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]*National Supercomputing Center in Wuxi, Wuxi 214000, China*
[4]*Zhejiang Lab, Hangzhou 310000, China*
[5]*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 Mar. 4, 2025

**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 artificial intelligence (AI) accelerators. Our work is anchored by four key contributions. First, we conduct a comprehensive analysis of the overhead composition within the transformer inference process, identifying the primary bottlenecks. Second, we leverage the management processing element (MPE) of the Shenwei AI (SWAI) accelerator, implementing a three-tier scheduling framework that significantly reduces the number of host-device launches to approximately 1/10 000 of the original PyTorch-GPU setup. Third, we introduce a zero-copy memory management technique using segment-page fusion, which significantly reduces memory access latency and improves overall inference efficiency. Finally, we develop 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 ms to 1041.72 ms. 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:** TP181

## 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., 2019), GPT-2 (Radford et al., 2019), LLaMA (Touvron et al., 2023), wav2vec2.0 (Baevski et al., 2020), and whisper (Radford et al., 2023). These transformer models have significantly advanced state-of-the-art models 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.

‡ Corresponding author
ORCID: Yulong ZHAO, https://orcid.org/0009-0003-2291-9499; Xin LIU, https://orcid.org/0000-0002-7870-6535

The first pertains to the optimization of specific operators, such as Softermax (Stevens et al., 2021) and FLASHATTENTION (Dao et al., 2022), and involves reducing accuracy and minimizing input/output (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 YJ et al., 2020), knowledge distillation (Fang et al., 2021), operator fusion, memory management (Chen SY et al., 2021), and parallel scheduling strategies (Du 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 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 uses the hardware control mechanisms of artificial intelligence (AI) accelerators. We begin 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 introduce a fast loading method that significantly reduces the overhead caused by loading models with PyTorch-GPU. We also develop a three-tier scheduling framework for interacting with the host and the controlling element on the accelerator, leveraging the accelerator's scheduling capabilities. To minimize device's launch expenditures, we embrace a holistic full-graph optimization strategy. Additionally, we implement 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 undertake a detailed analysis of the overhead constituents within the inference process, categorizing them into three main segments: model loading, application programming interface (API) invocations, and ancillary factors. We find that model loading and API invocations account for 96% of the total overhead, providing crucial insights for developing strategies to reduce the overhead.

2. We leverage the scheduling capabilities of the management processing element (MPE) on the Shenwei AI (SWAI) accelerator, implementing a three-tier scheduling framework that reduces the number of launch operations to 1/10 000 of the baseline.

3. We design 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's 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 reduces the total loading duration for the large model from 22 128.31 ms to a mere 1041.72 ms.

## 2 Background and related works

### 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.

Whisper, the most recent addition to the transformer family, 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 closely adheres 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-s audio segments. Transcribing longer sounds requires the prediction of timestamps and the segmentation of the Mel-spectrogram into multiple 30-s 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 launch operations 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 graphic processing unit (GPU)-based inference has been highlighted in recent scholarly work. These studies emphasized the need to address this overhead as it is critical to the performance of such networks. For instance, Kim S et al. (2021) revealed that the kernel launch overhead of lightweight neural networks on mobile GPUs is notably high. Fujii et al. (2013) discussed the challenges posed by the heterogeneity of GPU computing in data transmission, while Arafa et al. (2019) presented improvements in the performance of CUDA applications by reducing central processing unit (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 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 et al. (2021) provided an in-depth analysis of the runtime overhead 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 GY and Shen (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 et al. (2020) proposed a novel approach to achieve low latency and high bandwidth by dynamically fusing packing/unpacking GPU kernels to reduce expensive kernel launch overhead. Fujii et al. (2013) analyzed data transfer concerns for GPUs and characterized currently achievable data transfer methods in cutting-edge GPU technology. Sunitha et al. (2017) explored the effects of overlapping data transfer and kernel execution on the overall execution time of CUDA applications. Lee et al. (2013) verified various memory access technologies using different memory bindings on the AMD fusion system (Llano A8-3850). Boudier and Sellers (2011) explored

the impact of memory fusion on CPU–GPU heterogeneous computing.

Fang et al. (2021) and Wang et al. (2021) recognized the overhead caused by frequent kernel launches during runtime and effectively addressed it through operator fusion. LightSeq, as described by Wang et al. (2021), 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 general matrix multiplication (GEMM) operations. This approach reduced the number of launch operations per encoder layer to just six. Similarly, TurboTransformers, detailed in Fang 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, they have not eliminated this overhead entirely. 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 et al., 2017). 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 and Hwu, 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 et al., 2022). For instance, Huawei's Ascend series integrate 16 ARM cores based on the DaVinci architecture (Huawei, 2020), while Intel's Xeon Phi (Sodani et al., 2016; Mittal, 2020) and Intel Nervana NNP-I (Wechsler et al., 2019) both include x86 controller cores within their design. GraphH (Dai et al, 2019) 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 et al., 2020).

The SWAI accelerator exemplifies the manycore approach, which has proven to be an effective evolution from the manycore paradigm, emphasizing an even greater degree of parallelism (Peccerillo 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 core groups (CGs, also known as the slave core groups), and dual-mode peripheral component interconnect express (PCIe) components. Each CG contains a high bandwidth memory (HBM) 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 direct memory access (DMA) engine on the PCIe module facilitates efficient data transfer between the host and accelerator card.

## 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 overhead. 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

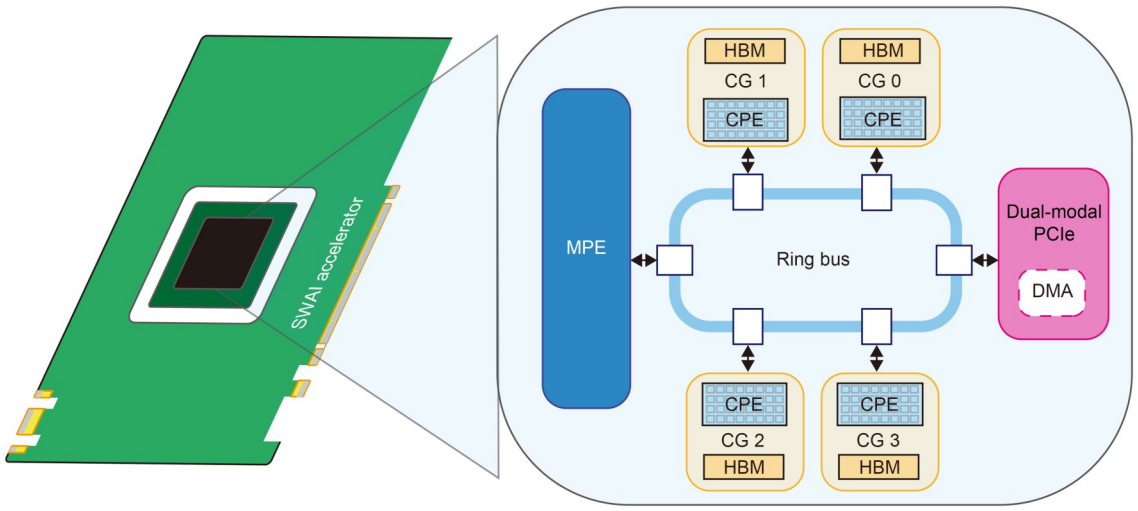


**Fig. 1 SWAI accelerator architecture**

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 a Nvidia Tesla V100S GPU.

### 3.1 Overhead analysis

Fig. 2 presents a 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-s 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.

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 launches or memcpy operations. For instance, the Conv1D operator involves a series of direct operations, including Copy_Kernel_Cuda, the nchwToNhwcKernel executed twice, the CUDAFunction_Add performed once, and the 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, contribute to the costs of the interaction between the host and device.

Following a comprehensive theoretical analysis of the whisper model's inference process, we proceed to conduct empirical experiments to evaluate the model's overall inference duration both on the base model (approximately 74 million parameters) and the large model (approximately 1550 million parameters). The experiments use the following official audio

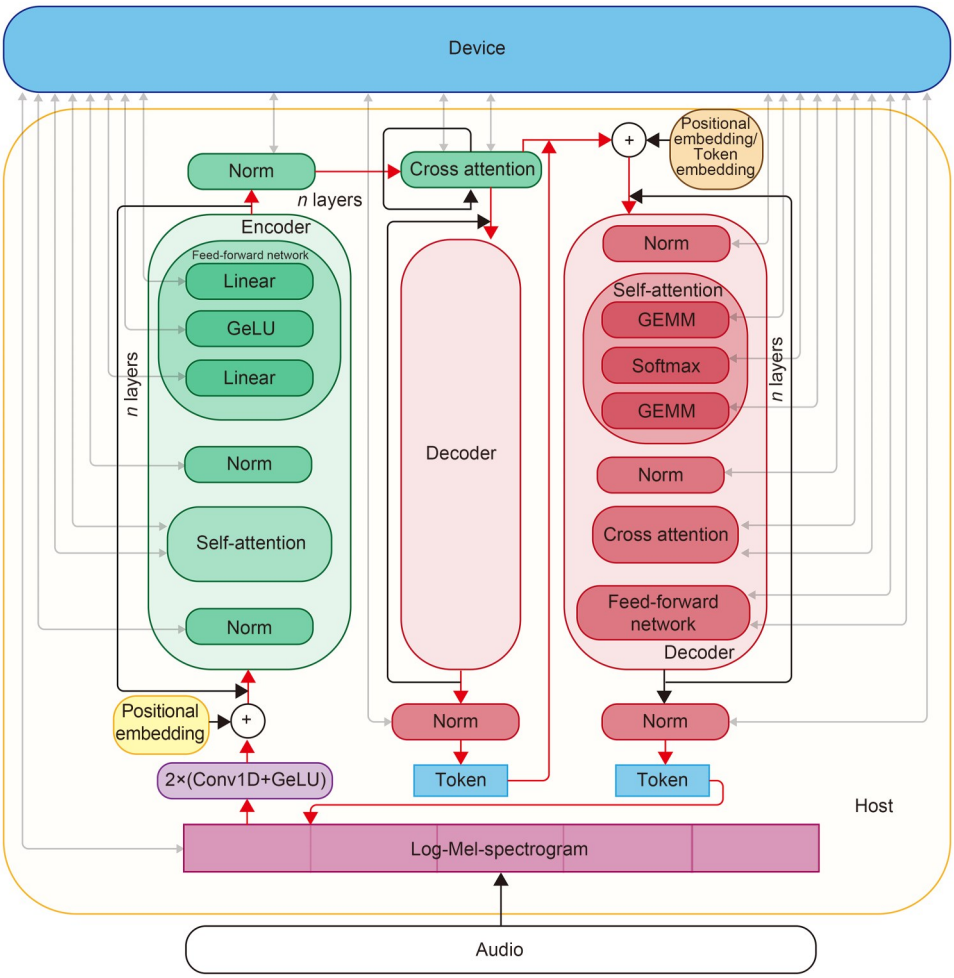


**Fig. 2 Interactions between the host and device. References to color refer to the online version of this figure**

samples: JFK.wav with an 11-s speech segment and HP0.wav with a 260-s speech segment. The experimental data, combined with the theoretical analysis, segment 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 Fig. 3a illustrates the base model's duration breakdown with JFK.wav, Fig. 3b depicts the large model's duration breakdown with JFK.wav, Fig. 3c details the base model's duration breakdown with HP0.wav, and Fig. 3d 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

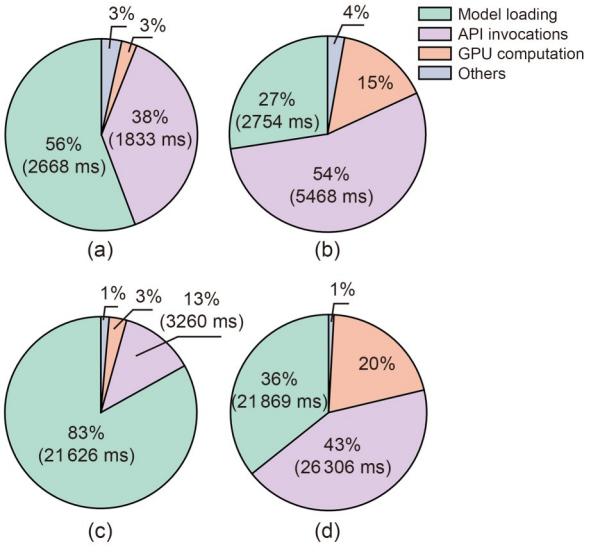


**Fig. 3 Breakdown of whisper inference duration. Only the integer portion of the statistics has been retained. References to color refer to the online version of this figure**

size and processing task length, becoming the main time-consuming element in prolonged audio inference. Together, these two components account for 79%–96% of the total overhead, highlighting the necessity for detailed analysis to better understand their performance characteristics. Further exploration 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 is 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's loading is significantly affected by model verification and initialization.

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

The model parameters are subsequently imported using 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 each consist of 32 layers, 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 ms per layer, totaling 5.6 s. This initialization step accounts for 53.13% of the overall whisper model initialization duration of 10.54 s.

### 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. Using the analytical tool nvprof, we conduct 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.

Extensive empirical test 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

**Table 1 Model loading breakdown table**

| Model | Time (ms) | | | | |
|---|---|---|---|---|---|
| | Model validation | Model parameter loading | Whisper model initialization | Whisper model-to-device | Load model total |
| Base | 399.70 | 1409.16 | 561.81 | 98.77 | 2469.44 |
| Large | 7717.77 | 3051.17 | 10 549.78 | 1215.35 | 22 534.07 |

**Table 2 Details of API invocations**

| Model (input) | | KE | CSIC | CF | CLK | CMU | CSCWF | CMA | CDGA | CGD |
|---|---|---|---|---|---|---|---|---|---|---|
| Base (JFK) | Time | | 4 | 4 | 10 104 | 443 | 16 | 207 | 3136 | 102 529 |
| | Duration (ms) | 127.13 | 63.06 | 625.32 | 568.64 | 182.26 | 127.71 | 30 | 16.5 | 36.34 |
| | Proportion (%) | 7.15 | 3.55 | 35.19 | 32.00 | 10.26 | 7.19 | 1.69 | 0.93 | 2.04 |
| Base (HP0) | Time | | 28 | 4 | 203 182 | 443 | 16 | 4131 | 3136 | 1 880 511 |
| | Duration (ms) | 1541 | 83 | 624.42 | 1970.4 | 195.2 | 118.56 | 107 | 12.6 | 534.93 |
| | Proportion (%) | 29.71 | 1.60 | 12.04 | 37.99 | 3.76 | 2.29 | 2.06 | 0.24 | 10.31 |
| Large (JFK) | Time | | 8 | 4 | 54 729 | 443 | 16 | 212 | 3136 | 534 280 |
| | Duration (ms) | 765 | 109 | 501 | 788.6 | 252.55 | 162.56 | 483 | 25.55 | 178.84 |
| | Proportion (%) | 23.42 | 3.34 | 15.34 | 24.14 | 7.73 | 4.98 | 14.79 | 0.78 | 5.48 |
| Large (HP0) | Time | | 61 | 4 | 1 168 077 | 443 | 16 | 4635 | 3136 | 10 418 693 |
| | Duration (ms) | 12 500 | 47 | 452 | 8578.6 | 309 | 143 | 749 | 19 | 3327 |
| | Proportion (%) | 47.85 | 0.18 | 1.73 | 32.84 | 1.18 | 0.55 | 2.87 | 0.07 | 12.73 |

KE: kernel execution; CSIC: CudaStreamIsCapturing; CF: CudaFree; CLK: CudaLaunchKernel; CMU: CuModuleUnload; CSCWF: CudaStreamCreateWithFlags; CMA: CudaMemcpyAsync; CDGA: CuDeviceGetAttribute; CGD: CudaGetDevice

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 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 invocation overhead, accounting for 40.06% of the 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 conduct a detailed analysis of the frequency and timing of CudaLaunchKernel invocations. Our investigation into the launch duration reveals that the initial kernel launches at the commencement of program execution exhibit a notably longer duration, on the order of hundreds of milliseconds. In contrast, the average duration for each subsequent launch is significantly reduced, averaging around 4.5 ms.

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:

$$\text{launch\_time}=n\text{layer}_{\text{encode}}\beta_{\text{encode}}+m\text{layer}_{\text{decode}}\beta_{\text{decode}}, \quad (1)$$

$$\text{launch\_time}=112t_1n_{\text{layer}}. \quad (2)$$

Eq. (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, $\text{layer}_{\text{encode}}$ and $\text{layer}_{\text{decode}}$ represent coefficients related to the encoding and decoding layer structure, respectively. The quantity of these layers is typically prescribed by the model architecture. Additionally, $\beta_{\text{encode}}$ indicates the interaction time per encoding layer, while $\beta_{\text{decode}}$ refers to the interaction time per decoding layer.

In Eq. (2), $t_1$ symbolizes the input task duration in seconds, and $n_{\text{layer}}$ refers to the number of layers.

Based on empirical data, a 30-s 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 Eq. (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. After decoding, the decoder applies greedy algorithms or beam_search algorithms to refine the optimal outcomes, calculates probabilities from the resulting matrix, and selects the 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 Section 3, 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 overhead and memory copy overhead, which together account for a significant portion of the total computational overhead.

### 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 launch operations, incurring substantial overhead and significantly contributing to the overall cost. To address this, we leverage the scheduling capabilities of the MPE on the accelerator card and introduce a three-tier scheduling framework specifically designed to minimize launch overhead. Furthermore, we adopt the full-graph descent strategy for scheduling between the MPE and the 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 used in PyTorch-GPU. In our innovative three-tier architecture, as depicted in Fig. 4b, 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

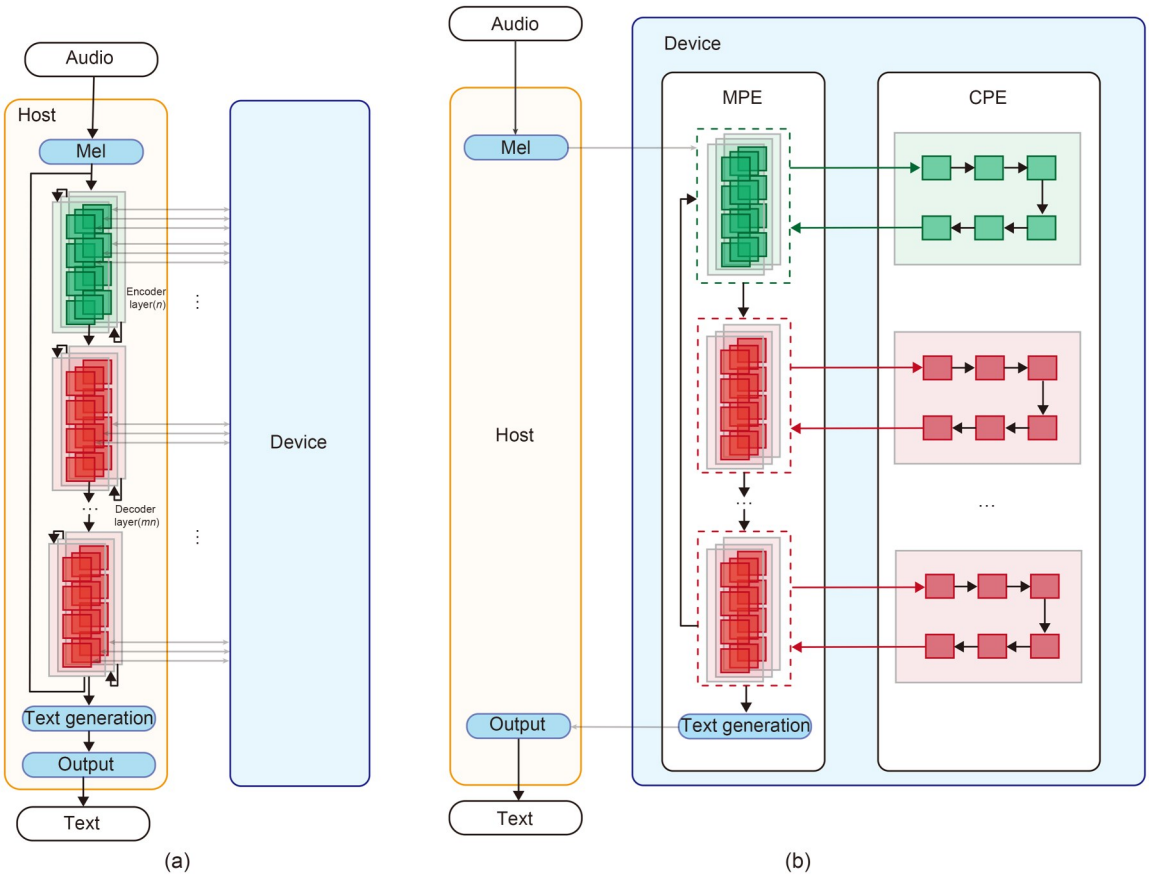


**Fig. 4 Structure comparison of two scheduling systems: (a) two-tier scheduling system; (b) three-tier scheduling system. References to color refer to the online version of this figure**

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, reducing the time required for a single launch to approximately one-tenth of its original duration, which leads 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 the MPE and the 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 bear only 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 use 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 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.

### 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 used the inherent physical structure of both the MPE and 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. 6a, when using a GPU for inference tasks, the storage devices between the device and host are not physically linked. Consequently, when the streaming multiprocessors (SMs) within the GPU require data from the host's storage device, the data 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 CPE on the accelerator card. Fig. 6b illustrates the conventional memory management model, where separate physical address ranges are allocated to the virtual address spaces of the MPE and the 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. 4a, the MPE and CPE follow an

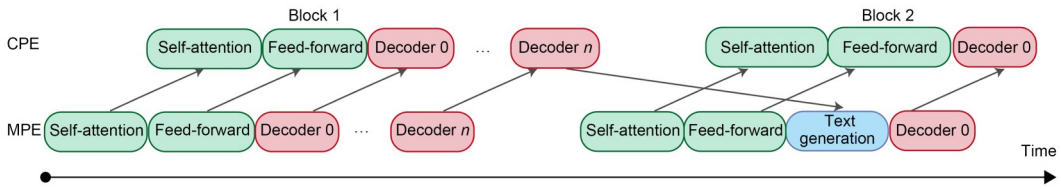


**Fig. 5 Asynchronous execution diagram**

**Table 3 Time breakdown comparison**

| | Time (ms) | | | |
|---|---|---|---|---|
| | Base (JFK) | Base (HP0) | Large (JFK) | Large (HP0) |
| Synchronization | 108 | 2542 | 534 | 14 895 |
| Asynchronization | 53 | 1144 | 279 | 7048 |

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 "Shared Mem" between the MPE and CPE, as shown in Fig. 6c. 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 details the mechanics of our zero-copy memory management implementation using the segment-page fusion technique.

MPE, a general-purpose processor with control capabilities, uses page tables to manage virtual memory, while CPE, a computational unit, relies on segment tables for virtual memory administration. A physical

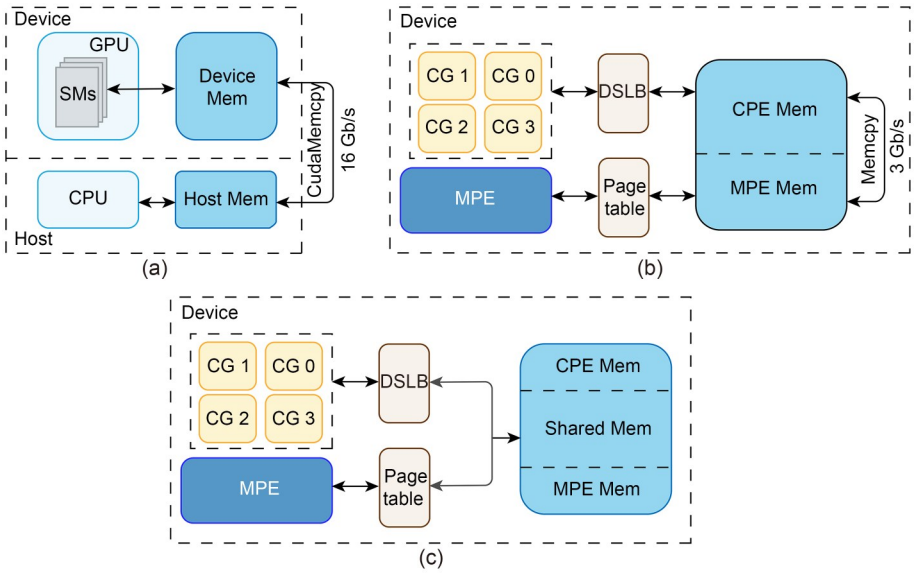


**Fig. 6 Segment-page fusion memory management mechanism: (a) CudaMemcpy via PCIe; (b) on-card data exchange; (c) shared Mem mechanism**

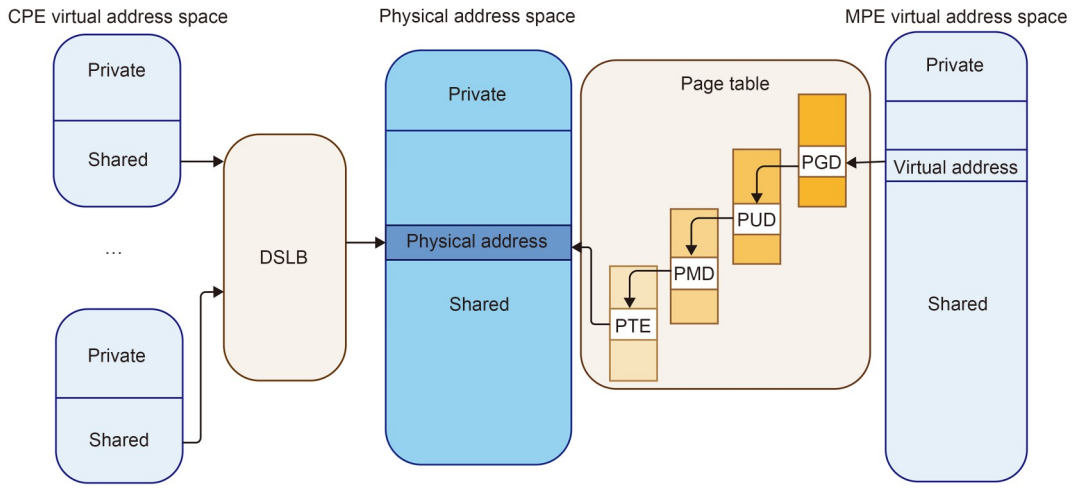


**Fig. 7 Zero-copy memory management. PGD: page global directory; PUD: page upper directory; PMD: page middle directory; PTE: page table entry**

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 model's 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 consumes 7717.77 ms, and model initialization requires 10 549.78 ms, together accounting for 81% of the total loading time for the large model. A comparative analysis of the process, before and after optimization, 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 uses the SHA-256 hash function for these verification operations. However, we identify that performing hash verification on the entire model content at each deployment introduces a significant computational overhead. Specifically, for the large whisper model, this process requires approximately 7.71 s, 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 RABs and MHA, undergoes an initialization phase that encompasses essential layers like linear and normalization layers. We denote the initialization times for the encoder and decoder as $t_{\text{Encoder}}$ and $t_{\text{Decoder}}$, respectively, calculated using the following equations:

$$t_{\text{Encoder}}=2t_{\text{Conv1D}}+n_{\text{RAB}}(6t_{\text{Linear}}+3t_{\text{LayerNorm}})+t_{\text{LayerNorm}}, \quad (3)$$

$$t_{\text{Decoder}}=t_{\text{Embedding}}+n_{\text{RAB}}(10t_{\text{Linear}}+3t_{\text{LayerNorm}})+t_{\text{LayerNorm}}, \quad (4)$$

where $t_{\text{Conv1D}}$ denotes the initialization time of a one-dimensional convolutional layer, $t_{\text{Embedding}}$ is the time of the embedding part, $t_{\text{Linear}}$ represents the initialization time of a linear layer, $t_{\text{LayerNorm}}$ is the initialization time of a layer normalization layer, and $n_{\text{RAB}}$ represents the number of RABs. The overall initialization time for the whisper model is the sum of $t_{\text{Encoder}}$ and $t_{\text{Decoder}}$.

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 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. 8a presents a schematic of the traditional model loading sequence in PyTorch, outlining the essential steps for model deployment. In contrast, Fig. 8b 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, which is characterized by three significant improvements. First, we

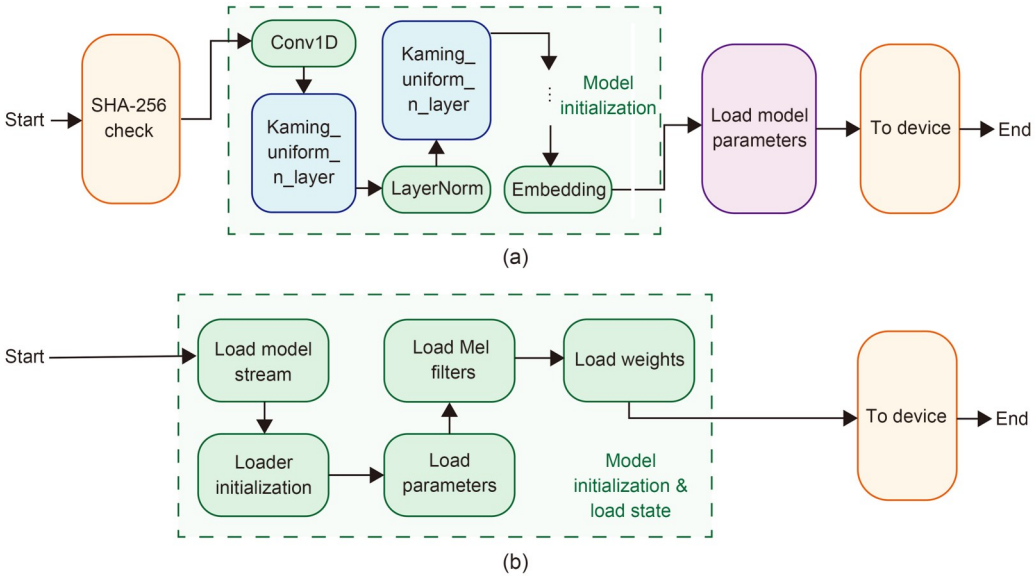


**Fig. 8 Model loading process: (a) PyTorch model loading process; (b) fast loading process**

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. 8b, strategically integrates model initialization with parameter loading into a singular and 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.

## 5 Experiments

### 5.1 Experimental setup

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

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

### 5.2 Launch overhead

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

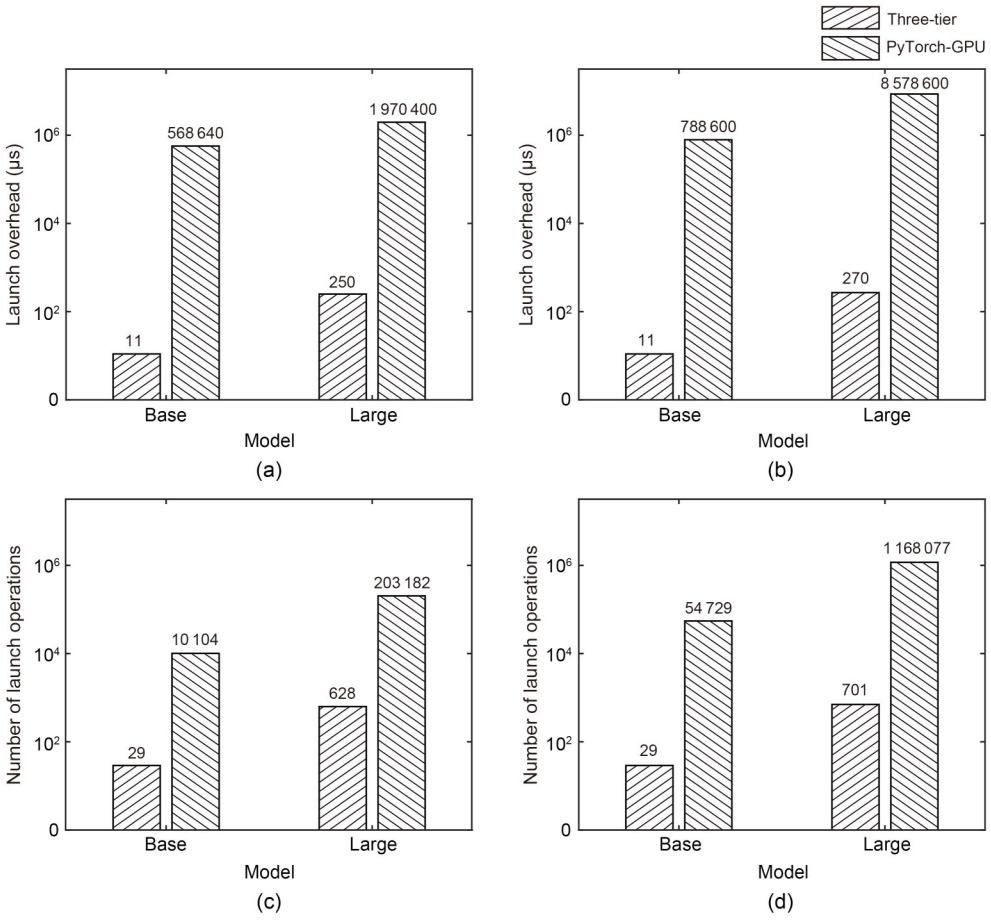


**Fig. 9 Launch overhead and numbers of launch operations for JFK and HP0 with base and large whisper models: (a) launch overhead with JFK; (b) launch overhead with HP0; (c) numbers of launch operations with JFK; (d) numbers of launch operations with HP0**

Figs. 9a and 9b illustrate the significant reduction in launch overhead with our approach. For the base model, the launch overhead for JFK and HP0 audio is dramatically decreased from 568 640 μs and 788 600 μs to approximately 11 μs. Similarly, for the large whisper model, the launch overhead for JFK and HP0 audio is 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 subsection 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 whisper model, the memcpy cost soars to 749 ms, making it the most time-consuming element within the entire inference process.

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 ms—a significant reduction when compared to overhead typically observed in GPU-based systems. Furthermore, the deployment of this technique has notably minimized 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 detailed 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, and under the most favorable conditions, our overhead is merely about 5% of the overhead associated with PyTorch.

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 using JFK for inference, the overhead due to graph construction is less than one-tenth of the API invocation overhead. On the contrary,

**Table 4 Memcpy time breakdown comparison**

| Model | Time for PyTorch-GPU | Time for SWAI | | | |
|---|---|---|---|---|---|
| | | Before | | After | |
| | | Host-to-device | MPE-to-SPE | Host-to-device | MPE-to-SPE |
| Base (JFK) | 30 ms | 45.92 ms | 1.29 ms | 44.63 ms | 0.3 μs |
| Base (HP0) | 107 ms | 75.76 ms | 27.5 ms | 48.71 ms | 0.4 μs |
| Large (JFK) | 483 ms | 49.03 ms | 1.22 ms | 47.81 ms | 0.2 μs |
| Large (HP0) | 749 ms | 79.93 ms | 29.2 ms | 50.73 ms | 0.5 μs |

**Table 5 Model loading time breakdown comparison**

| Model | Method | Time (ms) | | | | |
|---|---|---|---|---|---|---|
| | | Model validation | Parameter loading | Model initialization | Model-to-device | Load model total |
| 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 | 10 995.37 | 1674.16 | 22 128.31 |
| | Fast loading | 0 | 303.60 | | 738.12 | 1041.72 |

In fast loading method, the parameter loading and model initialization are combined into one process

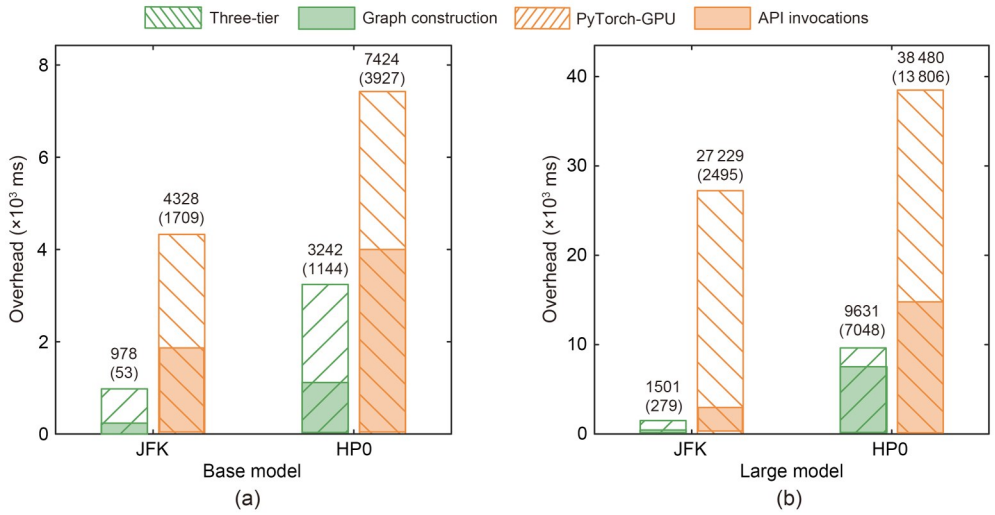


**Fig. 10 Overall overhead comparison: (a) base model; (b) large model. References to color refer to the online version of this figure**

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.

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 the MPE on the SWAI accelerator. Our contributions to the domain of transformer model optimization include addressing the challenge of minimizing inference overhead, 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 NLP, 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 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, contributed to the experimental design, and wrote significant portions of the paper. Chunzhi WU and Lufei ZHANG contributed to the statistical analysis of the data and the creation of the figures and tables. Yizhuo WANG, Yaguang ZHANG, Wenyuan SHEN, and Hao FAN contributed to the experimental design and data collection and analysis. Hankang FANG and Yi QIN contributed to the literature review and background research. Xin LIU helped organize the paper. Chunzhi WU, Yizhuo WANG, Yaguang ZHANG, and Lufei ZHANG revised and finalized the paper. All the authors contributed to the revision and read and approved the submitted version.

## Conflict of interest

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

## Data availability

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. IEEE High Performance Extreme Computing Conf, 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. http://developer.amd.com/afds/assets/presentations/1004_final.pdf [Accessed on Aug. 25, 2024].

Chen GY, Shen XP, 2015. Free launch: optimizing GPU dynamic kernel launches through thread reuse. 48th Annual IEEE/ACM Int Symp on Microarchitecture, 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. IEEE Int Conf on Cluster Computing, 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 Circ 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. 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 [Accessed on Aug. 25, 2024].

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. IEEE Int Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, and Social Computing & Networking, 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. IEEE Hot Chips 32 Symp, 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. https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf [Accessed on Aug. 25, 2024].

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. 58th ACM/IEEE Design Automation Conf, 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. Int Conf on Inventive Communication and Computational Technologies, 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. IEEE Hot Chips 31 Symp, 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. Int Conf on Parallel Processing, p.5-8.