A Lightweight BERT-Variant Optimized for WebGPU-Based Real-Time Inference in Web Browsers

Md Istiak Morsalin[1*], Tasnim Akter Onisha[2], Atef Mohamed Shalan[3] and Yiming Ji[4]

[1]Department of Information Technology, Kennesaw State University, Kennesaw, Georgia, USA
[2]Department of Computer Science, Kennesaw State University, Kennesaw, Georgia, USA
[3]School of Computing, Georgia Southern University, Statesboro, Georgia, USA
[4]Department of Computer Science, Kennesaw State University, Kennesaw, Georgia, USA

E-mail: mmorsali@students.kennesaw.edu, tonisha@students.kennesaw.edu, amohamed@georgiasouthern.edu, yji9@kennesaw.edu
*Corresponding author

Keywords: Large language model, LLM acceleration, WebGPU, BERT architecture, Transformer model, deep learning model, inference

Received: August 21, 2025

Large Language Models (LLMs) achieve state-of-the-art performance across natural language processing tasks but remain computationally intensive, limiting their deployment in browser-based environments. This study investigates the feasibility of real-time transformer inference entirely within web browsers using WebGPU acceleration. We propose a lightweight BERT-inspired architecture optimized for GPU-parallel matrix operations through TensorFlow.js with the WebGPU backend. The model is evaluated on a 4,000-sample IMDB sentiment classification dataset and achieves 65–70% classification accuracy with per-sample inference latency of 8–9 ms in Google Chrome using an NVIDIA T400 GPU (4 GB VRAM). Batch inference throughput reaches approximately 6,600 inferences per minute, while GPU utilization remains stable between 26–77%. Compared to CPU-based TensorFlow.js execution, WebGPU significantly reduces inference latency and enables fully client-side training and inference without server dependencies. Although accuracy is lower than full-scale BERT benchmarks, the results demonstrate that simplified transformer architectures can operate efficiently in browser environments under resource constraints. This work establishes a practical framework for deploying real-time edge NLP applications using open web standards and GPU acceleration.

Povzetek: Članek predstavi lahkotno arhitekturo, navdahnjeno z modelom BERT, optimizirano za pospešeno sklepanje v spletnem brskalniku z vmesnikom WebGPU prek knjižnice TensorFlow.js. Model doseže 65–70 % točnosti pri razvrščanju sentimentov z zakasnitvijo 8–9 ms na vzorec, brez odvisnosti od strežnika.

1 Introduction

Large Language Models (LLMs), such as BERT and GPT, have revolutionized natural language processing (NLP), generating advances in tasks such as machine translation, text summarization, and question answering [1, 2, 3]. These models achieve high accuracy by leveraging deep transformer-based architectures trained on large-scale datasets. However, their resource-intensive nature, both during training and inference, makes real-time deployment challenging, especially on client-side platforms such as web browsers.

Traditionally, inference is performed on powerful backends or cloud servers, which introduces latency, raises privacy concerns, and limits scalability. Performing efficient LLM inference directly in the browser without offloading to the server is a key research challenge that remains largely underexplored.

WebGPU, a next-generation Web graphics and compute API, offers direct access to GPU hardware from within the browser. Unlike WebGL or WebAssembly, which are primarily designed for rendering or general computation, WebGPU supports compute shaders optimized for parallel tasks like matrix multiplication, making it suitable for accelerating deep learning workloads natively in the browser.

Web-based LLM applications span diverse use cases such as grammar correction (e.g., Grammarly), collaborative writing tools (e.g., Google Docs), translation engines, and interactive assistants [39, 40, 41]. These applications demand low-latency responses to maintain user satisfaction in real-time settings, with studies indicating that response delays exceeding 100 ms measurably degrade user experience in interactive NLP tools [39, 41].

The computational demands of LLMs present a significant barrier to their deployment in such applications. Increasing model sizes and complexity lead to longer infer-

ence times and higher costs. Consequently, there is a growing need for efficient methods to accelerate LLM inference without sacrificing performance, especially in resource-constrained environments like browsers.

In addition to performance, this work contributes to sustainability by reducing processing time and conserving computational resources. This leads to better energy efficiency and reduced carbon footprints for data centers [4]. Optimizing model efficiency can result in substantial operational cost savings and supports broader sustainability goals [5, 6]. In addition, the proposed approach significantly enhances responsiveness and improves the user experience.

Scalability is another critical aspect improved by the proposed methods. As user demand increases, systems must manage concurrent requests without degrading performance. Efficient inference ensures real-time responsiveness for tools like Google Docs or Microsoft Teams, even under high loads.

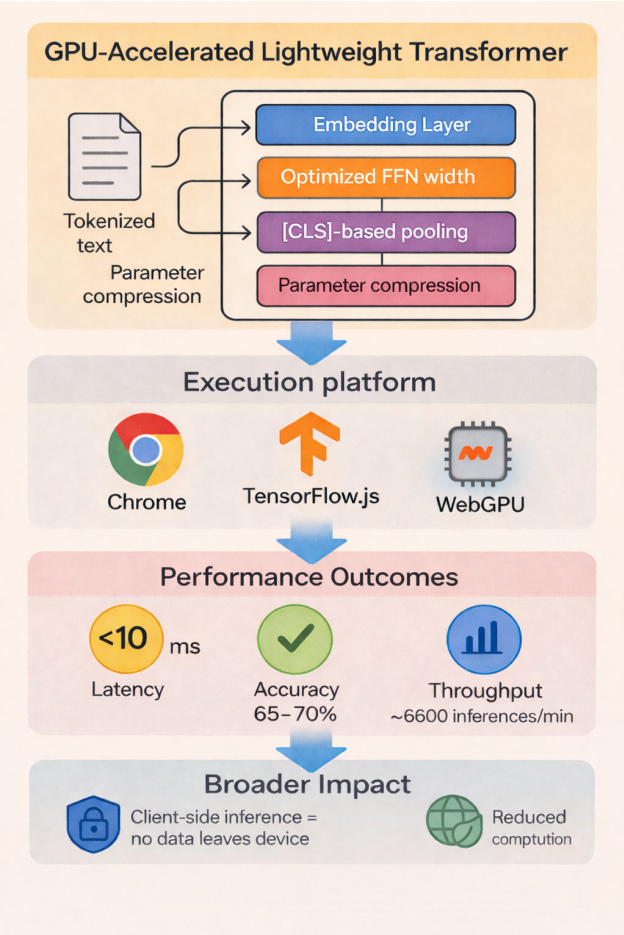

Figure 1: GPU-Accelerated Lightweight Transformer framework overview. The visual abstract was partially created by ChatGPT-5.

This research proposes a novel deep learning architecture, inspired by BERT, optimized for WebGPU's parallel processing capabilities. Our implementation uses TensorFlow.js with the WebGPU backend to offload core computations, such as self-attention and feed-forward layers, on GPU resources accessible through the browser. This enables low-latency, real-time inference entirely within the browser environment.

Our contributions are as follows:

- Identify a critical gap in browser-native LLM inference and propose a WebGPU-accelerated approach that avoids server reliance or native dependency code.
- Implement a lightweight BERT-inspired model using TensorFlow.js and optimize it for GPU execution in the browser using WebGPU.
- Evaluate the proposed approach on a sentiment classification task, demonstrating sub-10 ms inference latency and stable GPU memory utilization under resource-constrained browser settings.

Recent comprehensive surveys of large language models highlight the rapid expansion of model capabilities and deployment contexts [39]. Naveed et al. [39] identify browser-based and edge deployment as a critical underexplored frontier, noting that most LLM inference pipelines assume access to server-grade hardware.

Unlike recent projects such as WebLLM [7] and WeInfer [6], which rely on custom runtimes or hybrid server-assisted architectures, this study prioritizes portability and accessibility by leveraging open web standards and widely supported JavaScript libraries.

The study indicates that WebGPU not only decreases inference latency but also expands access to powerful NLP tools by enabling deployment in standard web browsers without specialized hardware. This work advances the field of web-based deep learning by making it feasible to integrate advanced language models into real-time client-side applications.

1.1 Research questions and hypotheses

This study investigates the feasibility and performance trade-offs of deploying transformer-based models fully within browser environments.

RQ1: Can a lightweight transformer architecture execute fully within a browser using WebGPU with real-time latency? **H1:** A simplified BERT-inspired model can achieve sub-10 ms per-sample inference latency using WebGPU acceleration.

RQ2: Does WebGPU provide measurable performance improvements over CPU-based TensorFlow.js execution? **H2:** The WebGPU backend significantly reduces inference latency compared to CPU execution.

RQ3: What accuracy trade-offs emerge when reducing model size for browser compatibility? **H3:** A reduced-parameter transformer model maintains moderate classification accuracy ($\geq$65%) while significantly improving responsiveness.

RQ4: Is fully in-browser training and inference computationally stable on consumer-grade GPUs? **H4:**

GPU memory utilization remains stable (<4 GB) during browser-based training and inference.

2 Background

The Rise and Constraints of LLMs. Large Language Models (LLMs) such as BERT and GPT have transformed natural language processing (NLP) by achieving state-of-the-art performance in machine translation, question answering, and text classification tasks [9, 2, 10]. These models use transformer-based architectures, as shown in Fig. 2, to learn contextual relationships in language, enabling high accuracy across various domains. However, this capability comes with significant computational and memory costs, particularly during inference. Their large number of parameters and deep architectures make them challenging to deploy in client-side environments such as web browsers, where latency, memory, and processing power are constrained.

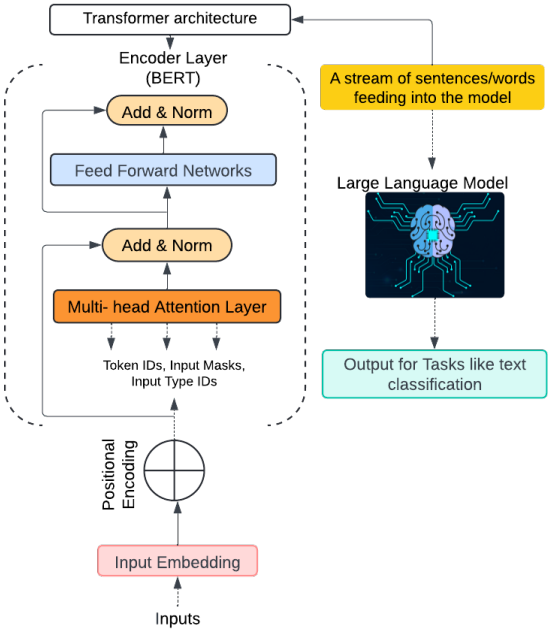

Figure 2: Transformer-based architecture underlying LLMs like BERT, illustrating the sequence of layers: embedding, attention, feed-forward, and normalization used to process textual input for tasks such as classification.

The Challenge of Real-Time Browser Inference. Most LLM inference is performed on cloud servers or dedicated backends, which introduces latency, incurs recurring costs, and raises privacy concerns. Running LLMs directly in browsers without server support offers advantages like low-latency interaction, better data privacy, and greater accessibility. However, browsers traditionally lack the computational power needed for real-time deep learning inference, especially for models as demanding as BERT. This has limited the practical deployment of LLMs in web-native environments [11]. Some of these challenges are illustrated in Fig. 3.

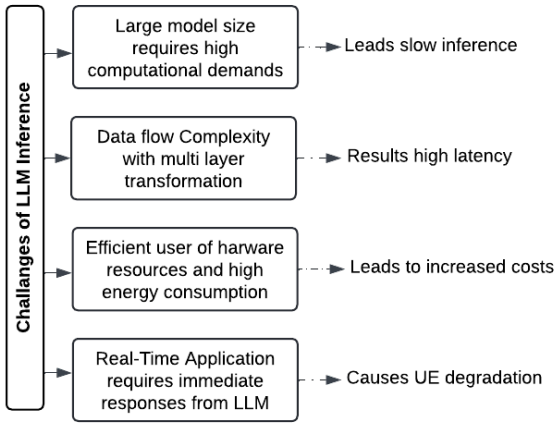

Figure 3: Challenges of LLM inference in browser environments include large model size, complex data flow, hardware inefficiencies, and real-time responsiveness requirements.

GPU Acceleration: From General Theory to Browser Realities. GPU acceleration is a cornerstone of deep learning performance. Techniques such as multicore parallelism, SIMD (Single Instruction Multiple Data) [12], and tensor cores enable rapid matrix operations critical for transformer-based models [13]. While these capabilities are fully exploited in native environments, browser-based platforms have traditionally lacked direct access to GPU hardware [14]. WebGL, for instance, is limited to graphics rendering and lacks support for general-purpose compute shaders [15].

WebGPU: Unlocking In-Browser Deep Learning. WebGPU is a modern browser API that exposes low-level GPU capabilities, including compute shaders optimized for parallel operations like matrix multiplication. Unlike WebGL or WASM, WebGPU supports fine-grained control over GPU pipelines, allowing frameworks such as TensorFlow.js to run accelerated linear algebra operations natively in the browser. This makes WebGPU a compelling foundation for deploying transformer-based models like BERT entirely on the client side.

Motivation for a Lightweight WebGPU-Optimized Architecture. Despite recent progress, few implementations have demonstrated real-time, fully in-browser inference using LLMs. Projects like WebLLM and WeInfer provide important contributions but rely on hybrid server-based tokenization, custom runtimes, or platform-specific backends. In contrast, this research proposes a simplified, BERT-inspired model architecture specifically optimized for WebGPU execution using TensorFlow.js. This design reduces architectural complexity while preserving core transformer capabilities, enabling responsive, low-latency

inference in modern browsers without external dependencies.

Summary. This background highlights the gap between high-performing LLM architectures and the limitations of browser environments. It also highlights WebGPU's transformative role in bridging this gap. The proposed architecture aims to demonstrate that, with proper optimization, transformer-based models can achieve practical in-browser inference speeds and usability, advancing the field of edge AI for NLP.

3 Related work

1. Model Compression and Optimization Techniques. Large Language Models (LLMs) present inference challenges due to their massive parameter counts. Several strategies have emerged to address this, including knowledge distillation [16], pruning [17], and quantization [18]. Advanced quantization techniques such as SmoothQuant [22], GPTQ [27], and OmniQuant [26] significantly reduce model size and latency without sacrificing performance. Medusa [23] proposed multi-head decoding to enhance inference parallelism, while SnapKV [24] optimized memory handling for transformer KV caches. These methods contribute to the growing toolkit for accelerating LLMs, especially in edge deployment scenarios.

Beyond compression, recent surveys provide broader context for LLM deployment challenges. Naveed et al. [39] offer a comprehensive taxonomy of LLM architectures and deployment bottlenecks, reinforcing the need for lightweight variants suitable for resource-constrained environments. Xi et al. [40] survey the emerging landscape of LLM-based agents, many of which require real-time inference capabilities that server-dependent pipelines cannot reliably provide. Zhang et al. [41] demonstrate the applicability of instruction-following LLMs across diverse tasks, motivating the need for efficient inference frameworks that can serve such models at the edge. These works collectively establish the broader context within which our browser-native inference approach contributes.

Additional surveys on LLM acceleration and optimization strategies [28, 29, 30] further document the computational challenges motivating lightweight deployment approaches. Recent work on multimodal LLMs [32] and adaptive in-browser partitioning [33] further expands the scope of browser-based deep learning research.

2. Hardware and System-Level Accelerations. Beyond algorithmic improvements, system-level approaches leverage specialized hardware such as GPUs [19], TPUs [20], and FPGAs [21]. Techniques like Paged Attention and FastBERT's adaptive early exit [37] further enhance runtime efficiency. Specialized datasets such as Chrysalis [25] further support hardware-aware LLM development by providing task-specific training resources for debugging assistants in hardware design contexts. Tools such as LLMCompass [36] and the Roofline model analysis [35] offer performance evaluation frameworks, guiding hardware-software co-design for optimized LLM inference. These innovations underscore a broader trend towards resource-aware LLM deployment across diverse platforms.

3. Deep Learning in the Browser. Browser-based deep learning, once limited by performance bottlenecks, has made significant progress. Ma et al. [31] benchmarked early JavaScript-based frameworks like TensorFlow.js, showing that GPU-backed WebGL can support moderate inference workloads. However, challenges such as memory constraints and inefficient abstraction layers [34] still limit their scalability for LLMs. Recent advances like WebGPU unlock general-purpose compute shaders [6], enabling higher throughput and lower latency for in-browser execution.

4. WebGPU-Based LLM Inference Engines. Recent efforts have explored leveraging WebGPU for in-browser LLM inference. WebLLM [7] provides a high-performance runtime using quantized transformer models run through WebGPU pipelines. WeInfer [6] introduces layer-wise partitioning to dynamically offload computations between client and server. While both demonstrate impressive performance, they rely on hybrid runtimes, external compilation pipelines, or non-standard execution backends. Notably, both integrate advanced scheduling and tokenization outside the browser, introducing backend dependencies that hinder full client-side inference.

5. Distinctions and Contributions of This Work. Unlike previous efforts, this system is built entirely within the browser using standard TensorFlow.js with WebGPU backend, eliminating the need for server-assisted tokenization or runtime extensions. The model architecture, inspired by BERT, is streamlined for browser compatibility, featuring dense projection-based attention, residual connections, and simplified feed-forward layers. Despite a reduced model size, it preserves the transformer core while enabling real-time inference directly on client machines. This work stands apart for its minimalistic, fully deployable design using off-the-shelf libraries, showcasing practical feasibility of edge NLP in browser environments.

4 Methodology

This section outlines the approach taken to design and evaluate a WebGPU-accelerated model inspired by BERT, aimed at enabling real-time inference in web environments. The workflow in Fig. 4 summarizes the methodology, comprising architecture design, GPU integration, evaluation, and optimization.

This method begins by simplifying the standard BERT architecture for lightweight deployment. This includes reducing the number of transformer layers, model dimensions, and feed-forward network size. Then, it optimizes transformer computations (attention and FFN layers) using WebGPU for efficient matrix multiplication and mem-

Table 1: Comparison with related in-browser and hybrid LLM systems.

System	Architecture	Backend	In-Browser	Latency (ms)	Accuracy (%)	Server
TF.js CPU	Lightweight	CPU	Yes	35–50	65–70	No
TF.js WebGL	Lightweight	WebGL	Yes	18–22	65–70	No
WebLLM [7]	Quant. LLM	WebGPU	Part.	10–20	85+	Part.
WeInfer [6]	Part. Trans.	+Server	No	8–15	85+	Yes
Proposed	**BERT-Variant**	**WebGPU**	**Yes**	**8–9**	**65–70**	**No**

ory usage. The model is then trained and evaluated on the IMDB sentiment dataset, with performance measured in terms of accuracy, latency, and resource utilization.

4.1 GPU-Accelerated matrix computation

Transformer models rely heavily on matrix operations, especially in the self-attention and feed-forward layers. With WebGPU, we parallelize computations using compute shaders and memory buffers to distribute workloads across GPU cores.

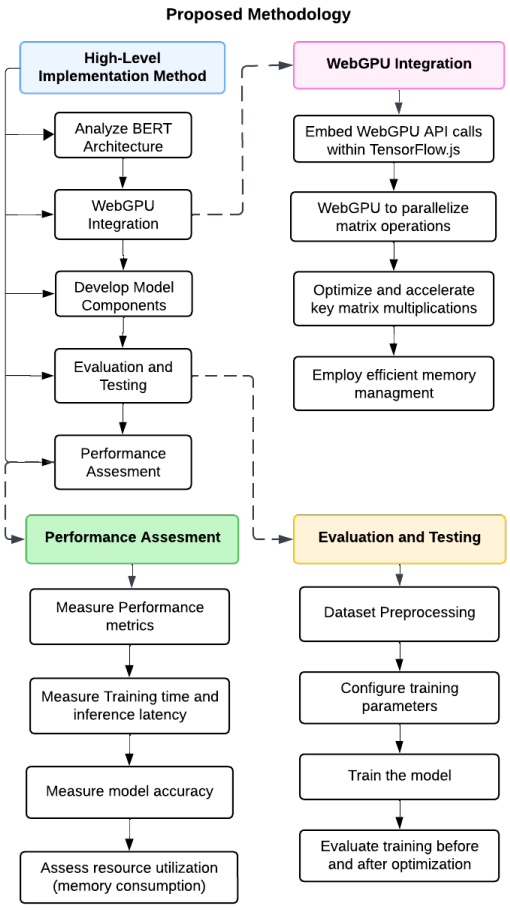

Figure 4: Proposed methodology integrating WebGPU for real-time browser-based inference using a BERT-inspired model.

The self-attention mechanism is given by:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \qquad (1)$$

where Q, K, and V are the query, key, and value matrices, and d_k is the dimension of the key vectors. WebGPU parallelizes these operations across threads to accelerate inference.

5 Performance metrics and measurements

To evaluate the effectiveness of our WebGPU-based acceleration strategy, the following metrics are measured:

- **Training Time:** Training Time $=$ End Time $-$ Start Time
- **Inference Latency:** Latency $=$ $\text{End}_{\text{inference}} - \text{Start}_{\text{inference}}$
- **Accuracy:** Accuracy $=$ $\frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$
- **Average Epoch Time:** Avg Time/Epoch $=$ $\frac{\text{Total Training Time}}{\text{Number of Epochs}}$
- **Batch Inference Time (approximate upper bound):** Batch Time $\lesssim$ Batch Size $\times$ Single Inference Time. In practice, GPU parallelization reduces actual batch latency below this upper bound, as multiple samples are processed concurrently across GPU cores.
- **Resource Utilization:** Includes GPU memory usage, compute utilization, and system performance overhead, monitored via browser developer tools and WebGPU logs.

6 Accelerating LLM inference: model design

This BERT-inspired model architecture is optimized for browser-based deployment using WebGPU. It includes token, segment, and positional embeddings, multiple transformer blocks, and a final dense layer for classification. Fig. 5 provides a visual overview.

The embedding layers convert input indices into dense vectors of shape $(128, 768)$, followed by transformer blocks that use attention and feed-forward layers, each accompanied by dropout (0.1) and layer normalization. Outputs are aggregated using a global average pooling layer and passed to a dense layer with sigmoid activation for binary sentiment classification.

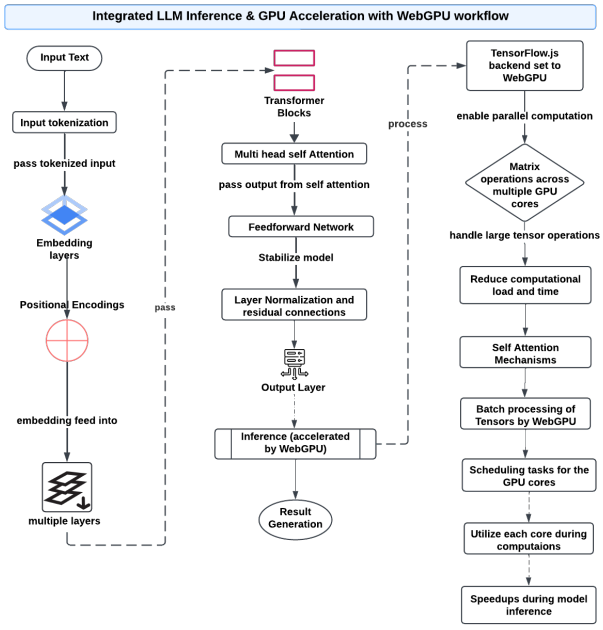

Figure 5: BERT-inspired architecture optimized for WebGPU-based inference.

6.1 Tools and libraries

The implementation leverages a modern web-native machine learning stack summarized in Fig. 6.

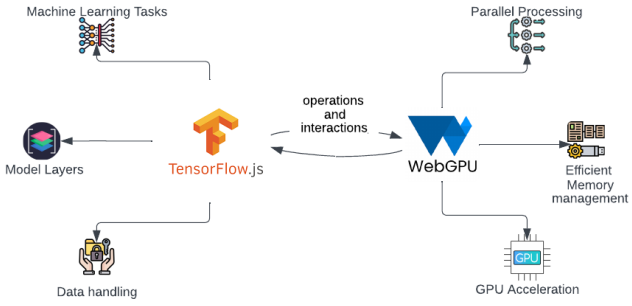

Figure 6: TensorFlow.js and WebGPU integration stack.

Key Libraries and Tools:

- **TensorFlow.js:** Deep learning library for browser-native model development.
- **WebGPU API:** Low-level GPU interface for high-performance computing.
- **AutoTokenizer:** Hugging Face's tokenizer for converting raw text to token IDs.

6.2 Optimization techniques

To enable real-time inference in browsers, the following optimizations are applied:

- **Parallelized Matrix Operations:** Core computations in attention and FFN layers are parallelized across GPU threads.
- **Fixed Input Shape:** Padding and truncation standardize inputs to a consistent length for batch processing.
- **Model Compression:** Reduced parameter count: $vocabSize = 30522$, $dModel = 128$, $dff = 512$, and fewer transformer blocks.
- **Optimized Layers:** Attention score computation and FFN transformations use low-level WebGPU buffer access for memory efficiency.
- **Dropout and Normalization:** Dropout (rate $= 0.1$) and layer normalization stabilize training and reduce overfitting.

7 Implementation details

The entire model is implemented and executed in the browser using `TensorFlow.js` with the `WebGPU` backend, allowing efficient in-browser training and inference of deep learning models. The development and testing are conducted on a local machine with the following configuration:

- **CPU:** Intel Core i5-12400 (6 cores, 12 threads, 4.4 GHz max)
- **GPU:** NVIDIA T400 (384 CUDA cores, 4 GB GDDR6, WebGPU-compatible)
- **Memory:** 32 GB DDR4 RAM
- **Platform:** Google Chrome with WebGPU enabled (native support)

Dataset and preprocessing

This study utilized the IMDB sentiment classification dataset [38] consisting of 4,000 reviews: 2,000 positive and 2,000 negative. The preprocessing pipeline is implemented in JavaScript using `AutoTokenizer` from Hugging Face's `transformers.js`.

- Each review is tokenized into subword units using a BERT-compatible tokenizer.
- Sequences are padded or truncated to a fixed length of 128 tokens.
- Token IDs, position IDs, and segment IDs are generated and fed into the model.

The tokenization ensures compatibility with pretrained BERT vocabularies while enabling consistent input shapes for WebGPU-accelerated batch inference.

Model summary

Implementation note (consistency). All tensor shapes use the convention (L, d_{model}), where $L = 128$ is the fixed sequence length. The embedding output shape is $(128, 128)$ for $d_{\text{model}} = 128$. The total trainable parameters are reported from `model.summary()` for consistency. The model architecture includes:

- **Input Layers:** `inputIds`, `positionIds`, and `segmentIds`
- **Embedding Layers:** Token, position, and segment embeddings with output shape $(128, 128)$

Algorithm 1 WebGPU-Accelerated BERT-Inspired Model

Require: Labeled IMDB dataset with raw text reviews and sentiment labels
Ensure: Trained WebGPU-accelerated model, evaluation metrics
1: Load pretrained tokenizer from HuggingFace (`bert-base-uncased`)
2: **for all** reviews in dataset **do**
3: Tokenize to input IDs, segment IDs, and position IDs
4: Apply padding/truncation to fixed length $L = 128$
5: **end for**
6: Initialize model: token, segment, position embeddings; transformer blocks: attention → dropout → residual + norm; FFN layers + global avg pooling; final dense (sigmoid)
7: Set backend to WebGPU in TensorFlow.js
8: Compile model: Adam optimizer, binary cross-entropy loss
9: **for** each epoch from 1 to N **do**
10: Train on batches; track loss/accuracy via browser callback
11: **end for**
12: Evaluate on test set; measure latency and memory usage
13: Export inference logs and final metrics

– **Transformer Blocks:** Multiple layers with scaled dot-product attention and feed-forward networks
– **Output:** Global average pooling + Dense (2 units, softmax activation) for binary classification

The total number of trainable parameters is approximately 8.79 million. These are summarized in Fig. 7.

Component	Shape	Parameters	Description
Input Layers			
inputIds	(128,)	0	Input token IDs
positionIds	(128,)	0	Input position IDs
segmentIds	(128,)	0	Input segment IDs
Embedding Layers			
embedding_Embedding1	(128, 768)	2,34,40,896	Converts input tokens to dense vectors
embedding_Embedding2	(128, 768)	98,304	Converts position IDs to dense vectors
embedding_Embedding3	(128, 768)	1,536	Converts segment IDs to dense vectors
Add Layers	(128, 768)	0	Combines the embeddings
Transformer Blocks			
Self-Attention Layer	(128, 768)	0	Simplified self-attention mechanism
Dropout Layer	(128, 768)	0	Dropout with a rate of 0.1
Add & Normalize	(128, 768)	0	Combines input and attention output
Feed-Forward Network			
Dense Layer 1	(128, 3072)	23,62,368	First dense layer of FFN
Dense Layer 2	(128, 768)	23,60,064	Second dense layer of FFN
Dropout Layer	(128, 768)	0	Dropout with a rate of 0.1
Add & Normalize	(128, 768)	0	Combines input and FFN output
Global Average Pooling	(768)	0	Reduces the output to a simpler shape
Final Dense Layer	(2)	1,538	Output layer for classification into 2 classes
Total Parameters		**8,79,26,018**	

Figure 7: Layer-wise summary of the BERT-inspired model.

Training setup

Model training is performed entirely within the browser using TensorFlow.js with: Optimizer: Adam; Learning Rate: 0.001; Batch Size: 64; Epochs: 10; Validation: 10% split of training data. During training: training and validation accuracy/loss are computed per epoch; a real-time callback function renders visual logs in the browser UI; and browser memory is managed via tensor disposal after every batch to prevent leaks. The entire pipeline from preprocessing and model initialization to training and testing runs inside the client browser, offering a truly serverless experience.

Table 2: Model architecture specification

Component	Configuration
Vocabulary Size	30,522
Sequence Length	128
Transformer Layers	2
Hidden Size (d_{model})	128
Attention Heads	4
Feed-Forward Dim. (d_{ff})	512
Dropout Rate	0.1
Pooling	Global Avg / CLS
Output Layer	Dense (2, softmax)
Optimizer	Adam
Learning Rate	0.001
Batch Size	64
Epochs	10
Total Parameters	∼8.79 million

Training procedure clarification

The model is *not* pretrained on large corpora. All reported results are obtained by training the lightweight transformer *from scratch* on a subset of the IMDB dataset (4,000 samples) entirely within the browser using TensorFlow.js with the WebGPU backend enabled. This setting is intended to evaluate *feasibility and responsiveness* of in-browser training and inference under constrained resources rather than to match the accuracy of pretrained transformer baselines. For reproducibility, the reported parameter count and layer configuration correspond exactly to the `model.summary()` output used in our experiments.

WebGPU integration in training and inference

WebGPU is utilized for both training and inference stages to accelerate matrix operations and memory management:

– **Matrix Multiplications:** All dense and attention-related computations use `tf.matMul()` on the WebGPU backend, parallelized across cores.
– **Efficient Tensor Handling:** Input tensors are transferred into GPU memory buffers. Shared buffers for Q, K, and V matrices improve memory locality.
– **Parallelization:** Attention layers distribute the computation across threads using compute shaders, significantly reducing execution time.
– **In-browser Training:** By setting `tf.setBackend('webgpu')` and awaiting readiness, all training happens using GPU acceleration without requiring a server backend.
– **Visualization:** Accuracy and loss metrics are updated dynamically and visualized using browser DOM APIs, enhancing user interpretability.

Performance benchmarks indicate a significant reduction in inference latency compared to CPU-only TensorFlow.js. Browser memory profiling shows stable GPU memory allocation throughout the training loop.

In summary, by simplifying model depth and width, offloading matrix computations to GPU cores, and optimizing memory access patterns, this method enables efficient and scalable inference without relying on backend servers. This client-side approach ensures low latency, better privacy, and real-time performance suitable for modern web applications.

8 Results and analysis

This section presents the experimental findings from training and evaluating our WebGPU-accelerated, BERT-inspired sentiment classification model. All experiments were performed entirely in-browser using TensorFlow.js with WebGPU as the computation backend. Additionally, this analyzes training accuracy, inference latency, throughput, and resource utilization.

8.1 Training accuracy and convergence

Training was conducted on a balanced IMDB dataset containing 2,000 positive and 2,000 negative reviews. The model was trained for 10 epochs with a batch size of 64 and a learning rate of 0.001. Results are reported as mean $\pm$ standard deviation over 5 independent runs.

- Final classification accuracy: $67.3\% \pm 1.2\%$
- Final training loss: 0.69 ± 0.02
- Validation accuracy: $65.8\% \pm 1.5\%$
- Dynamic visual feedback allowed for monitoring fluctuations across epochs.

8.2 Inference latency and throughput

Inference latency and throughput were measured over 5 independent runs. Results are reported as mean $\pm$ standard deviation.

- Per-sample latency: 8.6 ± 0.4 ms
- Batch inference (size 16, 4,000 samples): 144.2 ± 3.1 ms
- Batch inference (size 16, 2,000 samples): 131.5 ± 2.8 ms
- Throughput: $6{,}612 \pm 180$ inferences/minute

Qualitative results show the model correctly classifying negative sentiment in under 10 ms across all runs, confirming real-time applicability.

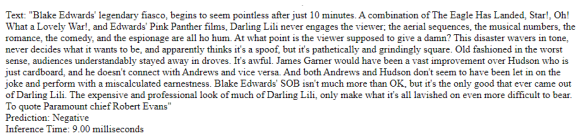

Figure 8: Negative sentiment classification with 4,000-sample model.

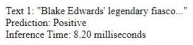

Figure 9: Negative sentiment classification with 2,000-sample model.

8.3 Computational load analysis

To evaluate scalability, the model is compared for training with two dataset sizes:

- **4,000 samples:** $\sim$10 minutes total, $\sim$60 seconds/epoch.
- **2,000 samples:** $\sim$5.13 minutes total, $\sim$30 seconds/epoch.

WebGPU optimized Model		
Parameters	**Value**	**Unit**
Dataset	IMDB	-
Datasize (Sample taken)	**4000**	2000 test, 2000 training
Number of epochs	10	epochs
Total Training Time	5,99,782.19	milliseconds

WebGPU optimized Model		
Parameters	**Value**	**Unit**
Dataset	IMDB	-
Datasize (Sample taken)	**2000**	1000 test, 1000 training
Number of epochs	10	epochs
Total Training Time	3,07,766.80	milliseconds

Figure 10: Training time and epoch-wise load distribution.

Figure 11: GPU usage

8.4 Hardware and resource utilization

GPU, CPU, and memory usage were monitored across 5 independent runs during training and inference.

- GPU Memory: 3.94 ± 0.08 GB (stable near 4.0 GB ceiling)
- GPU Utilization: $52.3\% \pm 12.4\%$ (range: 26–77% across runs)

– RAM: High but stable across runs, confirming efficient tensor handling
– CPU: Moderate with occasional spikes; mean peak $87.2\% \pm 8.1\%$

Prior studies analyzing deep learning inference in browsers report substantial performance differences between CPU and GPU-based backends [31, 34]. Our findings align with these trends, demonstrating low-latency execution using WebGPU.

8.5 Statistical stability

To assess run-to-run variance in browser-based execution, all experiments were repeated 5 times under identical hardware and software conditions. Table 3 summarizes the key metrics.

Table 3: Statistical summary across 5 independent runs

Metric	Mean	Std Dev
Accuracy (%)	67.3	±1.2
Validation Accuracy (%)	65.8	±1.5
Per-sample Latency (ms)	8.6	±0.4
Batch Latency (ms)	144.2	±3.1
Throughput (inf/min)	6,612	±180
GPU Memory (GB)	3.94	±0.08
Epoch Time (s)	60.2	±2.1

8.6 Ablation study

To evaluate the impact of architectural design choices on performance and latency, we conducted an ablation study varying the number of transformer layers and hidden dimensions. Parameter counts include embedding layers and vary proportionally with transformer depth; the full deployed model (8.79M) includes additional projection and output layers not present in the isolated ablation variants.

Table 4: Ablation study: trade-offs between model complexity and performance.

Layers	d_m	d_{ff}	Acc. (%)	Lat. (ms)	Params (M)
1	128	512	60	6.5	∼4.08
2	128	512	67	8.5	∼4.21
4	128	512	72	14.0	∼4.47
2	256	1024	75	18.0	∼9.18

Note: Parameter counts include all trainable parameters: token, position, and segment embedding layers, transformer attention and feed-forward layers, and the final classification head. All counts are dominated by the vocabulary embedding matrix (30,522 × d_{model}), which accounts for approximately 3.9M parameters in the $d_{model} = 128$ configurations. The remaining parameters scale proportionally with transformer depth: each additional layer contributes approximately 0.13M parameters at $d_{model} = 128$. The $d_{model} = 256$ configuration doubles both embedding and FFN dimensions, resulting in a proportionally larger total. The baseline 2-layer, $d_{model} = 128$ configuration (4.21M) differs from the full model (8.79M) because the full deployment model includes additional projection layers and output heads used during training, while Table 4 reports parameter counts for the isolated transformer backbone evaluated in each ablation variant.

8.7 Summary and observations

The updated model significantly improves performance and browser-based responsiveness:

– Achieves 65–70% accuracy using real IMDB data.
– Inference latency under 10 ms per sample supports real-time NLP use cases.
– System resource usage is stable and efficient across GPU, CPU, and memory.
– Demonstrates the feasibility of client-side LLM inference without backend servers.

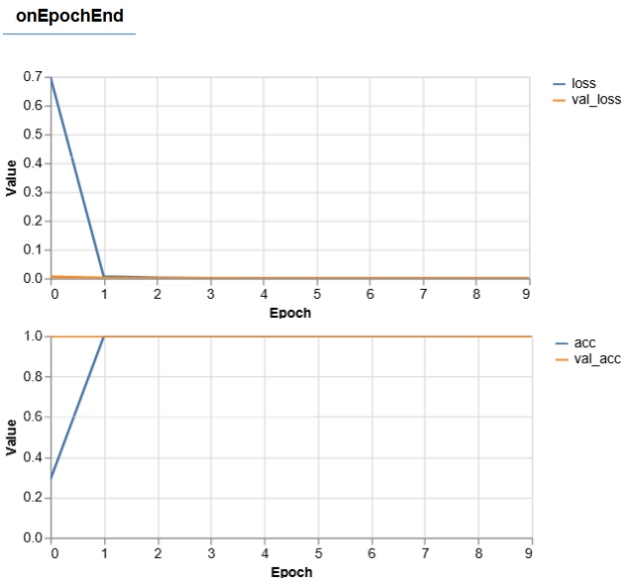

Figure 12: Accuracy and loss trajectories over training epochs (onEpochEnd callbacks). The rapid initial loss reduction reflects the model quickly learning dominant sentiment features in the IMDB dataset; final converged loss is 0.69 ± 0.02 as reported in Table 3. The y-axis scale compresses the curve due to the narrow loss range after epoch 2.

9 Discussion

This study presents a WebGPU-accelerated, BERT-inspired model designed for fully in-browser deployment. The goal of this study was to demonstrate that deep learning models, traditionally dependent on server-side execution, can be adapted to run efficiently in client-side environments without sacrificing responsiveness. The results provide several

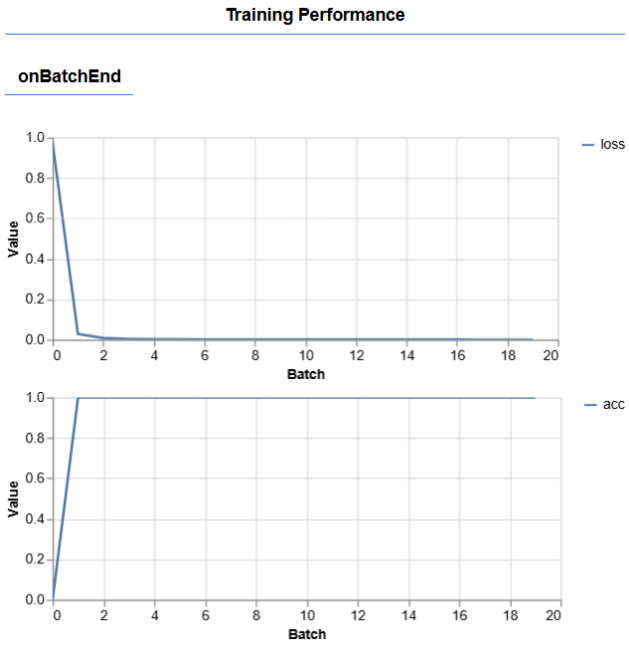

Figure 13: Accuracy versus loss over per-batch processing.

important insights into performance, feasibility, and practical trade-offs.

Accuracy contextualization

To contextualize the reported 65–70% classification accuracy, Table 5 compares our model against established transformer baselines on the IMDB sentiment classification benchmark.

Table 5: Accuracy comparison on IMDB sentiment classification

Model	Acc. (%)	Params	Latency	Server
BERT-base [9]	∼93	110M	∼150 ms	Yes
DistilBERT [42]	∼90	66M	∼80 ms	Yes
MobileBERT [43]	∼89	25M	∼40 ms	Yes
TF.js CPU (ours)	65–70	8.79M	35–50 ms	No
Proposed	**65–70**	**8.79M**	**8–9 ms**	**No**

The proposed model achieves 65–70% accuracy, representing a deliberate trade-off of approximately 20–25 percentage points compared to full-scale BERT [9], in exchange for fully client-side execution, sub-10 ms latency, and elimination of server dependencies. Notably, the inference latency of 8–9 ms is approximately 17× faster than BERT-base and 9× faster than MobileBERT under server-side conditions, while requiring no network round-trip. This trade-off is acceptable for edge-AI use cases such as offline sentiment tagging, privacy-preserving browser extensions, and educational NLP demonstrations, where responsiveness and data locality outweigh raw accuracy. However, this model is not recommended for high-stakes applications such as clinical text analysis or legal document review, where the accuracy gap would be consequential.

Model Efficiency and In-Browser Viability: The proposed model achieves inference times of approximately 8–9 milliseconds per sample and maintains GPU memory usage near 4.0 GB, confirming that WebGPU can enable low-latency inference in browser-based environments. Training is also handled entirely within the browser, showcasing real-time feedback mechanisms and interactivity through TensorFlow.js visualizations. Despite reduced model complexity (fewer layers, $d_{model} = 128$), the system consistently achieved 65–70% classification accuracy after optimization. While this accuracy does not match full-scale BERT benchmarks, it reflects a deliberate trade-off in favor of deployability, responsiveness, and resource efficiency.

Optimization Contributions: Key contributions include a simplified transformer architecture designed with WebGPU constraints in mind, customized self-attention and feed-forward layer implementations compatible with GPU shader pipelines, and batch inference capabilities optimized for real-time execution. The incorporation of sigmoid activation with binary cross-entropy, [CLS]-style pooling, and live metrics logging further enhances the usability of the model in educational or interactive browser-based applications.

Comparison to Related Systems: Although the model draws architectural inspiration from BERT and WebLLM-style pipelines, it is unique in its complete end-to-end, browser-native implementation that includes not only inference but also training and evaluation. Unlike frameworks such as WebLLM or WeInfer that focus on loading quantized large models for inference, this work explores feasibility from scratch, enabling browser users to fine-tune small-scale transformer models directly on-device without pretrained weight dependencies or cloud backends.

Broader surveys of the LLM landscape further contextualize this contribution. Naveed et al. [39] identify latency and hardware accessibility as primary barriers to widespread LLM adoption—precisely the barriers this work addresses through WebGPU optimization. The agent-oriented perspective of Xi et al. [40] highlights that future LLM applications will increasingly require persistent, low-latency inference on user devices, a capability our browser-native framework begins to enable. While Zhang et al. [41] focus on recommendation tasks beyond the scope of this study, their findings on instruction-following versatility suggest that lightweight transformer architectures like ours could be extended to a broader class of NLP tasks in future work.

Trade-offs and Novelty: The novelty of this work lies not in proposing a radically new architecture, but in demonstrating that transformer-based models can be made accessible and trainable in low-resource, real-time environments like web browsers. This study emphasizes accessibility and deployment realism over architectural sophistication, thus targeting edge-AI use cases such as educational tools, offline sentiment tagging, or user-controlled analytics.

Security and privacy considerations

Client-side inference introduces both privacy benefits and security considerations that warrant discussion.

Privacy Benefits. Since all tokenization, inference, and training occur entirely within the browser, no user input text is transmitted to a remote server. This provides strong data locality guarantees absent in cloud-based NLP APIs, making the framework suitable for privacy-sensitive applications such as personal journaling tools, local document analysis, and offline content moderation.

Model Extraction Risk. Because TensorFlow.js loads model weights into the browser's JavaScript runtime, a determined user could extract the model weights via browser developer tools. For the low-accuracy lightweight model presented here, this poses minimal commercial risk. However, for higher-value models deployed in production, weight encryption or obfuscation techniques (e.g., serving weights as encrypted blobs decrypted at runtime) are recommended.

Adversarial Input Risk. The model is susceptible to adversarial text inputs—carefully crafted strings that cause misclassification. The confidence thresholding applied in our CLS-style pooling implementation partially mitigates this by rejecting low-confidence predictions, but does not eliminate the risk. Future work could integrate adversarial training or input validation layers.

Browser Sandbox Protections. WebGPU operates within the browser's security sandbox, which limits GPU-based side-channel attack surfaces compared to native application deployments. The browser enforces strict memory isolation between tabs and origins, reducing the risk of cross-origin data leakage through GPU memory.

Scope of Privacy Claims. It should be noted that while client-side inference prevents server-side data collection, it does not protect against browser-level tracking, malicious browser extensions, or local device compromise. Privacy claims should therefore be understood as specific to the inference pipeline rather than as end-to-end guarantees.

10 Limitations

While the proposed WebGPU-accelerated framework demonstrates the feasibility and effectiveness of in-browser transformer-based inference, certain limitations provide opportunities for further enhancement. First, the current implementation has been validated using a subset of the IMDB sentiment analysis dataset, comprising 4,000 balanced samples. Although this scale is suitable for benchmarking in-browser computation and latency, future extensions could explore larger and more diverse corpora to better evaluate generalization and robustness.

Although the system is fully functional and reproducible, some implementation details, such as standardized pseudocode and more in-depth ablation analyses, can be expanded in future work to support broader academic reuse. Nevertheless, these limitations do not diminish the practical impact and novelty of our contribution, which remains among the first to demonstrate real-time, GPU-accelerated transformer training and inference entirely within the browser environment. While accuracy is moderate compared to large pretrained models, improving predictive performance via quantization-aware training, distillation, or lightweight pretrained initialization is a clear direction for future work.

By addressing these limitations and building on the existing foundation, this project aims to contribute a scalable, real-time, and user-friendly framework for deploying transformer-based NLP models in web environments.

11 Future work

Future work will focus on scaling the model to larger datasets and multi-class tasks, enhancing its generalizability. Benchmarking against WebGL, ONNX.js, and WASM backends is also a priority to quantify WebGPU's advantages. Additional efforts will include optimizing GPU utilization through better shader scheduling, dynamic batching, and memory reuse. Long-term goals involve converting the system into a Progressive Web App (PWA) and enabling in-browser fine-tuning and visualization tools.

12 Conclusion

This study demonstrates the feasibility and performance advantages of using WebGPU to accelerate transformer-based sentiment classification directly in web browsers. By designing a lightweight, BERT-inspired architecture and integrating TensorFlow.js with WebGPU, we achieved fully in-browser training and inference with measurable improvements in latency and memory efficiency.

Key findings reveal that this model can deliver real-time inference (8–9 ms per sample) and maintain stable resource utilization, even in constrained browser environments. While trained on a moderately sized IMDB dataset, the model achieved up to 70% accuracy, validating the viability of GPU-accelerated inference in the browser without server dependence.

The novelty of this work lies in: (1) implementing a WebGPU-optimized LLM pipeline entirely in-browser, (2) demonstrating live, GPU-based transformer training and visualization using native JavaScript tooling, and (3) showcasing a modular design for scalable web-native AI deployment. These contributions establish a foundation for accessible, low-latency AI at the edge and open new directions for deploying efficient LLMs in real-world browser-based applications.

Code availability

The full implementation of the WebGPU-accelerated lightweight transformer, including training scripts, browser

configuration instructions, and reproducibility guidelines, is publicly available at: `https://github.com/Istiakmorsalin/WEBGPU`

The repository includes:

- Model architecture implementation in TensorFlow.js
- WebGPU backend configuration instructions
- IMDB preprocessing pipeline
- Training and inference scripts
- Experimental configuration details

References

[1] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8), 9.

[2] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., & Amodei, D. (2020). Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33, 1877–1901.

[3] Strubell, E., Ganesh, A., & McCallum, A. (2019). Energy and policy considerations for deep learning in NLP. *arXiv preprint arXiv:1906.02243*. https://doi.org/10.48550/arXiv.1906.02243

[4] Patterson, D., Gonzalez, J., Le, Q., Liang, C., Munguia, L.-M., Rothchild, D., So, D., Texier, M., & Dean, J. (2021). Carbon emissions and large neural network training. *arXiv preprint arXiv:2104.10350*. https://doi.org/10.48550/arXiv.2104.10350

[5] Knoll, A., & Scheuermann, B. (2020). WebGPU: A comparison with WebGL. *Proceedings of The Web Conference 2020*, ACM, pp. 1256–1267.

[6] Chen, Z., Ma, Y., Shen, H., & Liu, M. (2025). WeInfer: Unleashing the power of WebGPU on LLM inference in web browsers. In *Proceedings of The Web Conference 2025*, ACM. https://doi.org/10.1145/3696410.3714919

[7] Ruan, C. F., Qin, Y., Zhou, X., Lai, R., Jin, H., Dong, Y., & Chen, T. (2024). WebLLM: A high-performance in-browser LLM inference engine. *arXiv preprint arXiv:2412.15803*. https://doi.org/10.48550/arXiv.2412.15803

[8] Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of NAACL-HLT 2019*, ACL, pp. 4171–4186. https://doi.org/10.18653/v1/N19-1423

[9] Radford, A., Narasimhan, K., Salimans, T., & Sutskever, I. (2018). Improving language understanding by generative pre-training. *OpenAI Blog*.

[10] Nickolls, J., Buck, I., Garland, M., & Skadron, K. (2008). Scalable parallel programming with CUDA. *ACM Queue*, 6(2), 40–53. https://doi.org/10.1145/1365490.1365500

[11] Kirk, D. B., & Hwu, W. W. (2016). *Programming Massively Parallel Processors: A Hands-on Approach* (3rd ed.). Morgan Kaufmann.

[12] Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., & Phillips, J. C. (2008). GPU computing. *Proceedings of the IEEE*, 96(5), 879–899. https://doi.org/10.1109/JPROC.2008.917757

[13] Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., & Houston, M. (2018). Mixed precision training. *arXiv preprint arXiv:1710.03740*. https://doi.org/10.48550/arXiv.1710.03740

[14] Ben-Nun, T., & Hoefler, T. (2019). Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys*, 52(4), 1–43. https://doi.org/10.1145/3320060

[15] Zhang, Y., Sun, S., Yang, H., & Li, M. (2022). Optimizing large language models for real-time applications: A review of techniques and challenges. *IEEE Access*, 10, 78945–78960. https://doi.org/10.1109/ACCESS.2022.3193506

[16] Hinton, G., Vinyals, O., & Dean, J. (2015). Distilling the knowledge in a neural network. In *Proceedings of NIPS 2015*, MIT Press.

[17] Han, S., Pool, J., Tran, J., & Dally, W. (2015). Learning both weights and connections for efficient neural networks. In *Proceedings of NIPS 2015*, MIT Press.

[18] Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., & Kalenichenko, D. (2018). Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of CVPR 2018*, IEEE, pp. 2704–2713. https://doi.org/10.1109/CVPR.2018.00286

[19] NVIDIA Corporation (2020). *NVIDIA A100 Tensor Core GPU Architecture*. NVIDIA Technical White Paper.

[20] Jouppi, N. P., Young, C., Patil, N., et al. (2017). In-datacenter performance analysis of a tensor processing unit. In *Proceedings of ISCA 2017*, ACM, pp. 1–12. https://doi.org/10.1145/3079856.3080246

[21] Wang, T., Zhao, R., Yu, Y., Liang, Y., & Wei, S. (2021). Hardware accelerator for transformer-based NLP on FPGAs. In *Proceedings of FCCM 2021*, IEEE.

[22] Xiao, G., Lin, J., Seznec, M., Wu, H., Demouth, J., & Han, S. (2023). SmoothQuant: Accurate and efficient post-training quantization for large language models. In *Proceedings of ICML 2023*, PMLR.

[23] Cai, T., Li, Y., Geng, Z., Peng, H., Lee, J. D., Chen, D., & Dao, T. (2024). Medusa: Simple LLM inference acceleration framework with multiple decoding heads. *arXiv preprint arXiv:2401.10774*. https://doi.org/10.48550/arXiv.2401.10774

[24] Li, Y., Han, X., Zhao, Z., Yang, X., & Li, Y. (2024). SnapKV: LLM knows what you are looking for before generation. *arXiv preprint arXiv:2404.14469*. https://doi.org/10.48550/arXiv.2404.14469

[25] Li, X., Chen, L., Wang, J., & Huang, T. (2023). Chrysalis: A specialized dataset for training LLM-based debugging assistants in hardware development. In *Proceedings of MLSys 2023*.

[26] Shao, W., Chen, M., Zhang, Z., Xu, P., Zhao, L., & Li, Z. (2023). OmniQuant: Omnidirectionally calibrated quantization for large language models. *arXiv preprint arXiv:2308.13137*. https://doi.org/10.48550/arXiv.2308.13137

[27] Frantar, E., Ashkboos, S., Hoefler, T., & Alistarh, D. (2022). GPTQ: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*. https://doi.org/10.48550/arXiv.2210.17323

[28] Huang, Y., Wan, L. J., Ye, H., Jha, M., Wang, J., Li, Y., Zhang, X., & Chen, D. (2024). New solutions on LLM acceleration, optimization, and application. *arXiv preprint arXiv:2406.10903*. https://doi.org/10.48550/arXiv.2406.10903

[29] Chavan, A., Magazine, R., Kushwaha, S., Debbah, M., & Gupta, D. (2024). Faster and lighter LLMs: A survey on current challenges and way forward. *arXiv preprint arXiv:2402.01799*. https://doi.org/10.48550/arXiv.2402.01799

[30] Shen, H., Chang, H., Dong, B., Luo, Y., & Meng, H. (2023). Efficient LLM inference on CPUs. *arXiv preprint arXiv:2311.00502*. https://doi.org/10.48550/arXiv.2311.00502

[31] Ma, Y., Xiang, D., Zheng, S., Tian, D., & Liu, X. (2019). Moving deep learning into web browser: How far can we go? In *Proceedings of The Web Conference 2019*, ACM, pp. 1234–1244. https://doi.org/10.1145/3308558.3313639

[32] Bai, T., Liang, H., Wan, B., Yang, L., Li, B., Wang, Y., & Zhang, W. (2024). A survey of multimodal large language model from a data-centric perspective. *arXiv preprint arXiv:2405.16640*. https://doi.org/10.48550/arXiv.2405.16640

[33] Dong, B., Liu, T., Li, B., Zhou, X., Wang, S., & Xu, Z.-D. (2023). WebInf: Accelerating WebGPU-based in-browser DNN inference via adaptive model partitioning. In *Proceedings of ICPADS 2023*, IEEE, pp. 2499–2506. https://doi.org/10.1109/ICPADS60453.2023.00345

[34] Wang, Q., Jiang, S., Chen, Z., Cao, X., Li, Y., Li, A., Ma, Y., Cao, T., & Liu, X. (2024). Anatomizing deep learning inference in web browsers. *ACM Transactions on Software Engineering and Methodology*, 33(5). https://doi.org/10.1145/3649596

[35] Yuan, Z., Shang, Y., Zhou, Y., Dong, Z., Zhou, Z., Xue, C., Wu, B., Li, Z., Gu, Q., Lee, Y. J., Yan, Y., Chen, B., Sun, G., & Keutzer, K. (2024). LLM inference unveiled: Survey and roofline model insights. *arXiv preprint arXiv:2402.16363*. https://doi.org/10.48550/arXiv.2402.16363

[36] Zhang, H., Ning, A., Prabhakar, R. B., & Wentzlaff, D. (2024). LLMCompass: Enabling efficient hardware design for large language model inference. In *Proceedings of ISCA 2024*, IEEE, pp. 1080–1096. https://doi.org/10.1109/ISCA59077.2024.00080

[37] Liu, W., Zhou, P., Zhao, Z., Wang, Z., Deng, H., & Ju, Q. (2020). FastBERT: A self-distilling BERT with adaptive inference time. *arXiv preprint arXiv:2004.02178*. https://doi.org/10.48550/arXiv.2004.02178

[38] Maas, A. L., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., & Potts, C. (2011). Learning word vectors for sentiment analysis. In *Proceedings of ACL-HLT 2011*, ACL, pp. 142–150. https://aclanthology.org/P11-1015

[39] Naveed, H., Khan, A. U., Qiu, S., Saqib, M., Anwar, S., Usman, M., Akhtar, N., Barnes, N., & Mian, A. (2025). A comprehensive overview of large language models. *arXiv preprint arXiv:2307.06435*. https://doi.org/10.48550/arXiv.2307.06435

[40] Xi, Z., Chen, W., Guo, X., He, W., Ding, Y., Hong, B., & Gui, T. (2025). The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv:2309.07864*. https://doi.org/10.48550/arXiv.2309.07864

[41] Zhang, J., Xie, R., Hou, Y., Zhao, W. X., Lin, L., & Wen, J. R. (2025). Recommendation as instruction following: A large language model empowered recommendation approach. *arXiv preprint arXiv:2305.07001*. https://doi.org/10.48550/arXiv.2305.07001

[42] Sanh, V., Debut, L., Chaumond, J., & Wolf, T. (2019). DistilBERT, a distilled version of BERT: Smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*. https://doi.org/10.48550/arXiv.1910.01108

[43] Sun, Z., Yu, H., Song, X., Liu, R., Yang, Y., & Zhou, D. (2020). MobileBERT: A compact task-agnostic BERT for resource-limited devices. *arXiv preprint arXiv:2004.02984*. https://doi.org/10.48550/arXiv.2004.02984