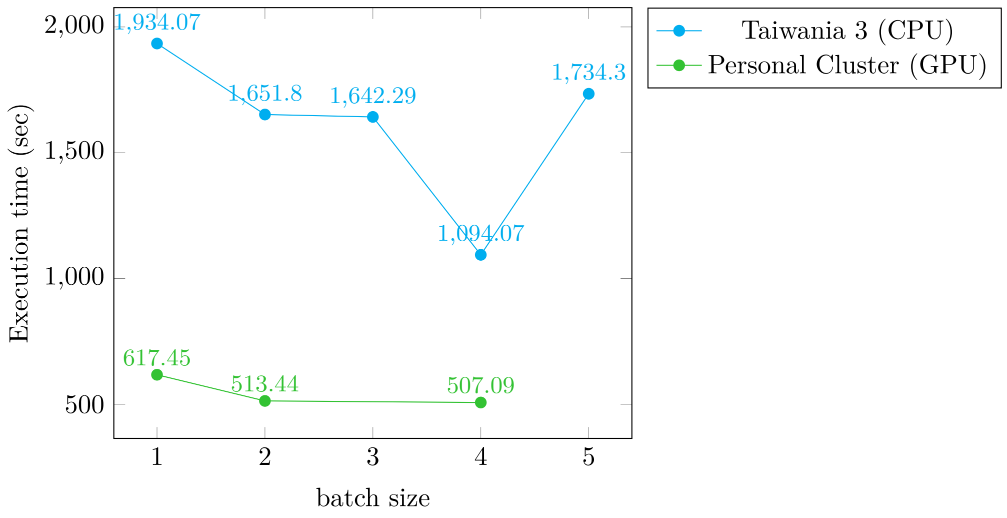

Figure 9: Best execution time of different number of models deployed simultaneously. The number is the obtained by tuning the configurations of producers and threads per model manually.

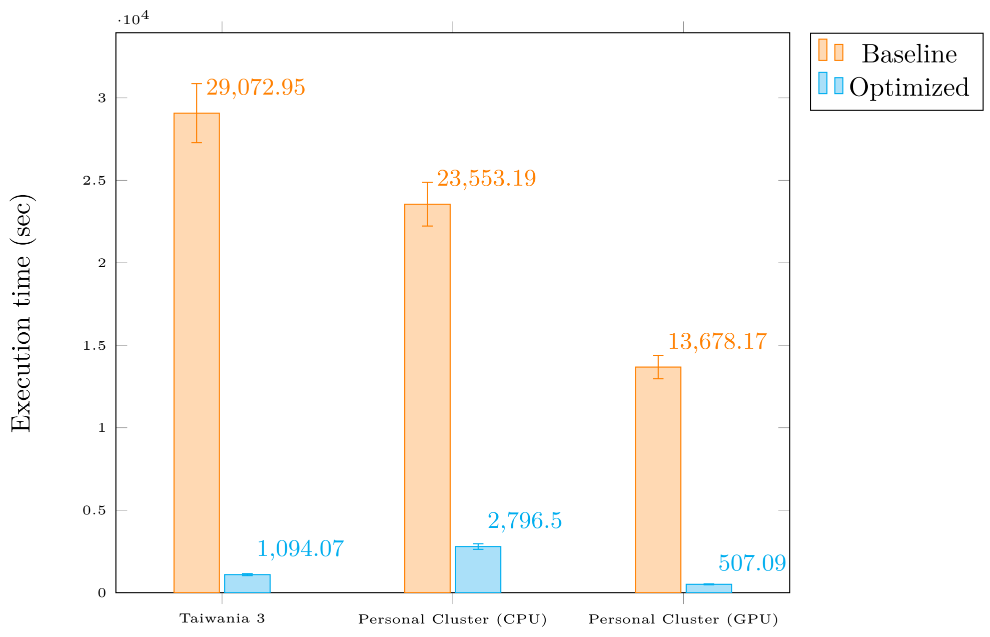

Figure 10: Execution time of baseline and optimized