Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

Minimizing transformer inference overhead using controlling element on Shenwei AI accelerator

Yulong ZHAO[†1], Chunzhi WU[1,2], Yizhuo WANG[3], Lufei Zhang[1], Yaguang ZHANG[3], Wenyuan SHEN[3], Hao FAN[1], Hankang FANG [4], Yi QIN[4], Xin LIU[†‡5]

[1]*State Key Laboratory of Mathematical Engineering and Advanced Computing, Wuxi 214000, China*

[2]*School of Non-Commissioned Officer, Space Engineering University, Beijing 100004, China*

[3]*ChinaNational Supercomputing Center in Wuxi, Wuxi 214000, China*

[4]*Zhejiang Lab, Hangzhou 310000, China*

[5]*China National Research Centre of Parallel Computer Engineering and Technology, Beijing 100081, China*

[†]E-mail: zhaoyl04@163.com; yyylx@263.net

Received May 28, 2024; Revision accepted Aug. 25, 2024; Crosschecked

Abstract: Transformer models have become a cornerstone of various Natural Language Processing (NLP) tasks. However, the substantial computational overhead during the inference remains a significant challenge, limiting their deployment in practical applications. In this study, we address this challenge by minimizing the inference overhead in Transformer models using the controlling element on AI accelerators. Our work is anchored by four key contributions. First, we conducted a comprehensive analysis of the overhead composition within the Transformer inference process, identifying the primary bottlenecks. Second, we leveraged the Management Processing Element (MPE) of the Shenwei Aritificial Intelligence (SWAI) accelerator, implementing a three-tier scheduling framework that significantly reduced the number of host-device launches, achieving a reduction approximately 10,000 times lower than that achieved by the original PyTorch-GPU setup. Third, we introduced a zero-copy memory management technique using segment-page fusion, which significantly reduced memory access latency and improved overall inference efficiency. Finally, we developed a fast model loading method that eliminates redundant computations during model verification and initialization, reducing the total loading time for large models from 22,128.31 milliseconds to 1041.72 milliseconds. Our contributions significantly enhance the optimization of Transformer models, enabling more efficient and expedited inference processes on AI accelerators.

Key words: Transformer inference optimization; Three-tier scheduling; Zero-copy memory management; Fast model loading
https://doi.org/10.1631/FITEE.2400453 **CLC number:**

1 Introduction

Over the past decade, pre-trained language models based on the Transformer architecture (Vaswani *et al.*, 2017) have become a leading paradigm in the domain of Natural Language Processing (NLP). Notable examples of such models include BERT (Devlin *et al.*, 2018), GPT2(Radford et al., 2019), Llama (Touvron et al., 2023), wav2vec2.0 (Baevski *et al.*, 2020), and Whisper (Radford A *et al.*, 2023). These Transformer-based models have significantly advanced the state-of-the-art in terms of accuracy, surpassing traditional models. Nonetheless, the computational intensity during the inference phase poses a significant barrier to their integration into real-world applications, which require strict criteria such as low latency, rapid inference capabilities, and cost-effective operational overhead.

There are currently two strategies for enhancing the inference efficiency of Transformer models. The first pertains to the optimization of specific operators,

‡ Corresponding author

such as Softermax (Stevens *et al.*, 2021) and FlashAttention (Dao T *et al.*, 2022), and involves reducing accuracy and minimizing I/O costs to enhance the operators' computational speed. The second focuses on the optimization of the model's runtime during inference, using various techniques such as structured pruning (Kim Y J *et al.*, 2020), knowledge distillation (Fang J *et al.*, 2021), operator fusion, memory management (Chen S *et al.*, 2021), and parallel scheduling strategies(Du J *et al.*, 2022). These approaches collectively aim to improve inference efficacy by reducing model size and enhancing overall system performance. However, the intrinsic computational overhead (Ma X *et al.*, 2021) associated with Transformer inference has received relatively little scholarly attention, despite its critical role and the need for comprehensive research in this area.

To minimize the overhead during the inference process, we propose an innovative approach that utilizes the hardware control mechanisms of AI accelerators. We began by conducting a detailed analysis of the Transformer model's inference control process, exploring the components of the runtime overhead through carefully designed experiments. Based on this analysis, we introduced a fast loading method that significantly reduces the overhead caused by loading models with PyTorch-GPU. We also developed a three-tier scheduling framework for interacting with the host and the controlling element on the accelerator, focusing on leveraging the accelerator's scheduling capabilities. To minimize device launch expenditures, we embraced a holistic full-graph optimization strategy. Additionally, we implemented a zero-copy memory management protocol based on segment-page fusion, which eliminates data transmission-related costs. These optimizations target the reduction of overhead throughout the inference lifecycle of Transformer models. Our paper's contributions are as follows:

1. We undertook a detailed analysis of the overhead constituents within the inference process, categorizing them into three main segments: model loading, API invocations, and ancillary factors. We found that model loading and API invocations account for 96% of the total overhead, providing insights crucial for developing strategies to reduce these overheads.

2. We leveraged the scheduling capabilities of the controller element (MPE) on the SWAI accelerator, implementing a three-tier scheduling framework that reduced the number of launches by 10,000 times compared to the baseline.

3. We designed a zero-copy memory management technique for the SWAI accelerator cards, leveraging a segment-page fusion memory management mechanism at the software level. This allows the MPE and Computing Processing Element (CPE) to access the same physical address via their distinct virtual addresses, significantly reducing data replication overhead during computation.

4. We propose a fast model loading method, following an in-depth examination of the Whisper model loading procedure. We find that the model verification and model initialization account for 81.26% of the total model loading duration. Our method eliminates redundant computations, such as parameter resetting in the linear layers during model verification and model initialization, integrating the initialization with the loading of the model parameters for efficient inference. This reduced the total loading duration for the large model from 22,128.31 milliseconds to a mere 1,041.72 milliseconds.

This paper is organized as follows: Section 2 introduces the fundamental concepts and provides an overview of related work. Section 3 offers a detailed analysis of torch-GPU. In Section 4, we present the three-tier scheduling framework, the segment-page fusion memory management mechanism, and fast model loading procedure. Section 5 presents results from various tasks and Section 6 summarizes the findings and discusses future works.

2 Background and Related Work

2.1 Basic concepts

The Transformer architecture, central to contemporary NLP, consists of multiple encoder-decoder stacks that encode input sequences and decode output sequences. A key feature of this architecture is the "self-attention" mechanism, which enables the model to capture long-range dependencies in input sequences in a non-sequential manner. This capability makes the Transformer architecture particularly well-suited for NLP tasks such as machine translation, text generation, and question answering, consistently

achieving a state-of-the-art performance.

The most recent addition to the Transformer family,Whisper, is based on pre-training models and weakly supervised learning.Unlike its predecessors, Whisper is distinguished by its comprehensive architecture, which includes components for converting audio to Mel-spectrograms and a language recognition module. It demonstrates remarkable versatility, handling multiple tasks such as transcription, translation, and speech activity detection from a single audio input. Whisper adheres closely to the standard Transformer model, making it an ideal candidate for our analysis and the optimization of inference overhead.

The combination of the Transformer model's characteristics with the demand of NLP tasks significantly increases operational overhead. This increase is primarily attributable to two factors. First, the Whisper model's design inherently increases overhead, as it is trained on 30-second audio segments. Transcribing longer sounds requires the prediction of timestamps and the segmentation of the Mel-spectrogram into multiple 30-second intervals, which must be processed sequentially. This procedure must be iterated multiple times for lengthy audio inputs. Second, the Transformer model itself introduces overhead within each time segment. The decoder component generates tokens sequentially, with each token's computation relying not only on the encoder's output but also on previously generated tokens. For each time segment, the encoder operates once, whereas the decoder undergoes multiple iterations to achieve completion. The encoder processes each time segment once, while the decoder requires multiple iterations to complete. This leads to a substantial number of launches and memory copies for device interaction, resulting in significant communication overhead, particularly with smaller models. This overhead can constitute the majority of the total computational cost.

2.2 Transformer Inference Overhead

The significance of data transmission and kernel launch overhead in the context of lightweight neural networks for GPU-based inference has been highlighted in recent scholarly work. These studies emphasize the need to address these overheads as they are critical to the performance of such networks. For instance, Kim S *et al*(2021) reveal that the kernel launch overhead of lightweight neural networks on mobile GPUs is notably high. Additionally, Fujii Y *et al*(2013) discuss the challenges posed by the heterogeneity of GPU computing in data transmission, while Arafa Y *et al*(2019) present improvements in the performance of CUDA applications by reducing CPU-GPU data transmission overhead.

When NLP tasks are executed using the Transformer model, they follow a linear operational sequence characterized by continuous information exchange between the CPU and GPU. The cost associated with these cross-device interactions has become a major concern, particularly for models of a smaller scale. According to Zhang L *et al.* (2019), the average GPU launch time for a single device is about 6 μs without any parameter transfer, and the average computation time of the kernel function under the basic model is less than 10 μs. Here, the launch overhead is a considerable part of the total time. Moreover, during the launch operation, data transfer and other interactions between the CPU and GPU also occur, such as querying the GPU device. The significant one-time overhead, combined with the numerous calls, makes the interaction costs between the host and device a significant bottleneck that prevents effective model inference.

2.3 GPU optimization method

The challenge of interaction overhead in the segregated architecture of control and computation has been widely recognized by researchers. To address this, numerous strategies have been proposed to mitigate this overhead. Ma X *et al.*(2021) provided an in-depth analysis of the runtime overheads associated with deep learning inference in neural network models, examining factors such as end-to-end performance, hardware platforms, memory bandwidth, and model structures. Kim S *et al.* (2021) developed a performance model predicting the optimal timing for kernel flushes to minimal overhead. Chen G *et al.* (2015) introduced a "free launch" technique, facilitating the expression of dynamic parallelism through sub-kernel launches, and presented a "launch removal" code transformation that replaces sub-kernel launches with parent thread reuse. Chu CH *et al.* (2013) proposed a novel approach to achieving low latency and high bandwidth by dynamically fusing

packing/unpacking GPU kernels to reduce expensive kernel launch overhead. Fujii Y *et al.* (2013) analyzed data transfer concerns for GPUs and characterized currently achievable data transfer methods in cutting-edge GPU technology. Sunitha NV *et al.* (2017) explored the effects of overlapping data transfer and kernel execution on the overall execution time of CUDA applications. Lee K *et al.* (2021) verified various memory access technologies using different memory bindings on the AMD Fusion system (Llano A8-3850). Boudier P *et al.* (2011) explored the impact of memory fusion on CPU–GPU heterogeneous computing.

Fang J *et al.* (2021) and Wang X *et al.* (2023) recognized the overhead caused by frequent kernel launches during runtime and effectively addressed it through operator fusion. LightSeq, as described in Wang X *et al.* (2023), implemented a specialized kernel function for layer normalization using the CUDA toolkit, ensuring a single kernel launch without intermediate results and integrating operations across all GEMM operations. This approach reduced the number of launches per encoder layer to just six. Similarly, TurboTransformers, detailed in Fang J *et al.* (2021), adopted operator fusion to improve the parallelization of operations like SoftMax.

Despite the efforts of LightSeq and TurboTransformers to reduce the launch cost, it is crucial to acknowledge that they have not entirely eliminated this overhead. Although they have effectively reduced the frequency of kernel launches and optimized specific operations, there still exists some residual launch cost within their runtime systems.

2.4 Controlling element on accelerators

Neural network inference, evaluating a network for a given input, provides many knobs for tuning and optimization. Substantial research has been performed in this direction, and many good hardware accelerators have been proposed to improve inference speed and energy efficiency (Sze V *et al.*, 2023). The concept of accelerators is not a recent innovation within the computing field; numerous accelerators have been conceptualized and realized over the years. An accelerator is essentially defined as "a separate architectural substructure [. . .] that is architected using a different set of objectives than the base processor, where these objectives are derived from the needs of a special class of applications" (Patel S *et al.*, 2008).

A common feature among accelerators is the integration of one or more controllers, which are pivotal in managing the accelerator's resources, including memory units and systolic arrays. Some of these accelerators leverage general-purpose cores as their primary controlling elements (Peccerillo B *et al.*, 2022). For instance, Huawei's Ascend series integrates 16 ARM cores based on the DaVinci architecture (Huawei, 2020), while Intel's Xeon Phi (Mittal S *et al.*, 2020; Sodani A S *et al.*, 2016) and Intel Nervana NNP-I (Wechsler O *et al.*, 2019) both include x86 controller cores within their design. GraphH (Dai G *et al*, 2018) opts for an ARM Cortex-A5 with a floating-point unit (FPU), and Baidu's Kunlun K200, a manycore accelerator, is equipped with an arithmetic logic unit (ALU) for basic instructions and a special function unit (SFU) for more complex operations like logarithms, exponentiation, and square roots, as part of their XPU-clusters (Ouyang J *et al.*, 2020).

The SWAI Accelerator exemplifies the manycore approach, which has proven to be an effective evolution from the multi-core paradigm, emphasizing an even greater degree of parallelism (Peccerillo B *et al.*, 2022). As depicted in Fig. 1, the SWAI accelerator features a ring network bus design that interconnects the MPE (also known as the master core), four CGs (core group, also known as the slave core group), and dual-mode PCIe components. Each CG contains an HBM (High Bandwidth Memory) storage controller and an array of 32 CPEs (also known as slave cores), arranged in a 4*8 matrix. Each CPE consists of a super-scalar processing core, an intra-core local storage and communication engine, and an intelligent acceleration core. The MPE is designated as the central coordinator responsible for the management of on-card resources, while the CPEs are specialized in executing computational tasks and are particularly well-suited for handling complex mathematical operations and data processing at high velocities. Additionally, a DMA (Direct Memory Access) engine on the PCIe module facilitates efficient data transfer between the host and accelerator card.

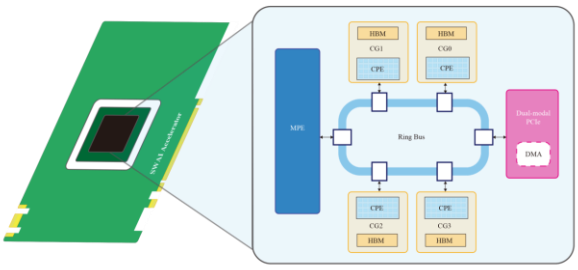


Fig. 1 SWAI Accelerator Architecture

3 Performance Analysis

In models that leverage GPUs for inference computation, the execution timeline of a program can be divided into the following distinct phases: model loading, API invocations, GPU computations, and other overheads. We refer to the time spent on GPU computations as "effective computation time" and to the cumulative duration of all other phases as overhead. Our goal is to significantly reduce the overhead associated with Transformer model inference. To achieve this, we conducted a thorough analysis of the control flow intrinsic to the inference task, using the Whisper model as a representative case study. This model was chosen due to its comprehensive architecture, which includes both encoder and decoder components, and its ability to handle multiple tasks, such as transcription and translation from audio inputs. Our analysis was designed to identify and quantify the various sources of overhead during the inference process. To validate our analysis, experimental validation was conducted on the inference workflow, executed on a server equipped with an Nvidia Tesla V100S GPU.

3.1 Overhead analysis

Fig. 2 presents a schematic diagram of the Whisper model's operational workflow, which is based on the standard Transformer architecture for executing tasks of translation and transcription on input speech data. The process begins with converting raw speech into Mel spectrograms, followed by segmenting the speech data into approximately 30-second intervals. Each segment undergoes an encoding phase, integrating positional encodings and passing through multiple encoder layers. The encoded features, combined with the preceding token, are then fed into a series of decoder layers to generate subsequent tokens, starting with a start symbol. This cycle continues until an end token is identified, marking the end of the token generation process. The commencement of the next temporal segment is determined to repeat the process until the entire speech sequence is fully translated or transcribed. A post-processing phase refines and selects the most optimal tokens from the encoder's output. In this context, the time allocated to computational operations at the device level is designated as effective computation time, while the remaining time is categorized as device-independent overhead.

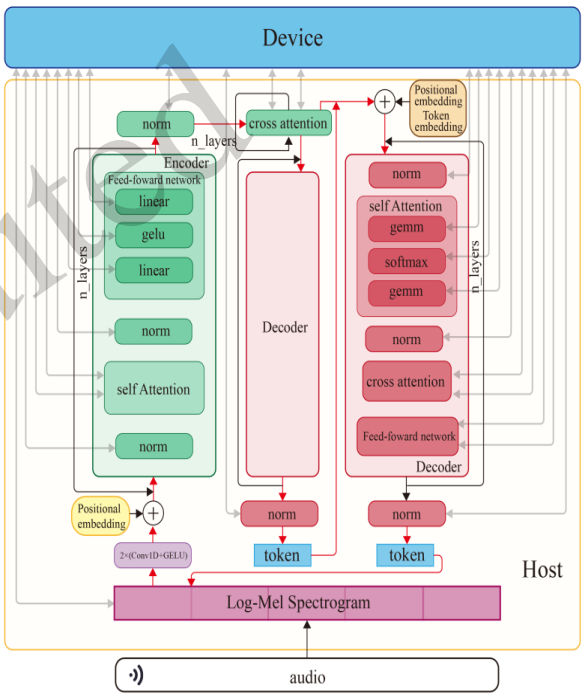


Fig. 2 Interactions Between Host and Device

The heterogeneous separation architecture of control and computation presents challenges, notably the frequent interaction required for each operator's computation, including launching operations and executing memory copies. Fig. 2 illustrates that the entire task involves multiple linear execution cycles, extending from the encoder to n decoders, with the main loop indicated in red. Each operator, whether implemented by cuDNN or cuBLAS, entails numerous launch or memcpy operations. For instance, the conv1d operator involves a series of direct operations,

Table 1 Model loading breakdown table

Model	model validation/ms	model parameters load/ms	Whisper model initialization/ms	Whisper model to device/ms	load model total/ms
base	399.70	**1409.16**	**561.81**	98.77	2469.44
large	**7717.77**	3051.17	**10549.78**	1215.35	22534.07

including Copy_Kernel_Cuda, the nchwToNhwcKernel executed twice, the CUDA-Function_Add perfmormed once, and computeOffsetsKernel launched once, adding up to seven operations in total. Additionally, auxiliary operations such as memcpy, getDevice, and moduleUnload, which handle CUDA resource allocation and deallocation, also contribute to the costs of the interaction between the host and the device.

Following a comprehensive theoretical analysis of the Whisper model's inference process, we proceeded to conduct empirical experiments to evaluate the model's overall inference duration on both the base model (approximately 74 million parameters) and the large model (approximately 1550 million parameters). The experiments utilized the following official audio samples: JFK.wav, with an 11-second speech segment, and HP0.wav, with a 260-second speech segment. The experimental data, combined with the theoretical analysis, segmented the inference process into model loading, API invocations (including encoder-decoder interactions), GPU computation, and other elements. The results are systematically presented in Fig. 3， where (a) illustrates the base model’s duration breakdown with JFK.wav, (b) depicts the large model's duration breakdown with JFK.wav, (c) details the base model's duration breakdown with HP0.wav, and (d) exhibits the large model's duration breakdown with HP0.wav.

A review of the experimental data in Fig. 3 shows that the model loading time increases with the model size, becoming the dominant component of the inference timeline for compact models and brief audio samples. The temporal demand of the API invocations demonstrates a positive correlation with both the model size and processing task length, becoming the main time-consuming element in prolonged audio inference. Together, these two components account for over 96% of the total overhead, highlighting the necessity for detailed analysis to better understand their performance characteristics. Further exploration

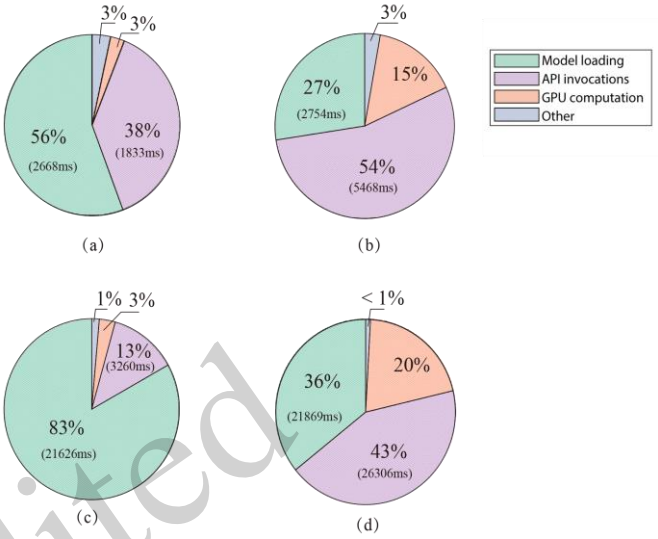


Fig. 3 Breakdown of Whisper inference duration

of these aspects will be conducted in subsequent sections.

3.2 Model loading

Upon a thorough review of the workflow within the Whisper model, it was observed that the loading time for the base and large models accounted for 56% and 83% of the total duration, respectively, as illustrated in Fig. 3. This significant time expenditure on model loading has driven an in-depth analysis of the loading process, which is composed of several key stages, including verifying the integrity of the model, loading the model to retrieve its fundamental parameters, initializing a Whisper instance based on the model's essential parameters, and transferring the Whisper model to the device. Table 1 presents a comprehensive breakdown of the time allocation for these stages in both model sizes, indicating that the base model's loading is predominantly influenced by parameter loading, while the large model is significantly affected by model verification and initialization.

The model validation procedure commences

with retrieving the model file and then employing the SHA-256 cryptographic algorithm to calculate the hash value. This hash value is subsequently verified to affirm the integrity of the model. The computational time for this hash algorithm is directly proportional to the file size, which is a critical consideration for large models.

The model parameters are subsequently imported utilizing the pickle serialization module. These parameters are subsequently employed as parameters to initiate a Whisper instance. The initialization procedure constructs the encoder and decoder layers, which consist of Residual Attention Blocks (RABs). The number of RABs is determined by the model's parameters, with each RAB comprising one or two multi-Head attention (MHA) layers. Each MHA layer consists of four linear layers responsible for initializing the weights of the Query-Key-Value(QKV) matrices using the kaiming_uniform_ method, ensuring a uniform initialization.

For the large model, the encoder and decoder consist of 32 layers each, with the encoder featuring single MHA layers in its RABs and the decoder containing pairs, totaling 96 MHA layers. The initialization of the linear layers is the most time-consuming part of the model parameter loading process. In the large model configuration, the initialization of 512 linear layers takes an average of 11 milliseconds per layer, totaling 5.6 seconds. This initialization step accounts for 53.13% of the overall Whisper model initialization duration of 10.54 seconds.

3.3 API invocations

API invocations constitute a significant proportion of the computational overhead during the inference process of the Whisper model. As the complexity of inference tasks increases, the time spent on API invocations is expected to become the dominant factor in the inference workflow's temporal expenditure. Utilizing the analytical tool **nvprof**, we conducted a comprehensive examination of the inference process to ascertain the GPU kernel execution time and identify the costs associated with the top eight API invocations, which collectively contribute to the majority of the inference duration. The detailed findings of this analysis are systematically presented in Table 2.

Table 2 API invocations details

model (input)		Kernel execution	cudaStream-IsCapturing	Cuda Free	Cuda Launch Kernel	cuMod-uleUnload	cudaStreamCre-ateWith-Flags	cudaMem-cpyAsync	cuDevice GetAttrib-ute	cudaGet Device
base (jfk)	times		4	**4**	10104	443	16	207	3136	102529
	Duration/ms	127.13	63.06	**625.32**	568.64	182.26	127.71	30	16.5	36.34
	Proportion%	6.8	3.41	**33.84**	30.75	9.85	6.90	1.62	0.89	1.96
base (hp0)	times		28	4	**203182**	443	16	4131	3136	1880511
	Duration/ms	1541	83	624.42	**1970.4**	195.2	118.56	107	12.6	534.93
	Proportion%	28.7	1.55	11.63	**36.69**	3.63	2.21	1.99	0.23	9.95
large (jfk)	times		8	4	**54729**	443	16	212	3136	534280
	Duration/ms	765	109	501	**788.6**	252.55	162.56	483	25.55	178.84
	Proportion%	22.9	3.27	15.04	**23.67**	7.58	4.88	14.48	0.77	5.36
large (hp0)	times		61	4	1168077	443	16	4635	3136	10418693
	Duration/ms	**12500**	47	452	8578.6	309	143	749	19	3327
	Proportion%	**46.04**	0.17	1.66	31.60	1.14	0.53	2.76	0.07	12.25

Extensive empirical testing has shown that for small-scale inference tasks, the API invocations overhead significantly exceeds the actual GPU computation time. This is particularly evident when using the base model with the JFK dataset. The most substantial overhead is found to be associated with the CudaFree operation, which is uniquely triggered at the onset of the initial GPU computation cycle. This can be attributed to PyTorch's adoption of a deferred initialization strategy for context initialization. Specifically, PyTorch executes the CudaFree(0) operation to signify the commencement of the initialized context. The overhead introduced by this operation remains constant regardless of the model size or input data, making it a critical component for optimization, especially for smaller models and inputs.

As both the model size and the input task complexity grow, CudaLaunchKernel increasingly becomes the predominant source of overhead within API invocations. The frequency of calls to CudaLaunchKernel rises with the length of the audio task and the size of the model, as longer audio segments necessitate more processing cycles and larger models require more encoder and decoder layers. For instance, when processing the HP0 audio file, its cumulative time attributed to CudaLaunchKernel often exceeds 30% of the total inference time.

It is important to emphasize that the GPU computation (Kernel execution) time also increases with larger model inputs. When using large models and the HP0 audio file for inference tasks, the GPU computation time is no longer significantly less than the API call overhead, accounting for 40.06% of the combined total of API invocations and GPU computation time. Even with the rise in GPU computation time for large models and longer audio input, API overhead, including CudaLaunchKernel, also increases and remains a significant portion.

To further explore the relationship between launch cost and audio length, we conducted a detailed analysis of the frequency and timing of cudaLaunchKernel invocations. Our investigation into the launch duration revealed that the initial kernel launch at the commencement of program execution exhibited a notably longer duration, on the order of hundreds of milliseconds. In contrast, the average duration for each subsequent launch was significantly reduced, averaging around 4.5 microseconds.

To enhance the precision of calculating launch times induced by various inference tasks, we have developed a formulaic approach to determining the total number of CUDALaunchKernel calls and to estimating the associated launch time:

$$launch\_times = n \times layer_{encode} \times \beta_{encode} + m \times layer_{decode} \times \beta_{decode} \quad (1)$$

$$launch\_time = 112 \times t_1 \times layer \quad (2)$$

Equation (1) is broadly applicable to Transformer models, facilitating the calculation of host–device interactions predicated on a specific model and task. In this equation, n signifies the count of encoding layers, while m denotes the number of decoding layers. Here, $layer_{encode}$ corresponds to the encoding layer construct, and $layer_{decode}$ represents the decoding layer structure. The quantity of these layers is typically prescribed by the model architecture. Additionally, β_{encode} indicates the interaction time per encoding layer, while β_{decode} refers to the interaction time per decoding layer.

In Equation (2), t_1 symbolizes the input task duration in seconds, and *layer* refers to the number of layers. Based on empirical data, a 30-second speech recording typically necessitates roughly one encoding operation and 60 decoding operations, with each encoding layer instigating about 56 launch operations. Employing these empirical data points, we have ascertained the average number of launch operations per second for audio processing, which is 112 in Equation (2). This formula allows us to swiftly approximate the number of launch operations expected in the transcription process for a given model and task, with projected outcomes exhibiting a discrepancy of less than 20% from actual measurements.

3.4 Others

Beyond the primary cost factors of model loading and API invocations, the Whisper inference process incurs several ancillary costs. Before the encoding phase, the input audio requires preprocessing to derive the Mel spectrogram. Post-decoding, the decoder applies greedy algorithms or beam_search algorithms to refine the optimal outcomes, calculates probabilities from the resulting matrix, and selects corresponding tokens from the vocabulary to generate the final output. Given that these elements constitute a minor fraction of the total cost and are of secondary importance, this study deliberately does not allocate undue focus on these components.

4 Proposed Method

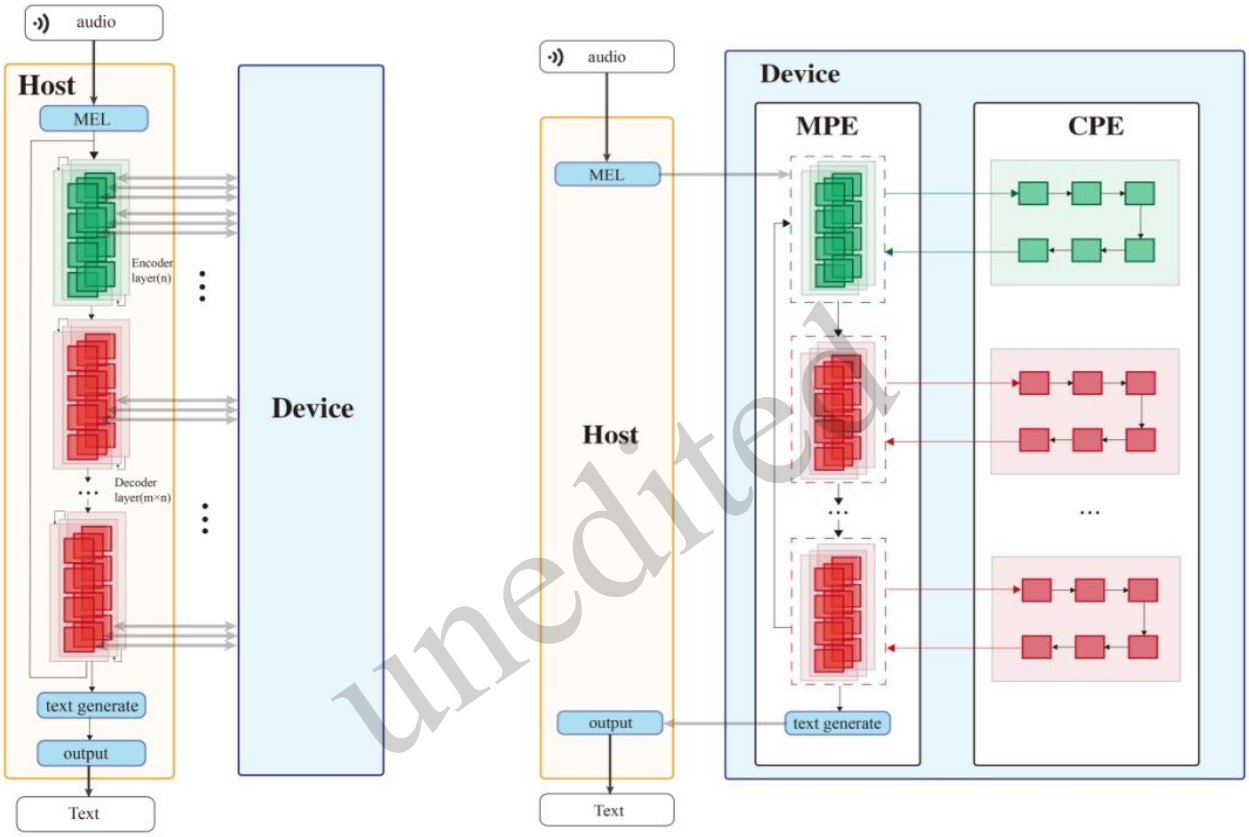
Following the comprehensive analysis detailed in the previous section, the primary sources of overhead in the Whisper inference process have been delineated into two main areas: model loading overhead and API invocation overhead. To address the model loading overhead, we propose an optimized fast loading methodology, meticulously designed to reduce this specific model loading expense.

In confronting the API invocation overhead, we

have developed a three-tier scheduling framework and implemented a zero-copy memory management model. These innovative improvements are designed to effectively reduce both launch and memory copy overheads, which together account for a significant portion of the total computational overhead.

egy of full-graph descent strategy for scheduling between MPE and CPE to further reduce the interaction overhead.

Fig. 4 presents a visual comparison between our proposed three-tier scheduling system and the conventional two-tier scheduling system utilized in PyTorch-GPU. In our innovative three-tier architec-



(a)two-tier scheduling system **(b)three-tier scheduling system**

Fig. 4 Comparison structure diagram of two scheduling systems

4.1 Three-tier scheduling framework

The launch overhead is a critical component of the total overhead within the transformer process. As the inference complexity increases, this overhead rises in a proportional manner. In a PyTorch-GPU environment, each operator triggers multiple launches, incurring substantial overhead and significantly contributing to the overall cost. To address this, we leveraged the scheduling capabilities of the MPE on the accelerator card and introduced a three-tier scheduling framework specifically designed to minimize launch overhead. Furthermore, we adopted the strategy of full-graph descent strategy for scheduling between MPE and CPE to further reduce the interaction overhead.

ture, as depicted in Fig. 4 (b), we identify three main components: Host, MPE, and CPE. The encoding phase is represented by the green segment, and the decoding phase is indicated by the red segment. Unlike the two-tier system, which requires launching each kernel across the PCIe interface, our three-tier system streamlines the workflow. It begins with a single PCIe launch operation after the completion of Mel processing, followed by the data transfer to the MPE for execution. Within the MPE, each encoding and decoding step is orchestrated by a single launch operation, using a computational graph to allocate tasks to the CPE. The CPE then executes these tasks

according to the internal structure of the computational graph, generates text, and conveys the results back to the host for the final presentation.

The three-tier scheduling framework integrates the CPU, MPE, and CPE, taking advantage of the MPE's robust logical control capabilities. This framework has successfully offloaded the complete transformer process from the CPU to the MPE, thereby transforming the traditional PCIe-based communication between the CPU and the device into a more efficient on-chip interaction between the MPE and the CPE. Following the completion of the model loading phase, all model data are seamlessly transferred from the host to the device. Subsequently, employing the MPE as the central logical controller, we commence the control process and activate the CPE to conduct kernel computations. The generated text is then efficiently copied back to the host for the final output. This architectural redesign effectively converts the conventional PCIe-crossed launch into a function call-like operation within the chip, significantly reducing the time required for a single launch only approximately one-tenth of its original duration—leading to a substantial enhancement in efficiency.

To further reduce the frequency of kernel launches, we have implemented a full-graph descent strategy within the scheduling paradigm between MPE and CPE. This approach uses the MPE to orchestrate a singular computational graph that includes multiple operators. Once constructed, a consolidated launch is executed, assigning numerous computational tasks to the CPE as an integrated computational graph rather than individual launches for each operation. Depending on the graph's size, this method can significantly reduce the number of launch operations, potentially to a few thousandths or even ten thousandths of the initial count.

It is important to acknowledge that the full-graph descent approach incurs some overhead during the graph construction process. To mitigate this, we have implemented an asynchronous execution strategy. While the CPE is computing the current graph, the MPE concurrently constructs the next graph, effectively overlapping part of the graph construction overhead. Ideally, we aim to fully parallelize these processes: constructing the decoder graph during the encoder's current round of computation and preparing the next encoder graph during the decoder's current round. In this optimal scenario, the entire inference process would only bear the cost of encoder graph construction once.

However, the size of the speech block for each inference task round is not static; it depends on the completion of the current decoder execution. This variability prevents the initiation of the next encoder graph construction before the conclusion of the current task round. To counter this challenge, we segment the encoder into self-attention and feed-forward network components. Within each time block, we initially construct the computation graph for the self-attention component and then strategically interweave the construction of the feed-forward network graph within the self-attention computation. Similarly, during the feed-forward network computation, we concurrently initiate the construction of the decoder's graph. Furthermore, we schedule the deferral of text generation from the current round to be executed by the CPE during the subsequent decoder computations, as illustrated in Fig. 5.

By adopting this methodology, we efficiently utilize the parallel capabilities of the CPE and MPE, markedly reducing the overhead introduced by graph construction. Table 3 illustrates the detailed comparative data on graph construction overhead before and after the optimization.

In contrast to conventional systems, we fully leverage the logical control capabilities of the accelerator card's MPE. This strategy primarily uses the MPE for the majority of scheduling control tasks, enabling the efficient offloading of the most sophisticated Whisper models onto the accelerator card, leading to a significant reduction in launch overhead.

Table 3 Time Breakdown Comparison

	base (jfk)	base(hp0)	large(jfk)	large(hp0)
synchronization	108 ms	2542 ms	534 ms	14895 ms
asynchronization	**53 ms**	**1144 ms**	**279 ms**	**7048 ms**

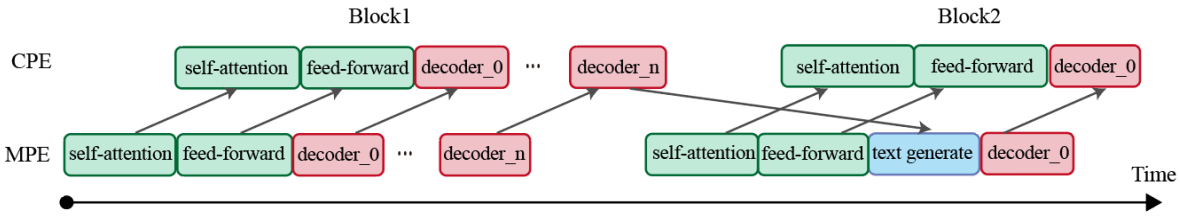


Fig.5 Asynchronous execution diagram

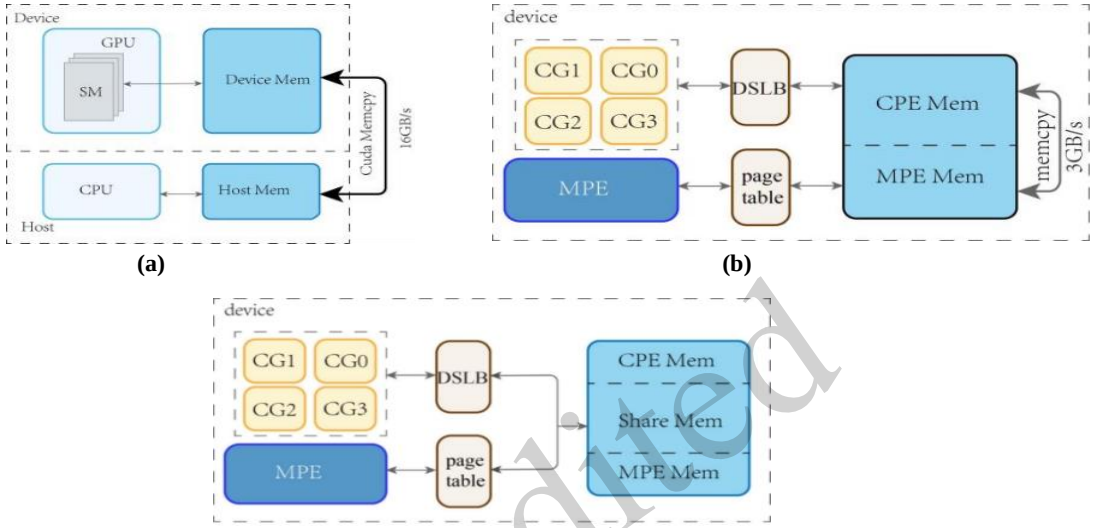


Fig. 6 Segment-page fusion Memory management mechanism

4.2 Segment-page fusion Memory management

The memcpy operation accounts for a significant component of the overhead within the transformer inference process. To mitigate this, we have utilized the inherent physical structure of both the MPE and the CPE facilitated by the ring network interconnections within the SWAI accelerator card. This approach has given rise to a zero-copy memory management technique, which maps separate virtual addresses of the MPE and CPE to a shared physical address, thereby effectively reducing the memcpy costs between the MPE and CPE.

As illustrated in Fig. 6(a), when utilizing a GPU for inference tasks, the storage devices between the device and host are not physically linked. Consequently, when the SMs (Streaming Multiprocessors) within the GPU require data from the Host's storage device, it must be transferred to the device via PCIe using CudaMemcpy, incurring additional overhead, especially with large models that necessitate frequent or substantial data copies.

In the SWAI accelerator, the memcpy overhead is divided into two main parts: the initial data transmission from host to device through the PCIe interface, similar to traditional GPU-based systems, and the internal data replication between the MPE and the CPE on the accelerator card. Fig. 6(b) illustrates the conventional memory management model where separate physical address ranges are allocated to the virtual address spaces of the MPE and CPE to maintain data consistency, security, and the prevention of resource contention. Despite sharing a common physical storage device, direct access to each other's memory resources is not allowed, requiring an on-card-memcpy operation for data exchange, which becomes a considerable overhead with frequent data exchanges.

The separation of the MPE and CPE memory space is primarily to prevent concurrent access or

modification of the same memory block. However, as shown in Fig. 2 and Fig. 5(a), the MPE and CPE follow an alternating serial execution pattern for Transformer model inference tasks. This execution paradigm involves the MPE in constructing the operational flow, the CPE in computation, and the MPE in logical processing for subsequent operations. In this scenario, there exists no potential for simultaneous access to the same physical memory by the MPE and CPE.

Leveraging this insight, we introduce a zero-copy memory management model tailored for Transformer inference tasks. Instead of the conventional separation of physical resources, we propose a shared memory segment, denoted 'share Mem' between the MPE and CPE, as shown in Fig.6(c). This allows both the MPE and CPE to access the same physical memory region, eliminating the need for on-card-memcpy and significantly reducing the data transfer overhead during inference.

Fig. 7 detailed the mechanics of our zero-copy memory management implementation using the segment page fusion technique.

The MPE, a general-purpose processor with control capabilities, uses page tables to manage virtual memory, while the CPE, a computational unit, relies on segment tables for virtual memory administration. A physical address is mapped to the MPE's virtual space through multi-level page tables, and the Dynamic Segment Lookaside Buffer (DSLB), or segment table, is used to map this address to the CPE's virtual space, enabling both the MPE and CPE to access the same data segment without the need for on-card-memcpy operations.

4.3 Fast loading process

After conducting a thorough examination of Whisper's model loading procedure, which includes four key components—model verification, loading model parameters, Whisper model initialization, and transferring the Whisper model to the device—we identified two specific areas with redundant calculations. These include unnecessary hash computations during the model verification phase and the superfluous parameter reset within the linear layer during Whisper model initialization. Specifically, hash verification consumed 7717.77 milliseconds, and model initialization required 10549.78 milliseconds, together accounting for 81% of the total loading time for the large model. A comparative analysis of the process, both before and afteroptimization, is detailed as follows.

Ensuring data integrity during model verification is achieved through hash verification, a process relying on cryptographic hash functions that transform variable-length input data into fixed-length hash values. Whisper utilizes the SHA-256 (Secure Hash Algorithm with a 256-bit output) hash function for these verification operations. However, we identified that performing hash verification on the entire model

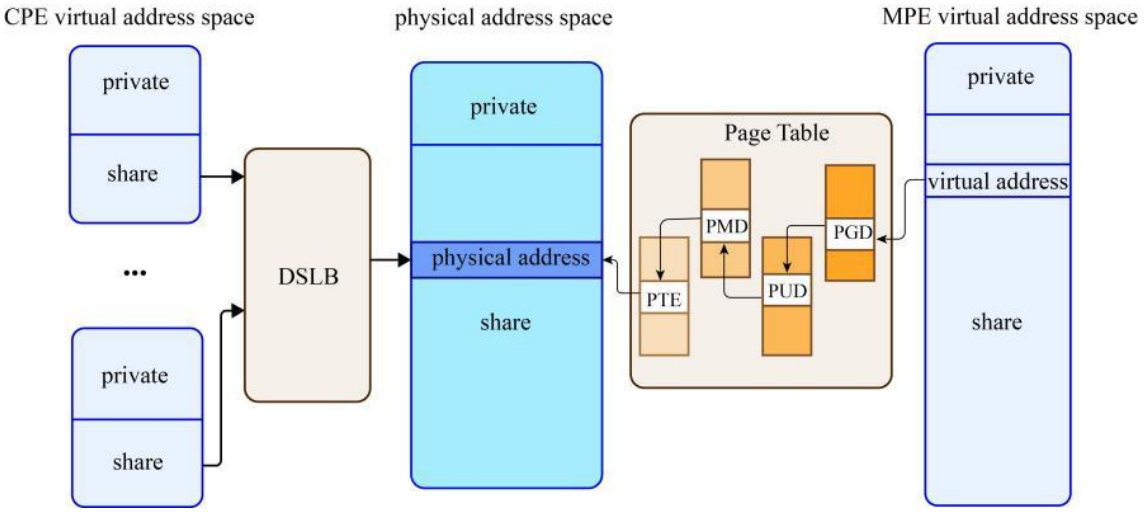


Fig. 7 Zero-copy memory management

content at each deployment introduces a significant computational overhead. Specifically, for the Whisper large model, this process requires approximately 7.71 seconds, which is double the duration required by loading model parameters. Moreover, as the model size increases, the verification time escalates proportionally. Recognizing that verifying the hash value during each deployment is not mandatory, we propose an innovative approach where verification is conducted solely during the model procurement phase, thus eliminating the need for repetitive verification during deployment.

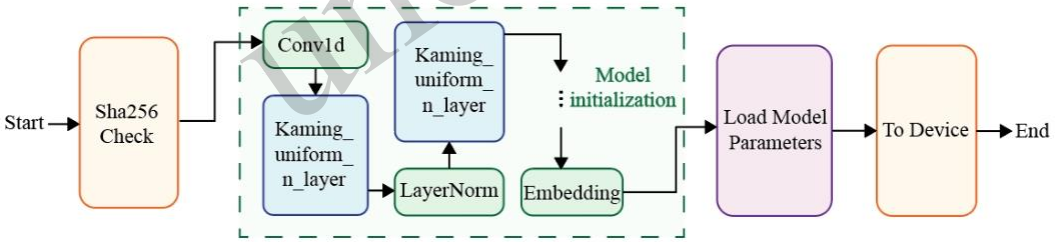
The Whisper model architecture, which includes both encoder and decoder layers along with components such as residual attention blocks and multi-head attention, undergoes an initialization phase that encompasses essential layers like linear and normalization layers. We denote the initialization times for the encoder and decoder as tEncoder andtDecoder , respectively, calculated using the following equations:

$$t_{Encoder} = 2 \times t_{Conv1d} + n_{layer} \times (6 \times t_{Linear} + 3 \times t_{LayerNorm}) + t_{LayerNorm} \quad (3)$$

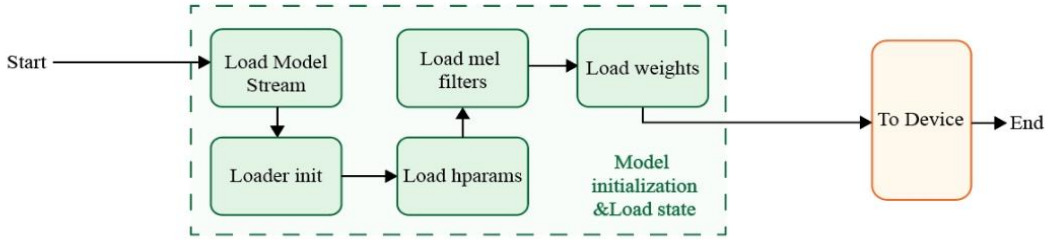
$$t_{Decoder} = t_{Embbending} + n_{layer} \times (10 \times t_{Linear} + 3 \times t_{LayerNorm}) + t_{LayerNorm} \quad (4)$$

Here, t_{Conv1d} denotes the initialization time of a 1D convolutional layer, t_{Linear} represents the initialization time of a linear layer, $t_{LayerNorm}$ is the initialization time of a layer normalization layer, and n_{layer} represents the number of residual attention blocks. The overall initialization time for the Whisper model is the sum of $t_{Encoder}$ and $t_{Decoder}$.

During initialization, the Whisper model inherits linear layers from PyTorch, which initializes learnable parameters using uninitialized data tensors. A parameter reset operation is employed to expedite convergence and mitigate overfitting, using the Kaiming_uniform_n_layer function for initialization. This is particularly beneficial for resuming training or proceeding to subsequent iterations, as it facilitates exploration of the search space and avoids entrapment in local minima. It is important to note that a parameter reset is not required during the inference process. Once initialized, the parameters of the Whisper model



(a) PyTorch model loading process



(b)Fast loading process

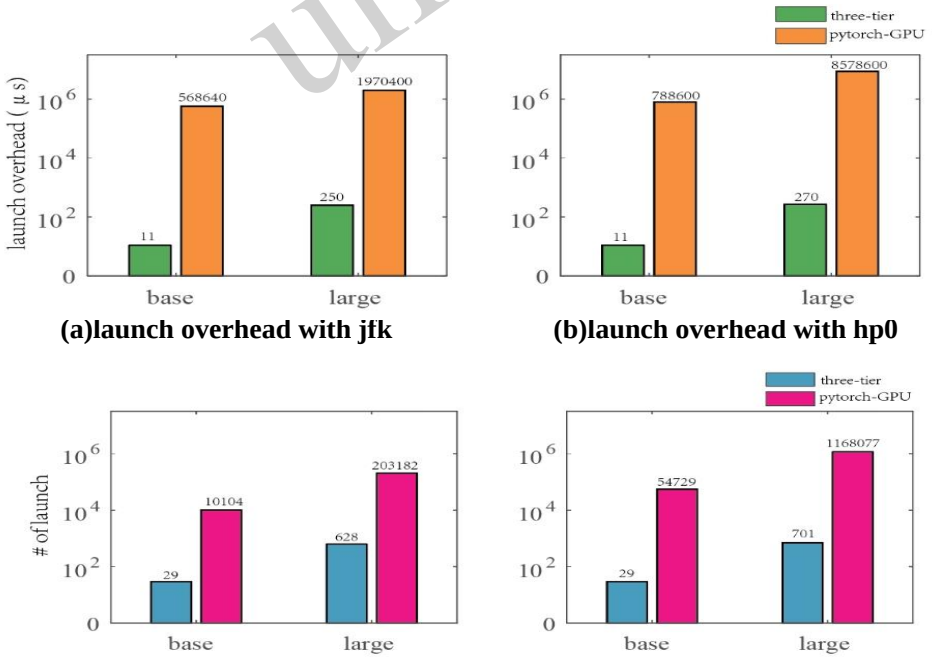
Fig. 8 Model loading process

are directly loaded from a pre-trained model, bypassing the parameter reset operation, ensuring that the parameters do not impact the accuracy of inference results.

To enhance the efficiency of model storage and retrieval, PyTorch traditionally decouples the loading of model parameters from the initialization of the Whisper model. However, this separation can lead to inefficiencies in model loading. An optimized approach integrates the initialization of the Whisper model with the loading of model parameters, reducing the overall model loading time.

Fig. 8(a) presents a schematic of the traditional model loading sequence in PyTorch, outlining the essential steps for model deployment. In contrast, Fig. 8(b) illustrates the fast model loading process implemented in our study, containing two critical modules: model initialization and loading, and the subsequent transmission of the model to the device, characterized by three significant improvements. First, we propose an innovative approach where verification is conducted solely during the model procurement phase. This strategy eliminates the need for repetitive verification during each deployment, thereby reducing unnecessary computational overhead. Second, we have identified that parameter reset, traditionally performed using the Kaiming_uniform_n_layer function for initialization, is not required during the inference process. Consequently, we have merged this initialization step into the model loading phase, streamlining the process and removing the need for separate parameter resetting. Third, our proposed approach enhances the model loading process by bypassing the intermediate storage of parameters, Mel filters, and weights in transient memory. Instead, these components are directly integrated into the Whisper model immediately after the read operation. This method of direct assignment eliminates the need for additional memory allocation, streamlining the loading sequence and aligning it with the performance expectations of contemporary AI applications. This refined methodology, as depicted in Fig. 9(b), strategically integrates model initialization with parameter loading into a singular, efficient step.

Collectively, these refinements reduce the memory footprint and shorten the model loading duration, making the model loading procedure more efficient and better suited to meet the rigorous requirements of high-performance AI applications.



(a)launch overhead with jfk **(b)launch overhead with hp0**

(c)# of launch with jfk **(d)# of launch with hp0**

Fig.9. Launch overhead and # for jfk and hp0 with base and large Whisper model

5 Experiments

5.1 Experimental setup

Our experiments aimed to evaluate the inference overhead of the Whisper model, utilizing a server equipped with a SWAI accelerator card and a Tesla V100S GPU. The server features an Intel (R) Xeon (R) Silver 4214 processor, boasting 24 cores and 48 threads, operating at a clock speed of 2.2GHz.

We designed experiments to perform a comparative analysis of the model loading times and launch overhead and the memcpy operation within the Transformer process. This analysis was conducted under both original and optimized conditions, implementing various strategies to reduce overhead. The final section of our experiments assessed the overall impact of these optimization approaches on reducing the operational costs of the Whisper model. These evaluations aimed to quantify the benefits of each optimization technique and to provide a comprehensive assessment of their combined effect on enhancing the model's efficiency.

5.2 Launch overhead.

This section evaluates the effectiveness of our proposed three-tier scheduling framework in reducing launch overhead. Comparative tests were conducted on two audio files, JFK and HP0, with varying durations, and both the base and large Whisper models. Detailed results are presented in Fig. 9.

Figs. 9(a) and (b) illustrate the significant reduction in launch overhead with our approach. For the base model, the launch overhead for JFK and HP0 audio was dramatically decreased from 56,8640 microseconds(μs) and 788,600 μs to approximately 11 μs. Similarly, for the large model, the launch overhead for JFK and HP0 audio was reduced from 1,970,400 μs and 8,578,600 μs to 250 μs and 270 μs, respectively. The data clearly show that our methodology can reduce the originally substantial launch overhead to a negligible level, exhibiting remarkable effectiveness across various model sizes and audio lengths.

Our three-tier scheduling scheme effectively converts the launch expense associated with each graph computation from a PCIe communication overhead to the efficiency of function calls, resulting in a significant reduction in launch overhead. Notably, the launch cost per unit time in our framework is approximately 0.5 μs, which is markedly lower than the typical launch cost of 4.9 μs associated with CUDA operations. As the complexity of the transcription tasks and the sophistication of the model architecture increase, the enhancement potential offered by our optimization strategy is expected to increase correspondingly.

5.3 Memory copy overhead

This section evaluates the effectiveness of our proposed method in reducing the memory copy overhead. Within the PyTorch framework, the memcpy operation's duration has been observed to escalate with the increase in model size and the length of audio tasks. As depicted in Table 4, during inference on the audio task HP0 with the large model, the memcpy cost soars to 749ms, making it the third most time-consuming element within the entire inference process.

Table 4 Memcpy Time Breakdown Comparison

	PyTorch-GPU	SWAI			
		Before		After	
		H-to-D	MPE-to-SPE	H-to-D	MPE-to-SPE
base (jfk)	**30ms**	45.92ms	1.29ms	44.63ms	**0.3us**
base (hp0)	107ms	**75.76ms**	27.5ms	48.71ms	**0.4us**
large (jfk)	483ms	**49.03ms**	1.22ms	47.81ms	**0.2us**
large (hp0)	749ms	**79.93ms**	29.2ms	50.73ms	**0.5us**

Originally, the memcpy overhead in the SWAI is composed of two distinct elements. However, the integration with zero-copy technology post-optimization has effectively limited this overhead to a maximum of 50 milliseconds—a significant reduction when compared to overheads typically observed in GPU-based systems. Furthermore, thedeployment of this technique has notably mini-

mized the data transfer time between the MPE and the CPE, nearly eliminating the latency associated with traditional data copying.

5.4 Model loading overhead

In the traditional PyTorch framework, the model loading process is typically divided into the following four distinct stages: model validation, model parameter loading, model initialization, and model-to-device transfer. Our optimized implementation streamlines this process by eliminating the model validation step during deployment, opting to perform this check at the time of model acquisition. This strategy not only simplifies the deployment phase but also ensures model integrity before operational use.

Initially, the standard PyTorch implementation requires separate stages for loading model parameters and initializing the Whisper model. Our enhancements have consolidated these stages, enabling the simultaneous initialization of the Whisper model alongside the loading of model parameters, as de

tailed in Table 5. This streamlined model loading approach results in a significant reduction in the overall loading duration. Specifically, for the base model, the loading time has been reduced from its original 2317.6 ms to 186.57 ms. For the large model, the improvement is even more pronounced, with the loading time decreasing from 22,128.31 ms to 1041.72 ms. These enhancements underscore the efficacy of our optimization strategy in enhancing the efficiency of the model loading process.

5.5 Overall overhead

The cumulative overhead results are clearly illustrated in Fig. 10, which provides a visual representation of the significant reduction in control process overhead during inference execution achieved by our method. In addition, our strategy exhibits an enhanced performance for larger models. Typically, the inference execution overhead incurred by our method is less than half of that experienced with PyTorch CUDA, and under the most favorable conditions, our overhead is merely about 5% of the overhead associated with PyTorch.

Table 5 Model Loading Time Breakdown Comparison

model	method	model validation/ms	parameters load/ms	model initialization/ms	model to device/ms	load model total/ms
base	PyTorch	310.67	1417.31	490.90	98.72	2317.6
	Fast loading	**0**		**116.20**	70.37	**186.57**
large	PyTorch	6646.13	2812.65	10995.37	1674.16	22128.31
	Fast loading	**0**		**303.60**	738.12	**1041.72**

It is noteworthy that, in comparison to the PyTorch-GPU, we have included the overhead associated with graph construction during the development of the computation graph on the MPE. However, as depicted in Fig. 10, when utilizing JFK for inference, the overhead due to graph construction is less than one-tenth of the API invocation overhead. On the contrary, when employing HP0 for inference, the most extreme scenario results in an overhead that is approximately half of the API invocation overhead, indicating that graph construction does not incur an excessively increased time overhead.

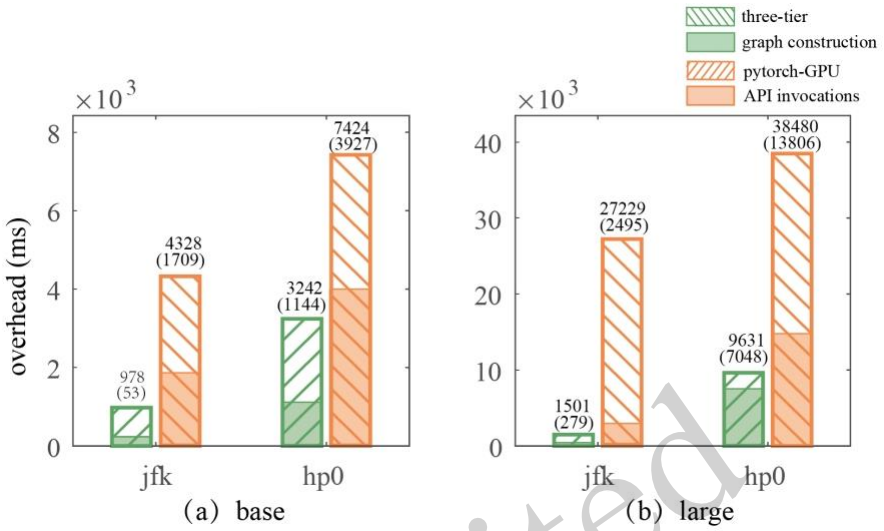


Fig.10. Overall overhead comparison

Our research has specifically targeted the often-overlooked issue of inference overhead, aiming to alleviate the overall overhead associated with Transformer inference execution. The results of our empirical study confirm the effectiveness of our approach in reducing executing overhead. Together, these findings collectively validate the positive impact of our strategy on decreasing the computational costs related to transformer inference.

6 Conclusions

In this paper, we have conducted a comprehensive analysis of the overhead composition in the Transformer inference process and proposed a novel approach to minimize this overhead using MPE on the SWAI Accelerator. Our contributions to the domain of Transformer model optimization are twofold: addressing the challenge of minimizing inference overhead and facilitating more expeditious and efficient inference processes by the user of the logical control capabilities of AI accelerators, and implementing sophisticated memory management techniques. These enhancements are expected to provide significant advantages to a wide range of applications reliant on Transformer models, including but not limited to Natural Language Processing, machine translation, and speech recognition.

Looking ahead, our future endeavors will explore additional optimization techniques and examine the scalability of our approach when applied to larger Transformer models. Furthermore, we also plan to evaluate the performance of our solution across a variety of AI accelerator architectures and continually refine our methodologies to further enhance inference efficiency. Collectively, our research opens new avenues for improving the inference performance of Transformer models in practical applications.

Contributors

Yulong ZHAO designed the research, also contributed to the experimental design and wrote significant portions of the manuscript . Chunzhi WU and Lufei ZHANG were responsible for the statistical analysis of the data and the creation of the figures and tables. Yizhuo WANG, Yaguang ZHANG, Wenyuan SHEN, Fan HAO were responsible for the experi-

mental design, data collection, and analysis. Hankang FANG and Qin Yi contributed to the literature review and background research for the study. Xin Liu helped organize the manuscript. Chunzhi WU, Yizhuo WANG, Yaguang ZHANG and Lufei ZHANG revised and finalized the paper. All authors contributed to manuscript revision, read, and approved the submitted version.

Conflict of interest

All the authors declare that they have no conflict of interest.

Data availability

Data available on request from the authors. The data that support the findings of this study are available from the corresponding author,upon reasonable request.

References

Arafa Y, Badawy AHA, Chennupati G, et al., 2019. Low overhead instruction latency characterization for NVIDIA GPGPUs. 2019 IEEE High Performance Extreme Computing Conf (HPEC), p.1-8. https://doi.org/10.1109/HPEC.2019.8916466

Baevski A, Zhou H, Mohamed A, et al., 2020. wav2vec 2.0: A framework for self-supervised learning of speech representations. Proc 34th Int Conf on Neural Information Processing Systems, p.12449-12460.

Boudier P, Sellers G, 2011. Memory system on fusion APUs: The benefits of zero copy. AMD Fusion Developer Summit, AMD. https://ia803207.us.archive.org/7/items/AFDS2011/1004_final.pdf

Chen GY, Shen XP, 2015. Free launch: Optimizing GPU dynamic kernel launches through thread reuse. 2015 48th Annual IEEE/ACM Int Symp on Microarchitecture (MICRO), p.407-419. https://doi.org/10.1145/2830772.2830818

Chen SY, Huang SY, Pandey S, et al., 2021. E.T.: Re-thinking self-attention for transformer models on GPUs. Proc Int Conf for High Performance Computing, Networking, Storage and Analysis, p.1-14. https://doi.org/10.1145/3458817.3476138

Chu CH, Khorassani KS, Zhou QH, et al., 2020. Dynamic kernel fusion for bulk non-contiguous data transfer on GPU clusters. 2020 IEEE Int Conf on Cluster Computing (CLUSTER), p.130-141. https://doi.org/10.1109/CLUSTER49012.2020.00023

Dai GH, Huang TH, Chi YZ, et al., 2019. GraphH: A processing-in-memory architecture for large-scale graph processing. *IEEE Trans Comput-Aided Des Integr Circuits Syst*, 38(4):640-653. https://doi.org/10.1109/TCAD.2018.2821565

Dao T, Fu DY, Ermon S, et al., 2022. FLASHATTENTION: Fast and memory-efficient exact attention with IO-awareness. Proc 36th Int Conf on Neural Information Processing Systems, p.16344-16359.

Devlin J, Chang MW, Lee K, et al, 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. https://arxiv.org/abs/1810.04805

Du JS, Liu ZM, Fang JR, et al., 2022. EnergonAI: An inference system for 10-100 billion parameter transformer models. https://arxiv.org/abs/2209.02341

Fang JR, Yu Y, Zhao CD, et al., 2021. TurboTransformers: An efficient GPU serving system for transformer models. Proc 26th ACM SIGPLAN Symp on Principles and Practice of Parallel Programming, p.389-402. https://doi.org/10.1145/3437801.3441578

Fujii Y, Azumi T, Nishio N, et al., 2013. Data transfer matters for GPU computing. 2013 Int Conf Parallel and Distributed Systems, p.275-282. https://doi.org/10.1109/ICPADS.2013.47

Huawei, 2020. DaVinci: A scalable architecture for neural network computing. https://www.cmc.ca/wp-content/uploads/2020/03/Zhan-Xu-Huawei.pdf.

Kim S, Oh S, Yi Y, 2021. Minimizing GPU kernel launch overhead in deep learning inference on mobile GPUs. Proc 22nd Int Workshop on Mobile Computing Systems and Applications, p.57-63. https://doi.org/10.1145/3446382.3448606

Kim YJ, Awadalla HH, 2020. Fastformers: Highly efficient transformer models for natural language understanding. https://arxiv.org/abs/2010.13382

Lee K, Lin HS, Feng WC, 2013. Performance characterization of data-intensive kernels on AMD fusion architectures. *Comput Sci Res Dev*, 28(2):175-184. https://doi.org/10.1007/s00450-012-0209-1

Ma X, Li GL, Liu L, et al., 2021. Understanding the runtime overheads of deep learning inference on edge devices. 2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom), p.390-397. https://doi.org/10.1109/ISPA-BDCloud-SocialCom-SustainCom52081.2021.00061

Mittal S, 2020. A survey on evaluating and optimizing performance of Intel Xeon Phi. *Concurr Comput*, 32(19):e5742. https://doi.org/10.1002/cpe.5742

Ouyang J, Noh M, Wang Y, et al., 2020. Baidu Kunlun: An AI processor for diversified workloads. 2020 IEEE Hot Chips 32 Symposium (HCS), p.1-18. https://doi.org/10.1109/HCS49909.2020.9220641

Patel S, Hwu WMW, 2008. Accelerator architectures. *IEEE Micro*, 28(4):4-12. https://doi.org/10.1109/MM.2008.50

Peccerillo B, Mannino M, Mondelli A, et al., 2022. A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives. *J Syst Architect*, 129:102561. https://doi.org/10.1016/j.sysarc.2022.102561

Radford A, Wu J, Child R, et al., 2019. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8):9.

Radford A, Kim JW, Xu T, et al., 2023. Robust speech recognition via large-scale weak supervision. Proc 40th Int Conf on Machine Learning, p.28492-28518.

Sodani A, Gramunt R, Corbal J, et al., 2016. Knights landing: Second-generation Intel Xeon Phi product. *IEEE Micro*, 36(2):34-46. https://doi.org/10.1109/MM.2016.25

Stevens JR, Venkatesan R, Dai S, et al., 2021. Softermax: Hardware/software co-design of an efficient softmax for transformers. 2021 58th ACM/IEEE Design Automation Conf (DAC), p.469-474. https://doi.org/10.1109/DAC18074.2021.9586134

Sunitha NV, Raju K, Chiplunkar NN, 2017. Performance improvement of CUDA applications by reducing CPU-GPU data transfer overhead. 2017 Int Conf on Inventive Communication and Computational Technologies (ICICCT), p.211-215. https://doi.org/10.1109/ICICCT.2017.7975190

Sze V, Chen YH, Yang TJ, et al., 2017. Efficient processing of deep neural networks: A tutorial and survey. http://arxiv.org/abs/1703.09039

Touvron H, Lavril T, Izacard G, et al., 2023. LLaMA: Open and efficient foundation language models. https://arxiv.org/abs/2302.13971

Vaswani A, Shazeer N, Parmar N, et al., 2017. Attention is all you need. Proc 31st Int Conf on Neural Information Processing Systems, p.6000-6010.

Wang XH, Xiong Y, Wei Y, et al., 2021. LightSeq: A high performance inference library for transformers. http://arxiv.org/abs/2010.13887

Wechsler O, Behar M, Daga B, 2019. Spring hill (NNP-I 1000) Intel's data center inference chip. 2019 IEEE Hot Chips 31 Symp (HCS), p.1-12. https://doi.org/10.1109/HOTCHIPS.2019.8875671

Zhang LQ, Wahib M, Matsuoka S, 2019. Understanding the overheads of launching CUDA kernels. 2019 Int Conf on Parallel Processing (ICPP), p.5-8.