# Methodologies and Architectures for AI Inference Hardware: From Foundational Networks to Large Language Models

Huwan Peng

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2025

Reading Committee:

Michael Taylor, Chair

C.J. Richard Shi, Chair

Ang Li

Program Authorized to Offer Degree:
Electrical and Computer Engineering

University of Washington

**Abstract**

Methodologies and Architectures for AI Inference Hardware: From Foundational Networks to Large Language Models

Huwan Peng

Co-Chairs of the Supervisory Committee:
Michael Taylor
Department of Electrical and Computer Engineering

C.J. Richard Shi
Department of Electrical and Computer Engineering

The rapid advancements in large language models (LLMs) have significantly reshaped the artificial intelligence landscape, enabling transformative applications. However, these developments pose profound challenges for hardware architectures, particularly concerning performance, efficiency, and scalability. This dissertation investigates these critical challenges, proposing novel methodologies and architectural designs for specialized hardware, with a primary focus on optimizing large-scale LLM inference.

Core contributions of this thesis include the development of ReaLLM and Chiplet Cloud. ReaLLM is a holistic simulation framework for LLM serving, designed to bridge detailed accelerator-level insights with system-wide performance evaluations. This framework facilitates rapid exploration and precise simulation of both hardware architectures and software strategies. Chiplet Cloud is a cloud-scale architecture optimized for the Total Cost of Ownership (TCO) of LLM inference. Its key architectural innovations include fitting model parameters within on-chip memory to improve performance, co-optimizing chip size with software mapping to reduce TCO, and effectively exploiting model sparsity to support larger models.

Additionally, the thesis discusses ChronoStack, a 3D memory architecture developed as part of a collaborative research effort, featuring a novel Time-Multiplexed KV-Prefetching

technique, specifically optimized for the demands of long-context LLMs. The dissertation also incorporates foundational research on accelerators for earlier AI paradigms, including Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), and deep reinforcement learning, providing a broad perspective on AI hardware evolution.

Together, this body of work presents a detailed investigation into architectures and methodologies for AI inference hardware, tracing a clear progression from foundational network acceleration to modern large language model serving. The research aims to contribute novel approaches and critical insights towards achieving efficient, high-performance computing for the advancing field of artificial intelligence.

## TABLE OF CONTENTS

Page

## LIST OF FIGURES

## ACKNOWLEDGMENTS

Completing this PhD journey feels surreal. Though I've imagined this moment countless times, it is still hard to believe that it has finally arrived. One of my greatest motivations to persevere through the challenges was precisely this chance—to formally thank the many incredible individuals who have supported me along the way.

First and foremost, my deepest gratitude goes to my advisors, Professor Richard Shi and Professor Michael Taylor. My journey into the world of research began with Richard nearly a decade ago, during my undergraduate years. His guidance was instrumental in sparking my passion for this field, and I am immensely thankful for his mentorship. Michael has profoundly shaped me as a researcher, teaching me not just the intricacies of research but also the skills of effective writing, presenting, and handling public discourse.

I am also grateful to my committee members, Professor Ang Li and Professor James Zhang, for their insightful feedback and for dedicating their time to my doctoral work.

Throughout my PhD, I have been fortunate to learn from and collaborate with exceptional mentors and colleagues. I owe a special thanks to Professor Chixiao Chen. Leading a small and relatively inexperienced team, including myself, to successfully tape out two chips in such a short period was an incredible feat. His dedication, hard work, and passion for his research were truly inspiring, and I cherish the memories of our time working together. I must also extend a heartfelt thank you to Scott Davidson. Though a fellow PhD student, Scott is one of the most knowledgeable individuals I have ever met. I thoroughly enjoyed every conversation we had and deeply value our collaboration and friendship. My gratitude also goes to my other esteemed collaborators: Professor Dustin Richmond, Professor Shuaiwen Leon Song, Ameen Akel, and Xingyao Zhang. Their contributions and insights have been invaluable. I also want to thank Timothy Heil, my manager during my internship at Microsoft, for the guidance and opportunities he provided.

At UW, I was fortunate to be part of two exceptional research groups: SSRL and BSG. These communities provided not only intellectual stimulation but also lasting friendships. Many thanks to Hongwei Ding, Ailing Piao, Professor Aili Wang, Chang Liu, Rongjin Xu, and Jialin Wang from SSRL. A special acknowledgment goes to Cindy Liu, with whom mutual encouragement helped us both reach this finish line. From BSG, I warmly thank Shaolin Xie, Chun Zhao, Paul Gao, Dan Petrisko, Tommy Jung, Farzam Gilani, and Rico Li. I want to offer a special word of encouragement to Yuan-Mao Chueh and Anoop Mysore. You are both talented researchers and wonderful friends; keep pushing forward, your hard work will undoubtedly pay off.

Outside of academia, I am blessed with many supportive friendships. Xiyang Liu has been an invaluable friend throughout this journey, particularly during the challenging COVID-19 period. Our weekend hangouts were often the only moments of relaxation during those challenging times. His serious approach to both research and life has been a source of inspiration. Thanks also to my landlord Ms. Wu and my roommate Tom, whose kindness and support have greatly eased my daily life. A special thank you to the Seattle United Running Club. The warmth and friendliness of everyone I met there made me feel at home in a foreign land for the first time. There are so many other friends who offered help and support along the way, and I am wholeheartedly thankful for each and every one of you, even if I cannot list all of your names here.

My deepest and most heartfelt gratitude goes to my family. To my parents, thank you for your unconditional love and unwavering support. It has been a source of regret that I have been away from home for so long and unable to visit as much as I would have liked during my PhD. Despite the distance, I have always felt your love, and it has been a constant source of strength.

Last, but certainly not least, I want to thank my beloved wife, Xuanfeng. Your constant support, understanding, and respect as I pursued this often lonely path have meant everything to me. For most of this journey, we were separated by nations, and I know you endured your own difficult and lonely times. You are more resilient and stronger than I am. Thank

you for believing in me, even during moments when I doubted myself. This achievement is as much yours as mine. It is, profoundly and unequivocally, all because of you.

Portions of this work were partially supported by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement numbers FA8650-18-2-7863 and FA8650-18-2-7856; NSF grants SaTC-1563767, SaTC-1565446. This work intersects and leverages research and infrastructure created by the members of the Bespoke Silicon Group, spanning across accelerators ([20, 53, 221, 210, 142, 140, 224, 17, 204, 162, 61, 175, 10, 163, 60, 62, 95, 203, 13]), ML ([202]), ASIC Clouds ([194, 211, 191, 181, 96, 97, 124, 183]), open source hardware ([193, 184, 50]) RISC-V ([147, 159, 158, 42, 223, 7, 201, 88, 156, 130, 35, 155, 115, 91]), Network-on-Chips ([89, 148, 226, 188, 101]), security ([23, 12, 22, 69, 70, 8]), benchmark suites ([196, 105, 14]), dark silicon ([180, 179, 180, 182, 18, 63, 60, 203]), multicore ([31, 30, 147, 67, 65, 66, 178, 188, 190, 185, 177, 101, 186, 189, 187, 205]), compiler tools ([82, 56, 80, 58, 81, 79, 57, 218, 2, 11]) and FPGAs ([225, 76, 14]).

## DEDICATION

To my mother, Xiuhong Hu.

# Chapter 1

# INTRODUCTION

The last decade has seen significant advancements in the capabilities and applications of Artificial Intelligence (AI), leading to transformative changes across various sectors. From influencing industries like healthcare and finance to enabling everyday applications such as natural language translation, recommendation systems, and autonomous vehicles, AI is increasingly impacting modern life [160]. Central to these advancements are sophisticated machine learning models, particularly Deep Neural Networks (DNNs), which have demonstrated notable performance across a spectrum of complex tasks [114].

The trajectory of AI progress has been characterized by a trend towards developing larger and more complex models. While foundational network architectures like Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) established key principles, the recent emergence of Large Language Models (LLMs) such as GPT-3 [21], PaLM [34], and Llama [197] has enabled significant progress in natural language understanding, generation, and reasoning. These models, often comprising hundreds of billions or even trillions of parameters [51], generally follow scaling laws which suggest that model performance improves with increased scale (data, parameters, and computation) [93]. However, this demand for increased scale translates directly into substantial growth in computational requirements for both training these models and, importantly, for deploying them for inference at scale. The computational power needed to serve these models to many users in real-time presents a significant challenge, straining existing hardware infrastructure and impacting operational costs and energy consumption.

The computational support for the current AI developments has largely been provided by general-purpose hardware, primarily Central Processing Units (CPUs) and Graphics Processing Units (GPUs). While CPUs offer flexibility, their architecture is not ideally suited for the massively parallel computations that are prevalent in AI workloads. GPUs, origi-

nally developed for graphics rendering, have been widely adopted for AI due to their parallel processing capabilities and have become common solutions for training and deploying many AI models [133, 134]. However, as AI models, especially LLMs, continue to grow in size and complexity, the limitations of these general-purpose architectures are becoming more evident. GPUs, despite their power, retain overheads from their general-purpose design and may not provide the optimal balance of performance, power efficiency, and cost when deployed at the scale required by modern AI services. A key challenge is the *memory wall*, where the rate of improvement in processor speed has outpaced improvements in memory bandwidth and latency [208]. LLM inference, characterized by large model sizes and substantial intermediate data (such as the KV cache in Transformers [200]), is often memory-bandwidth bound, particularly during the token generation phase [150]. This can lead to underutilization of compute units and reduce overall system efficiency. Furthermore, the energy consumption associated with running these large models on general-purpose hardware at scale raises considerations regarding environmental sustainability and economic viability [144].

To address the performance, efficiency, and scalability limitations of general-purpose hardware, the development of Application-Specific Integrated Circuits (ASICs) tailored for AI workloads has become an important direction [176]. ASICs offer the potential for substantial improvements in performance and energy efficiency by designing hardware specifically optimized for the computational patterns and data movement characteristics of AI algorithms, such as matrix multiplications, convolutions, and attention mechanisms. By removing unnecessary general-purpose logic and co-designing the architecture with the target algorithms, AI ASICs can achieve higher throughput and lower power consumption per operation [157]. While the Non-Recurring Engineering (NRE) costs associated with ASIC design are considerable [98], the large volume of computations required for large-scale LLM deployment can make this investment practical. The significant operational Total Cost of Ownership (TCO) of running LLM inference on existing hardware means that even moderate improvements in TCO per token, offered by a specialized ASIC, can lead to considerable savings, potentially amortizing the NRE costs. These economic factors, along with the continued demand for AI, are driving the development of specialized AI hardware.

The primary objective of this doctoral research is:

*To design and evaluate high-performance and efficient ASIC accelerator architectures and supporting methodologies for diverse and evolving AI applications, with a primary focus on Large Language Models.*

This dissertation examines the challenges in AI hardware and presents methodologies and architectural designs to address them. The work is organized as follows, with each chapter detailing specific contributions toward this goal:

- **Chapter 2: Background** provides an overview of fundamental AI models, including early paradigms like CNNs, RNNs and deep reinforcement learning, as well as the architecture of modern Large Language Models. It also discusses system techniques for AI inference, principles of specialized hardware acceleration, and the economic considerations motivating ASIC development.

- **Chapter 3: Foundational Work: Accelerating Early AI Paradigms** details initial research on accelerating earlier AI workloads. This chapter discusses contributions to the development of **iFPNA** [26, 25], a flexible deep learning processor adaptable to various network types, and presents **DRLP**, a specialized accelerator for Deep Q-Learning that introduced the efficient **F|B|C (Filter-Batch-Channel) dataflow**. This foundational work provided insights into architectural flexibility, dataflow optimization, and hardware-software co-design relevant to subsequent research.

- **Chapter 4: ReaLLM: A Holistic Simulation Framework for LLM Serving** introduces **ReaLLM**, a comprehensive, multi-level hardware system simulation framework developed to address the complexity of evaluating modern LLM inference systems [146]. It is designed to bridge detailed accelerator-level insights with system-wide performance evaluations, enabling exploration and simulation of hardware architectures and software strategies for LLM serving through features like a precomputed kernel library and trace-driven analysis.

- **Chapter 5: Chiplet Cloud: A TCO-Optimized LLM Hardware Architecture** proposes **Chiplet Cloud**, a cloud-scale, chiplet-based ASIC architecture [145].

This work focuses on the economic viability of large-scale LLM deployment and is optimized for the Total Cost of Ownership (TCO) of LLM inference. Key aspects include fitting model parameters within on-chip memory, co-optimizing chip size with software mapping, and exploiting model sparsity.

- **Chapter 6: ChronoStack: A 3D-Memory Architecture for Long-Context LLMs** describes **ChronoStack**, a 3D memory architecture developed as part of a collaborative research effort to which the author contributed, designed to tackle the memory bandwidth challenges posed by increasing context lengths in LLMs. It features a **Time-Multiplexed KV-Prefetching** technique, aiming to leverage the high bandwidth of hybrid-bonded 3D DRAM to accelerate attention operations and improve end-to-end latency in long-context LLMs.

- **Chapter 7: Conclusion** summarizes the key findings of this dissertation, reiterates the main contributions of the research, and discusses potential directions for future work in the field of AI hardware.

Collectively, the research detailed in these chapters offers an investigation into architectures and methodologies for AI inference hardware, covering aspects from foundational network acceleration to the specific requirements of contemporary large-scale language model serving.

**Bibliographies:** The work of Section 3.1 on iFPNA was published at [25] and [26]. The work of Chapter 4 ReaLLM simulator will be published at [146], and opensource at `https://github.com/bespoke-silicon-group/reallm`. The work of Chapter 5 Chiplet Cloud was published at [145].

# Chapter 2

# BACKGROUND

## *2.1 Fundamentals of Early Models*

In the landscape of artificial intelligence, early progress was profoundly influenced by foundational models such as Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), and the emergence of Deep Reinforcement Learning (DRL) algorithms. This section presents a concise overview of the core architectures and methodologies that define CNNs, RNNs, and DRL.

### *2.1.1 Convolutional Neural Networks and Recurrent Neural Networks*

Convolutional Neural Networks (CNNs) are specialized neural networks designed to process grid-like data structures such as images. They leverage spatial locality by using convolutional layers, pooling layers, and fully connected layers. The convolution operation for a two-dimensional input $g(x, y)$ with a two-dimensional filter $f(i, j)$ is defined as:

$$(f * g)(x, y) = \sum_i \sum_j f(i, j) \cdot g(x - i, y - j) \tag{2.1}$$

This process extracts hierarchical feature maps that encode edges, textures, and complex patterns. Pooling layers reduce spatial resolution to improve computational efficiency and translation invariance. The fully connected layers map these high-level features to output classes. CNNs have dramatically advanced tasks like image classification, exemplified by networks such as AlexNet [106] and ResNet [71].

Recurrent Neural Networks (RNNs) are designed to handle sequential data, maintaining a hidden state that captures temporal dependencies. The basic RNN computation at time step $t$ is:

$$
\begin{aligned}
h_t &= \sigma(W_{xh}x_t + W_{hh}h_{t-1} + b_h) \\
o_t &= W_{ho}h_t + b_o
\end{aligned}
\tag{2.2}
$$

where $\sigma$ is the non-linear function such as $tanh$, $o_t$ is the output, $h_t$ is the hidden state, $x_t$ the current input, and $W_{ho}$, $b_o$, $W_{xh}$, $W_{hh}$, and $b_h$ are learnable parameters. Traditional RNNs often suffer from vanishing or exploding gradients; hence, LSTMs [73] and GRUs [36] were introduced, utilizing gating mechanisms to manage information flow effectively.

CNNs, RNNs, and their variants have distinct architectural patterns and computational characteristics. To effectively support the wide diversity of models and operations, the underlying hardware must be designed with versatility in mind. In Section 3.1, we present a flexible deep neural network accelerator architecture that can efficiently execute all these workloads.

### *2.1.2 Deep Reinforcement Learning*

Deep Reinforcement Learning (DRL) combines reinforcement learning with deep neural networks to allow agents to learn optimal policies from environment interactions. It is applied to applications that need to make real-time decisions continuously, such as video and board game playing, and robot motion control[171, 161].

One common reinforcement learning methodology is called *Q-learning method*. It learns a value function $q(s, a)$, that estimates the expectation of the discounted return, or value, of an action in a certain state. Taking one of the most popular DRL algorithms, DQN [128], as an example. It uses deep neural networks to approximate the value function. It takes the environment states as inputs and outputs the value of each valid action. Unlike other DNNs, DRL generates datasets for training by interacting with the environment.

Similar to DNNs, DRL is also optimized to minimize the loss $E$ between the network output and the target output $\hat{O}$,

$$
E = (q_\theta(S_i; A_i) - \hat{O})^2 \tag{2.3}
$$

where $q_\theta$ is the value function (DNN) with parameter $\theta$, also called *online network*. The time step $i$ is randomly sampled from the replay memory. The target output of DQN is

defined as,

$$\hat{O} = R_i + \gamma \max_{a'} q_{\bar{\theta}}(S_{i+1}; a') \tag{2.4}$$

The intuition of this function is the value $q$ of a state $S_i$ with an action $A_i$ is equal to the immediate reward $R_i$ plus the maximum $q$ value of the next state $S_{i+1}$. $\bar{\theta}$ represents the parameters of a *target network*. It is a periodic copy of the online network, and not directly optimized. This helps to reduce the correlation with the target value.

DQN has been widely used in later Q-learning algorithms, such as Rainbow [72], Ape-X [75], and R2D2 [94]. In Section 3.2, we design an accelerator that optimizes these Q-learning DRL algorithms.

### *2.1.3 Neural Network Training*

Neural network training adjusts model parameters to minimize a loss function through iterative optimization. The primary three phases are Forward Propagation (FP), Backward Propagation (BP), and Weight Gradient Computation (WG). In Figure 2.1, we illustrate the tensors and operations on a convolutional layer in these phases. Each phase has two input tensors and one output tensor, which are addressed using seven dimensions. $B$ refers to the input batch, $C$ refers to the input channel, $F$ refers to the number of filters and the output channel. $R$, $H$, and $E$ refer to the 2D spatial size of filter weight, input and output activation, respectively, with $x$, $y$ are the 2D spatial coordinates. The color of the tensor indicates whether it is filter weight $W$ (blue), input activation $I$ (yellow) or output activation $O$ (green). Gradient tensors are represented by the dashed lines, such as the output activation gradient $\frac{\partial E}{\partial O}$ in BP.

During FP, a sliding-window 3D convolution is performed by filter weight $W$ and input activation $I$ to generate the output activation $O$. The detailed calculation process is shown in a 7-level nested loop, corresponding to 7 different dimensions across 3 tensors. The operation pattern in BP retains unchanged as what in FP. Note that the filter weight $W$ are transposed at the $C$ and $F$ dimensions, and each plane of $W^{\top}_{R_x \times R_y}$ is rotated 180 degrees.

**Gradient aggregation embeds in gradient computation.** Normally, the operations in WG phase are shown at the left side of Figure 2.1-(3). The weight gradients of the loss

on each sample in a batch, i.e., $\frac{\partial e_1}{\partial W}, ..., \frac{\partial e_N}{\partial W}$, are computed first through a 2D convolution between the input activation and its corresponding output activation gradients. Then, the optimization function uses the mean values of all weight gradients to update the weight. The data movement for gradient aggregation becomes a performance bottleneck for DRL and many other DNNs training using data parallelism [119, 120, 116]. When we design acceleartor for DRL in Section 3.2, we combine the weight gradient computation and aggregation in one phase. As shown in the right side of Figure 2.1-(3), the dimension $B$ of $\frac{\partial E}{\partial O}$ and $I$ becomes the second axes of the tensor, which is accumulated during the computation. This combines the weight gradient computation and aggregation in one unified operation, thereby eliminating unnecessary data movement. Meanwhile, the operation pattern in WG is also unified with FP and BP as the same 7-level nested loop through this reshaping. Therefore, the dataflow introduced in Section 3.2 can be seamlessly applied to these three phases.

## *2.2 Fundamentals of Large Language Models*

### *2.2.1 Decoder-Only Transformer Architecture*

Large Language Models (LLMs) predominantly utilize the Transformer architecture, introduced by Vaswani et al. [200]. Specifically, decoder-only Transformers are employed for autoregressive tasks such as text generation. The architecture, as shown in Figure 2.2 comprises a stack of multiple identical decoder blocks, each containing a **Multi-Head Self-Attention (MHSA)** mechanism, a **Feed-Forward Network (FFN)**, **Residual Connections** and **Layer Normalization**.

Each token in the prompt is first embedded into a vector space and combined with positional encodings to retain sequence information. The output of the final decoder block is projected onto the vocabulary space using a linear transformation followed by a softmax function to obtain probability distributions over the next token.

*Multi-Head Self-Attention (MHSA)*

The self-attention mechanism allows the model to weigh the importance of different tokens in the input sequence when encoding a particular token. For an input sequence represented

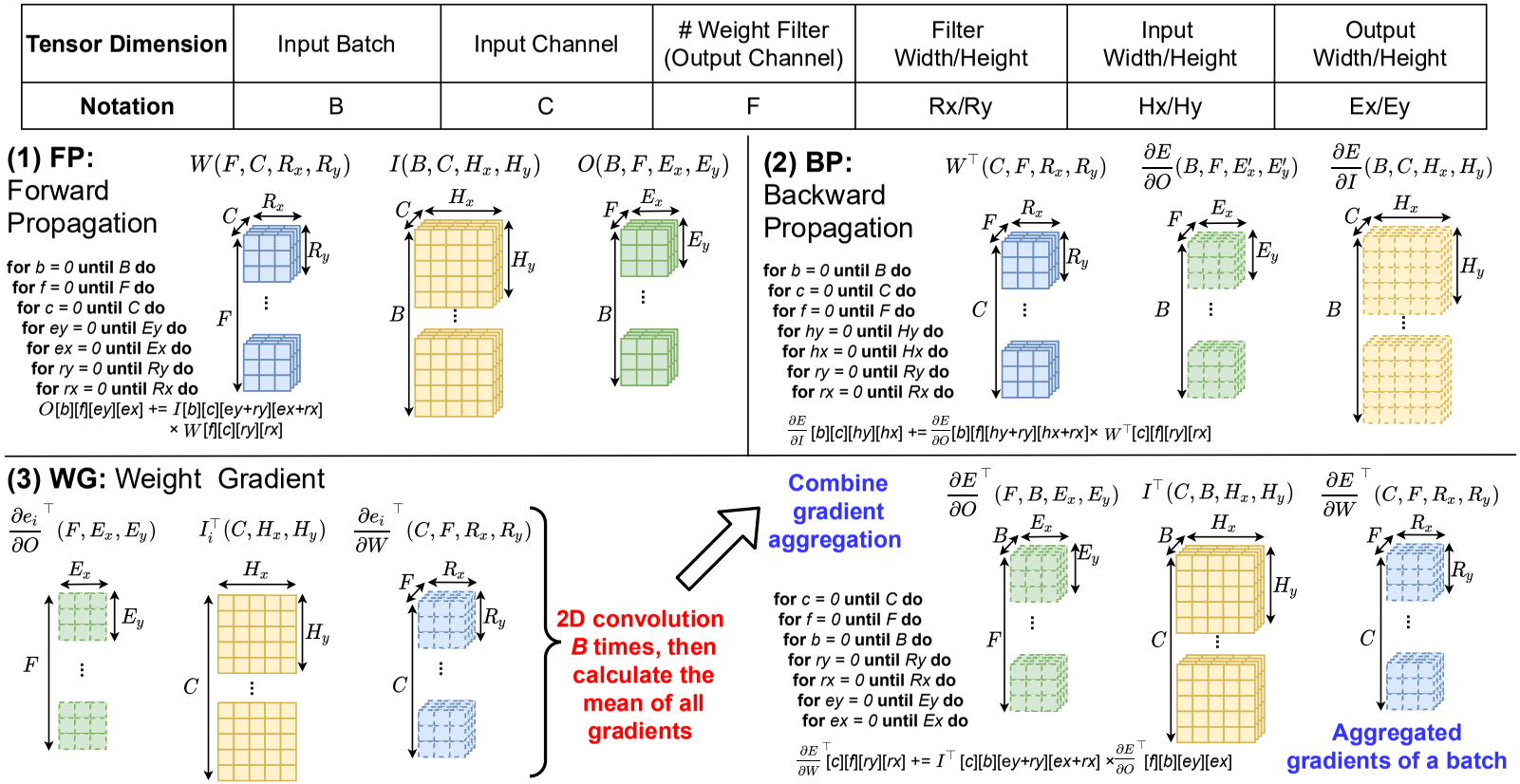



Figure 2.1: Tensors and operations in three phases of a convolutional layer are shown in the block diagram and loop nest, with a unified pattern.

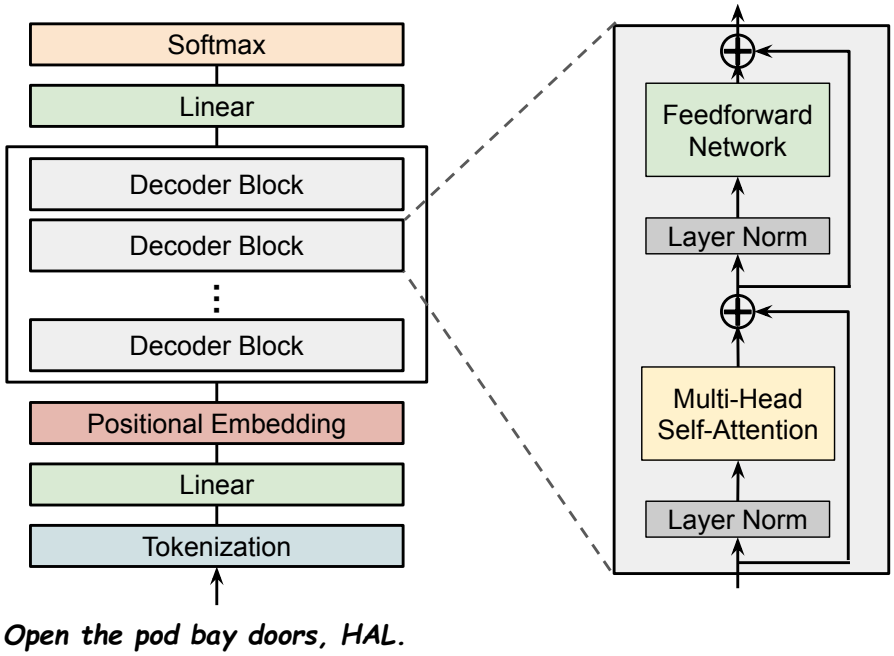



Figure 2.2: The decoder-only transformer model architecture.

by matrix $X \in \mathbb{R}^{n \times d}$, where $n$ is the sequence length and $d$ is the model dimension, the self-attention is computed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V \tag{2.5}$$

Here, the queries $Q$, keys $K$, and values $V$ are linear projections of the input:

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V \tag{2.6}$$

where $W^Q, W^K, W^V \in \mathbb{R}^{d \times d_k}$ are learnable weight matrices, and $d_k$ is the dimension of the key vectors.

To capture different types of relationships and features, multiple self-attention mechanisms, known as *heads*, are run in parallel:

$$\text{MHSA}(X) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \tag{2.7}$$

where each head is computed as:

$$\text{head}_i = \text{Attention}(XW_i^Q, XW_i^K, XW_i^V) \tag{2.8}$$

and $W^O \in \mathbb{R}^{hd_k \times d}$ is a learnable projection matrix.

To improve efficiency and scalability, several variants of the attention mechanism have been proposed:

- **Multi-Query Attention (MQA)**: Uses a single set of keys and values for all heads, reducing memory usage and computational cost [167].

- **Grouped-Query Attention (GQA)**: Groups multiple queries to share keys and values, balancing between MHA and MQA in terms of performance and efficiency [6]. Figure 2.3 shows the block diagram of a transformer block with GQA.

- **Multi-head Latent Attention (MLA)**: Compresses key and value inputs into lower-dimensional latent vectors, which significantly reduces the Key-Value (KV) cache size for more efficient inference [46, 47].

These variants are particularly beneficial during the decoding phase of inference, where efficiency is critical.

*Feed-Forward Network (FFN)*

Each Transformer layer includes a FFN, which usually consists of two linear projections with a non-linear activation in between:

$$\text{FFN}(x) = \sigma(xW_1)W_2 \tag{2.9}$$

where $\sigma$ is the non-linear activation such as ReLU, $W_1 \in \mathbb{R}^{d \times d_{ff}}$, $W_2 \in \mathbb{R}^{d_{ff} \times d}$, and $d_{ff}$ is typically larger than $d$, allowing the model to capture complex patterns [200].

Variants using Gated Linear Units (GLU) introduce a different structure [168]. An FFN with GLU, such as SwiGLU (using the Swish activation) or GEGLU (using GELU),

employs three linear projections to create a gating mechanism that can selectively control the information flow, often leading to improved performance and training dynamics:

$$\mathrm{FFN}_{\mathrm{GLU}}(x) = (\sigma(xW) \otimes xV)W_2 \tag{2.10}$$

where $xV$ acts as the gate and is element-wise multiplied by the activated output of the first linear projection. Figure 2.3 shows the block diagram of a transformer block with SwiGLU.

*Mixture-of-Experts*

Mixture-of-Experts (MoE) models introduce sparsity by activating only a subset of expert networks for each input, allowing for larger models without a proportional increase in computational cost. The routing mechanism determines which experts are activated:

$$\mathrm{MoE}(x) = \sum_{i=1}^{E} G_i(x) \cdot \mathrm{Expert}_i(x) \tag{2.11}$$

where $E$ is the total number of experts, $\mathrm{Expert}_i$ is the $i$-th expert network, and $G_i(x)$ is the gating function output, often implemented as a softmax over the experts [169].

### *2.2.2 Phases of LLM Inference*

This inference process of generative LLM unfolds in two phases: prompt processing, or *prefill*, and token generation, or *decode*. The *prefill* stage occurs first and is used to generate only the first token of the response. All subsequent tokens are generated in the *decode* stage. This iterative process continues until a specific end-of-sequence (EOS) token is generated or the sequence reaches a predefined maximum length. The key operations of an attention head in prefill and decode are shown in the right part in Figure 2.3.

During the *prefill* stage, we compute over the full context of the input prompt to generate the first new token. This often turns the computation bounded as we have dense computation over the full input sequence. We also keep the key and value projections inside each layer of the self-attention block into a data structure called the *key-value (KV) cache*.

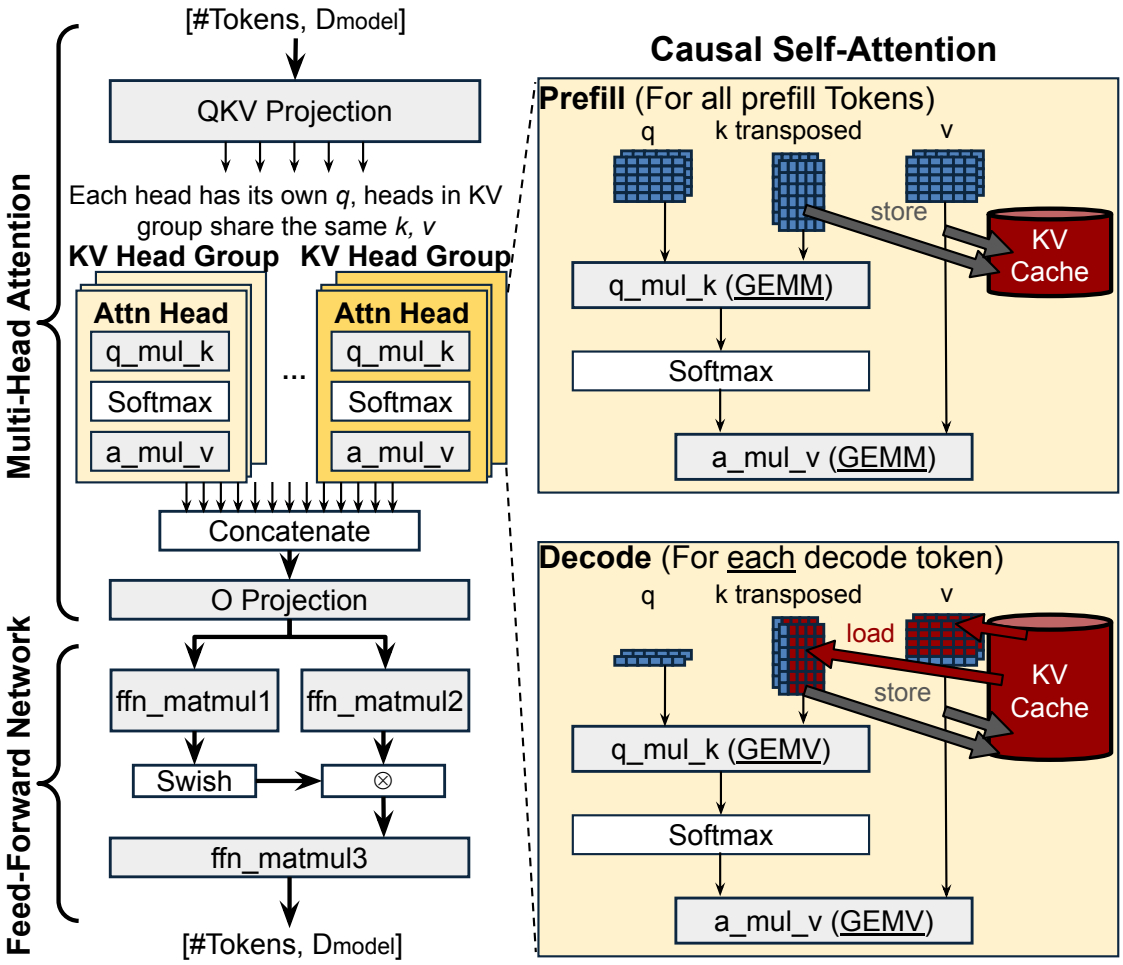



Figure 2.3: Key operations in a transformer block with grouped-query attention and SwiGLU activation function.

During the *decode* stage, only the last generated token is given as the input context . Inside the self-attention block, we need the KV projection for the entire context, thus the new token's KV projections are concatenated with the cached KV projection to attain the full context KV projection. This new token's KV projections are also written back to the cache for future tokens. Because the transformer decoder block is causal and masks backwards connections, the result using the KV cache is algebraically equivalent to recomputing the entire model on the full context.

The KV cache will significantly reduce the number of operations per token at the expense of maintaining the KV cache. It also makes the model significantly more memory bound. In the decode stage, as shown in lower right of Figure 2.3, each decode token has to do the self-attention independently. Therefore, two MatMul operations `q_mul_k` and `a_mul_v` become matrix-vector multiplications (GEMVs), which require a large KV cache load as the context length increases.

## *2.3 System Techniques for AI Inference*

### *2.3.1 Multi-Device Parallelism*

Large AI models such as LLMs require distributing operations across multiple devices (*mapping*) due to their computational and memory demands. This section covers five key multi-device parallelism paradigms. While Data Parallelism and Model Parallelism are broadly applicable to Deep Neural Networks (DNNs), Context Parallelism and Expert Parallelism are primarily employed in LLMs.

**Data Parallelism (DP)** [107, 117] replicates the entire model on each device, with each processing a subset of the input batch. DP is simple to implement but memory-intensive. It is commonly used in large batched DNN training to improve the training throughput, while the batch size of inference tasks can be very small, in which case data parallelism cannot take advantage of all nodes.

**Model Parallelism** [43] partitions the model itself, assigning parts to different devices. This reduces per-device memory but can increase inter-device communication. Depending on the partition dimension, there are **Tensor Model Parallelism (TP)** and **Pipeline**

**Model Parallelism (PP)** [78, 104, 213, 55]. TP partitions individual model layers across devices. Operations within layers are distributed, requiring frequent synchronization, such as partial sum accumulation and layer normalization. PP assigns blocks of layers to devices, and all devices operate in a pipelined fashion. Compared to TP, PP eliminates most of the inter-node communication.

**Context Parallelism (CP)** [118, 212] distributes the input sequence across multiple devices or processing units. Instead of processing an entire sequence on a single device, context parallelism divides the tokens so that different parts of the sequence are handled in parallel. CP helps reduce memory pressure and improves throughput, especially for long-context inputs, by enabling devices to work on separate portions of the sequence simultaneously while synchronizing only the necessary intermediate states.

**Expert Parallelism (EP)** [51], used in Mixture of Experts (MoE) models, distributes numerous specialized *expert* subnetworks across devices. EP scales model capacity while keeping per-token computation manageable, but faces challenges in load balancing and efficient token routing.

These parallelism strategies are often combined, and the choice of which to use, or how to combine them, is critical for achieving optimal performance and efficiency in large-scale AI inference.

### *2.3.2 Dynamic Batching and Scheduling for LLM Serving*

Efficiently serving LLMs requires sophisticated batching and scheduling techniques to maximize throughput and resource utilization while maintaining acceptable latency for users. Batching, the process of grouping multiple requests for simultaneous processing, is particularly critical for improving throughput, especially for inference tasks like token generation during decoding, which often have a low arithmetic intensity. However, naive batching strategies applied to interactive requests, such as those in chat applications, can introduce unacceptable end-to-end latency. Dynamic batching techniques aim to strike a balance between these competing objectives.

Several batching strategies have been developed to address different aspects of the latency-

throughput trade-off in LLM serving:

**Standard Batching** (or Static Batching) is the traditional approach where incoming requests are grouped into batches, often of a fixed size, before execution. A key drawback is potential underutilization: all requests in a batch are typically processed until the longest sequence completes, often involving padding shorter sequences. This means compute resources can be idle while waiting for the entire batch to finish, especially when sequence lengths vary significantly.

**Continuous Batching (CB)** [216] improves upon static batching by processing a dynamic batch of requests. In each iteration, the system processes all active requests in the current batch. New requests can be added to the batch as they arrive and capacity permits, and completed requests are removed. Consequently, new requests do not have to wait for all preceding requests in a fixed batch to finish. This significantly improves system utilization and reduces average latency, especially for dynamic workloads with varying request rates and lengths.

**Mixed Continuous Batching (MCB)** [74] further refines CB by enabling the prefill phase of new requests to be processed concurrently with the decode phase of ongoing requests within the same iteration, as illustrated in the top plot of Figure 2.4. This strategy efficiently interleaves the compute-intensive prefill operations with the memory-bandwidth-bound decode operations. MCB enhances hardware utilization and reduces queueing delays, proving particularly beneficial in interactive settings where responsiveness is key.

**Chunked Mixed Continuous Batching (C-MCB)** [4] addresses the challenge of large prefill requests, which can still significantly increase iteration times even in MCB. C-MCB mitigates this by dividing large prefill computations into smaller, more manageable chunks. This allows decode operations for active requests to be interleaved more finely with these prefill chunks, minimizing the latency impact of long prompts on other requests, as shown in the bottom plot of Figure 2.4. By carefully selecting chunk sizes, C-MCB can prioritize decode tasks, thereby improving overall system utilization and throughput. While chunking might imply additional KV-cache operations, this typically incurs negligible performance overhead because the prefill phase is generally compute-bound rather than memory-bound.

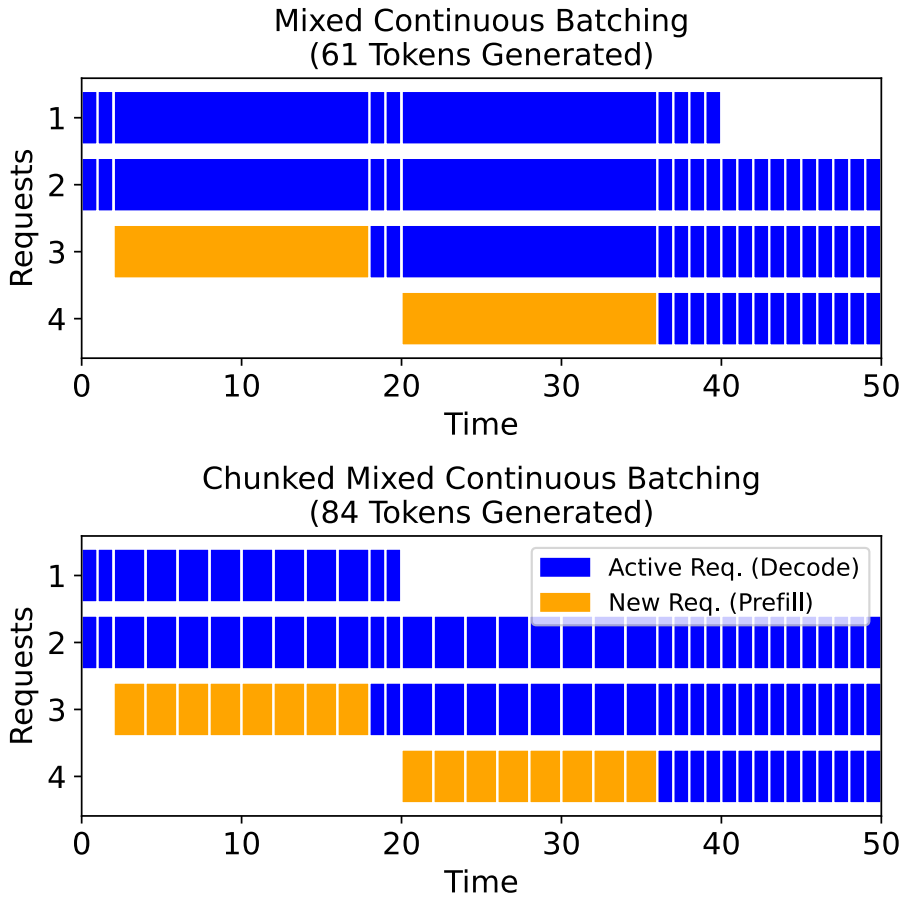



Figure 2.4: Comparison of mixed continuous batching (MCB) and chunked mixed continuous batching (C-MCB).

### *2.3.3 Hardware-Aware Model Co-design*

Beyond optimizing system-level scheduling and batching for existing models, a significant trend in improving AI inference efficiency involves hardware-aware model co-design. This paradigm no longer views AI models and hardware architectures as independent entities, but instead advocates a synergistic approach where model architectures, algorithms, and even training methodologies are developed with explicit consideration of the underlying hardware's capabilities and constraints [64]. This holistic strategy becomes increasingly important, especially for large-scale models like LLMs, where the interplay between model complexity and hardware limitations determines the overall performance, cost, and energy efficiency.

Recent large-scale LLM deployments provide compelling examples of this co-design concept. For example, the DeepSeek-V3 system demonstrates how hardware-aware model design choices can lead to more cost-efficient training and inference on existing GPU clusters [222]. Key aspects of such co-design include:

**Memory Efficiency through Model Architecture:** The enormous memory footprint of LLMs, particularly due to the Key-Value (KV) cache in Transformer attention mechanisms, is a primary target for co-design. Techniques like Multi-head Latent Attention (MLA), as employed in DeepSeek-V3, aim to reduce KV cache demands by compressing key-value representations, thereby improving memory efficiency and allowing for longer context lengths or larger batch sizes on memory-constrained hardware [47, 222]. Other model-side innovations like Grouped-Query Attention (GQA) [6] and Multi-Query Attention (MQA) [167] also represent efforts to make attention mechanisms more hardware-friendly from a memory perspective.

**Computational and Communication Co-optimization:** For models incorporating techniques like Mixture of Experts (MoE), which scale model capacity by selectively activating sub-networks (experts), co-design is crucial. The routing of tokens to experts and the communication patterns for aggregating expert outputs must be optimized in conjunction with the hardware's interconnect topology and parallelism capabilities. DeepSeek-V3 [222] discusses optimizing their MoE architecture for the specific NVIDIA H800 GPU cluster, con-

sidering computation-communication trade-offs and employing techniques like node-limited routing.

**Leveraging Low-Precision Arithmetic:** Modern AI hardware often includes specialized units for lower-precision arithmetic (e.g., FP8, INT8) to improve throughput and reduce memory bandwidth. Hardware-aware model co-design involves training or fine-tuning models to be robust to these lower precisions (e.g., through quantization-aware training or specific training recipes for formats like FP8). This ensures the model can effectively utilize the hardware's peak performance capabilities without significant accuracy degradation.

**Hardware-Aware Neural Architecture Search (NAS):** Another dimension of co-design involves incorporating hardware performance metrics (e.g., latency, energy) directly into the search process for optimal neural network architectures [49].This ensures that the discovered architectures are not only accurate but also efficient on the target hardware platform.

The insights from such large-scale deployments and research efforts highlight that achieving optimal efficiency for demanding AI workloads, particularly LLMs, increasingly requires a departure from a siloed approach to model and hardware development. Instead, a deeply integrated co-design process, where algorithmic innovations evolve in tandem with architectural features, is becoming the norm to push the frontiers of AI system performance and scalability.

## *2.4 Specialized Hardware Acceleration*

The computational demands of state-of-the-art AI models, especially Large Language Models (LLMs), have exceed the capabilities of general-purpose processors like CPUs and GPUs. This has created an urgent need for specialized hardware accelerators to deliver the required performance and energy efficiency.

### *2.4.1 Motivation for ASICs*

General-purpose CPUs are designed for a wide range of tasks and offer great flexibility, but their architecture is not inherently optimized for the highly parallel, matrix-multiply-intensive operations that dominate AI computations. While GPUs, originally designed for

parallel graphics rendering, have been successfully repurposed for AI workloads due to their massively parallel architecture, they still retain some general-purpose overheads and may not always provide the optimal balance of performance, power, and cost for specific AI tasks at scale.

Application-Specific Integrated Circuits (ASICs) offer a solution by providing hardware tailored to the computational patterns of AI algorithms [87, 27, 41]. By designing circuits from the ground up for tasks like convolution, matrix multiplication, and activation functions, ASICs can achieve the following:

**Higher Performance:** Dedicated data paths and massive parallelism allow ASICs to perform AI computations significantly faster than general-purpose hardware.

**Greater Energy Efficiency:** By eliminating unnecessary general-purpose logic and optimizing data movement, ASICs consume less power for the same AI workload, a critical factor for both edge devices and large-scale datacenter deployments.

**Lower Cost at Scale:** While the initial design cost (Non-Recurring Engineering, or NRE) of ASICs is high, for high-volume applications, the per-unit manufacturing cost can be lower than that of complex general-purpose chips, leading to better Total Cost of Ownership (TCO).

The rise of LLMs, with their enormous parameter counts and computational needs, has further intensified the drive towards AI ASICs. The sheer scale of computation required to train and serve these models makes efficiency paramount, and ASICs provide a pathway to achieve this at the scale demanded by modern AI applications.

### *2.4.2 Principles of Hardware Acceleration for AI*

Effective hardware acceleration of AI depends on several key architectural principles and optimization techniques:

**Specialized Compute Units:** Instead of general-purpose ALUs, AI accelerators feature dedicated units for operations like MAC (Multiply-Accumulate), vector processing, and sometimes specific activation functions. Tensor cores with systolic arrays, which perform matrix-matrix multiplications efficiently, are a prime example [109, 87, 132].

**Dataflow Architectures:** The way data (activations, weights, partial sums) is moved and processed within the accelerator is critical. Different dataflow strategies (e.g., weight stationary, output stationary, input stationary, row stationary as discussed in Section 3.1 or more flexible schemes like $F|B|C$ as discussed in Chapter 3.2 for DRLP) aim to maximize data reuse and minimize energy-intensive data movement to and from off-chip memory. Systolic arrays are a common architectural pattern that implements specific dataflows efficiently [28, 29, 112].

**Memory System Optimization:** Traditionally, this involves memory hierarchies with large on-chip SRAM (caches, scratchpads, buffers) to keep frequently accessed data (weights, activations, LLM KV cache) near compute units, reducing costly off-chip data movement (e.g., DRAM or HBM) [87, 29]. Emerging paradigms further minimize data transfer. Near-memory processing moves compute closer to memory (e.g., in 3D stacks). In-memory computing or processing-in-memory (PIM) performs computations directly within memory arrays (e.g., using ReRAM or specialized SRAM/DRAM) [5, 166, 165].

**Reduced Precision and Quantization:** Many DNNs can tolerate computations at lower numerical precision (e.g., 16-bit floating point (FP16/BF16), 8-bit integers (INT8), or even lower) with minimal loss in accuracy. Using reduced precision reduces memory footprint, memory bandwidth requirements, and the energy/area of compute units [68, 86].

**Sparsity Exploitation:** DNN weights and activations can often be sparse (containing many zero values). Hardware that can skip computations involving zeros or store sparse data in compressed formats can significantly improve performance and efficiency [68].

The effectiveness of AI accelerators is evaluated based on several metrics:

- **Throughput:** Operations per second (e.g., TOPS, FLOPS) or inferences or tokens per second.

- **Latency:** Time taken to complete a single inference or generate a token.

- **Energy Efficiency:** Operations throughput normalized by power (e.g., TOPS/W), or energy per operation (e.g., Joule/Inference, Joule/Token).

- **Area Efficiency:** Operations per unit of silicon area (e.g., TOPS/$mm^2$).

- **Total Cost of Ownership (TCO):** Includes capital expenditure (chip cost, server cost) and operational expenditure (power, cooling) over the system's lifetime. For cloud providers and large-scale deployments, TCO per unit of performance (e.g., TCO/Token) is often the ultimate metric.

### *2.4.3 Challenges in AI Accelerator Design*

Despite the potential benefits, designing efficient and effective AI accelerators presents numerous challenges [176, 157].

**The Memory Wall:** As compute capabilities increase, providing sufficient memory bandwidth and capacity to keep the PEs fed with data becomes a primary bottleneck. This is particularly acute for memory-bound operations common in LLMs, such as the attention mechanism and large embedding table lookups.

**Scalability:** AI models continue to grow in size and complexity. Designing architectures that can scale efficiently to accommodate future models, both within a single chip and across multiple chips/servers, is a major hurdle. This involves scalable interconnects, memory systems, and programming models.

**Flexibility vs. Efficiency Trade-off:** Highly specialized ASICs can achieve peak efficiency for a specific model or task but may perform poorly on others. As AI algorithms rapidly evolve, designing accelerators that offer a good balance between efficiency and programmability/flexibility to support a range of current and future models is crucial.

**Power Delivery and Thermal Management:** High-performance accelerators can consume significant power, leading to challenges in power delivery networks and heat dissipation, especially in dense server environments or power-constrained edge devices.

**Hardware-Software Co-design:** Achieving optimal performance requires close interaction between hardware architecture and software (compilers, runtime systems, mapping strategies). Compilers must efficiently map complex AI models onto the parallel hardware, optimizing data movement and resource utilization [40].

**Cost and Non-Recurring Engineering (NRE):** The design and verification of an

ASIC involve substantial upfront NRE costs, including expenses for design tools, engineering effort, and silicon mask sets. For an ASIC to be economically viable, these NRE costs must be amortized over a sufficiently large volume of chips or justified by significant TCO savings compared to alternative solutions [98, 123].

### *2.4.4 The NRE Challenge and Amortization in the Era of LLMs*

The high NRE costs associated with ASIC development have traditionally been a barrier to their widespread adoption, especially for applications with uncertain market sizes or rapidly changing algorithms. However, the landscape is shifting with the advent of large-scale AI deployments, particularly for LLMs.

The immense demand for LLM inference capabilities, driven by applications like chatbots, content generation, and AI-powered search, translates into a massive volume of computations. As discussed in [145], the operational TCO of running these workloads on existing general-purpose hardware (like GPUs) at the required scale can be extraordinarily high. For instance, serving applications like ChatGPT or integrating LLMs into high-volume web search implies a TCO that can run into hundreds of millions or even billions of dollars annually. In such high-demand scenarios, the substantial TCO savings offered by an efficient, specialized ASIC can quickly outweigh its initial NRE. Even if an ASIC offers only a modest percentage improvement in TCO/Token compared to GPUs, the sheer volume of tokens generated means that the cumulative TCO savings can amortize the NRE relatively quickly.

Figure 2.5 shows the minimum required TCO/Token improvement in order to justify the NRE. We extend the NRE model from Moonwalk [98] to use a 7nm technology node and estimate the NRE of an ASIC accelerator for large language models to be approximately $35M, including silicon mask cost, CAD tools, IP licensing, flip-chip BGA packing, server designs, and labor. Even if it were $100M, the current cost of running workloads like ChatGPT and web search with integrated LLMs is so massive that it not only justifies the cost of creating ASIC supercomputers but going even further as to co-optimize those supercomputers for specifics LLMs for additional improvement in TCO per token.

Therefore, while NRE remains a significant factor, the unprecedented scale of LLM de-

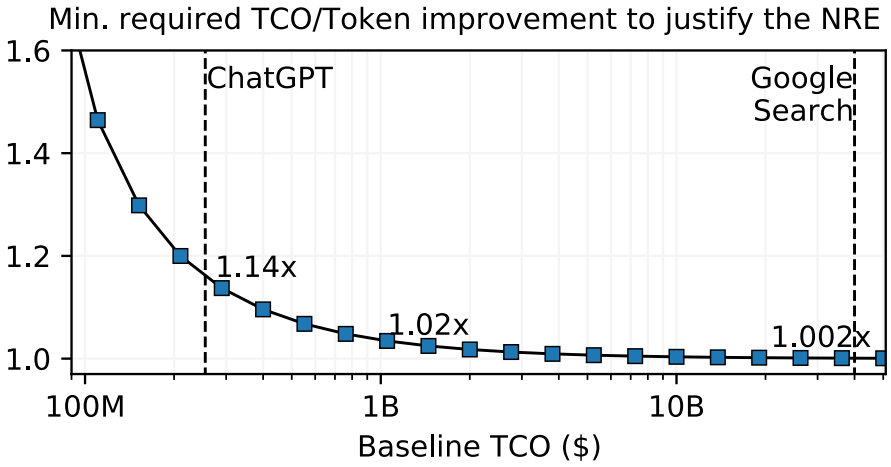



Figure 2.5: Minimum TCO/Token improvement required from an ASIC to justify NRE costs as a function of baseline TCO on existing hardware.

ployment is creating a scenario where the economic benefits of specialized ASICs can overcome this initial investment, paving the way for more customized and efficient AI hardware.

# Chapter 3

# FOUNDATIONAL WORK: ACCELERATING EARLY AI PARADIGMS

The rapid evolution of artificial intelligence has been characterized by increasingly complex neural network models. Before the recent dominance of Large Language Models (LLMs), significant research focused on developing efficient hardware for earlier paradigms like Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), and Deep Reinforcement Learning (DRL). These foundational models, while significantly smaller in scale than modern LLMs, presented their own unique challenges for hardware designers, including diverse computational patterns, memory access bottlenecks, and the need for flexibility. This chapter details foundational research undertaken during the author's doctoral studies, exploring architectures designed to accelerate these earlier AI workloads. These explorations provided critical insights and experience that informed the subsequent focus on LLM acceleration. Specifically, we will discuss iFPNA, a flexible processor targeting diverse early models, and DRLP, a specialized accelerator tackling the unique demands of Deep Q-Learning, including its emulation on cloud FPGAs.

## *3.1 iFPNA: A Flexible and Efficient Deep Learning Processor in 28nm CMOS*

The discussion, figures, and tables related to the iFPNA architecture presented in this section are based on and reprinted from our prior work published in [26].

As deep learning algorithms evolved rapidly, the need for hardware that could adapt to various network types (CNNs, RNNs, Fully Connected networks) became apparent. Fixed data flow schemes in early processors limited their coverage of different algorithms, motivating the development of more flexible solutions.

The instruction and Fabric Programmable Neuron Array (iFPNA) architecture was developed collaboratively to address this need [26]. Presented as a 28nm CMOS chip prototype, iFPNA aimed to accelerate a variety of DNNs efficiently on a single platform. The iFPNA

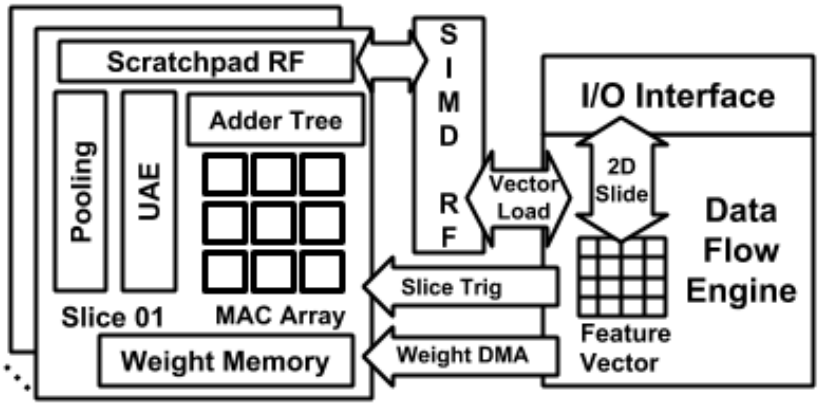



Figure 3.1: The iFPNA architecture.

architecture combines instruction-level programmability as in an Instruction Set Architecture (ISA) with logic-level reconfigurability as in a Field-Programmable Gate Array (FPGA) in a sliced structure for scalability. Four data flow models, namely weight stationary, input stationary, row stationary and tunnel stationary, are described as the abstraction of various DNN data and computational dependence. The iFPNA compiler partitions a large-size DNN to smaller networks, each being mapped to, optimized and code generated for, the underlying iFPNA processor using one or a mixture of the four data-flow models. Experimental results have shown that various CNNs, RNNs, and FC networks can be mapped to the iFPNA processor, achieving the near ASIC performance.

*Architecture Overview*

Figure 3.1 illustrates the overall iFPNA accelerator architecture. It features a programmable data flow engine for executing instructions and 16 neuron slices for performing data-intensive computing in parallel. Each slice consists of a reconfigurable multiply-and-accumulate (MAC) array, a programmable adder tree, a universal activation engine (UAE), a pooling block, local scratchpad registers, and a weight memory block.

Each MAC array is highly configurable, supporting multiple vector/bit modes: specifically, it can be configured as 9×16-bit, 16×8-bit, 25×6-bit, or 36×4-bit MAC units by

| Instruction (Opcode) | Operand I | Operand II | Operand III |
|---|---|---|---|
| Weight Load (WL) | Length | Src. Addr. | Dst. Addr. |
| Vector Collect (VC) | Src. Addr. | Dst. Reg. | Sld./Sps. |
| Computing Execute (EX) | Mode | Dst. Addr. | |

Table 3.1: iFPNA Instruction Set.

programming its constituent 27×8-bit and 9×4-bit multipliers and adder trees. All slices share a feature vector input generated by the data flow engine and computing instructions but possess their own weight memory. To support data flows like row stationary, slices can accumulate partial sums from neighboring slices.

A register file (RF) bridges the data flow engine and the slices, containing three types of registers: an 8×8-bit configuration register array, an 8×8-bit general-purpose register array (for addresses, etc.), and a 4×128-bit SIMD register array for collecting/feeding data from/to slices in parallel. Off-chip memory access utilizes a 16-bit custom I/O interface.

*Programmability*

iFPNA features multi-level programmability:

**Instruction Set:** The central controller uses an ISA extended with DNN-specific instructions (Table 3.1) like *Weight Load* (for DMA), *Vector Collect* (for loading features into SIMD registers with options for sliding/sparse data), and *Computing Execute* (to trigger computation in the slices). This allowed different data reuse patterns and PE operations to be defined by compiled programs, offering more flexibility than fixed FSM controllers.

**Fabric Reconfigurability:** The MAC units supported various quantization modes (4 bit to 16 bit). The activation/pooling engine could be configured for different functions (ReLU, Sigmoid, Tanh, various pooling types, element-wise operations for LSTM/Batch Norm). Inter-slice communication fabric allowed for partial sum transfer between neighboring slices, enabling more complex data flows like Row Stationary.

*Data Flow Models*

To handle diverse computational patterns and data dependencies, iFPNA supported four distinct data flow models, selectable by the compiler based on network layer characteristics and hardware constraints:

- **Weight Stationary (WS):** Weights are held stationary in slice memory, and input features slide across them (Figure 3.2). Simple but potentially high latency due to repeated feature loading.

- **Input Stationary (IS):** Input features are held stationary in registers/scratchpads, while different weights are shuffled through the PEs (Figure 3.3). Effective for layers with many kernels, reducing feature load latency.

- **Row Stationary (RS):** Exploits convolutional reuse by holding rows of weights stationary and streaming input rows, accumulating partial sums diagonally across slices using inter-slice communication (Figure 3.4). Reduces movement of all data types but requires more scratchpad space and specific PE array configurations.

- **Tunnel Stationary (TS):** An enhancement over RS, fragmenting weights differently (like $1 \times 1 \times 9$ tunnels) to hold smaller input feature segments stationary, reducing input scratchpad needs and improving PE utilization without requiring inter-slice communication for accumulation (Figure 3.5).

*Evaluation*

A silicon prototype of the iFPNA processor has been designed and fabricated in a 28nm HPC technology. The prototype system, as shown in Figure 3.6, consists of a PC running the compiler, a PCIe link that transfers data from the PC to the chip, and an FPGA board to buffer the data transmitted to the chip. Table 3.1 shows the measured performance and comparison with Eyeriss for CNNS [28], and OCEAN for RNNS [24]. The iFPNA processor,

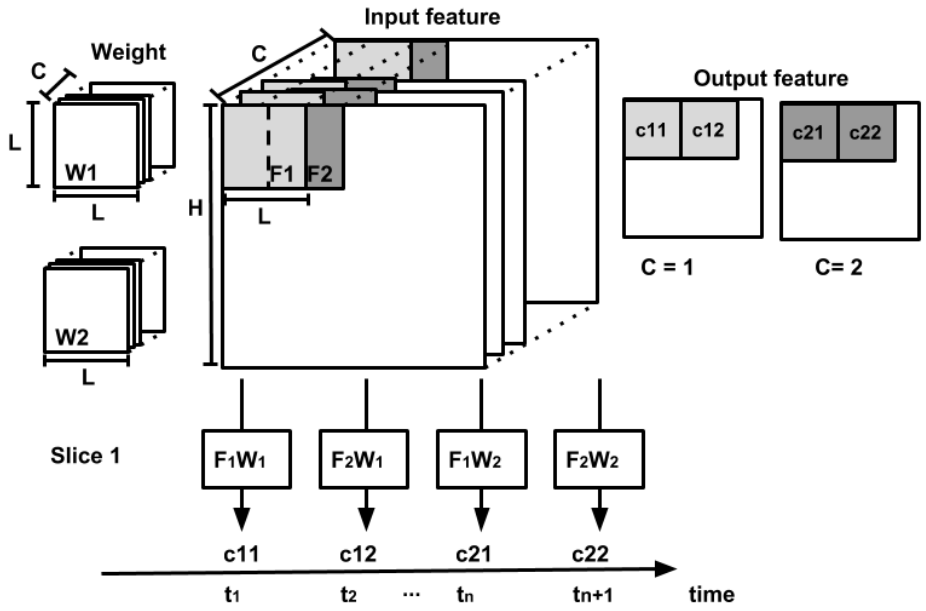



Figure 3.2: Weight Stationary on the iFPNA architecture.

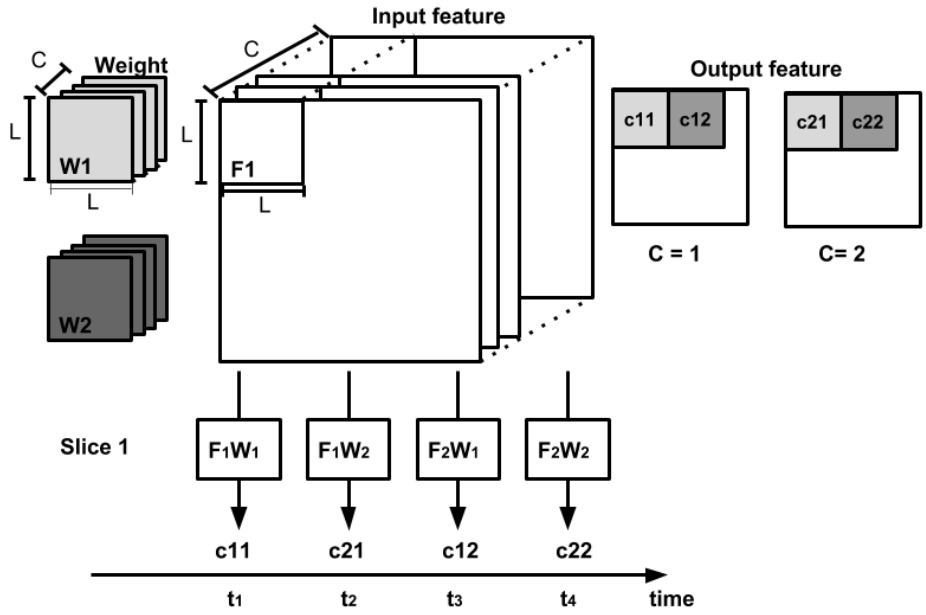



Figure 3.3: Input Stationary on the iFPNA architecture.

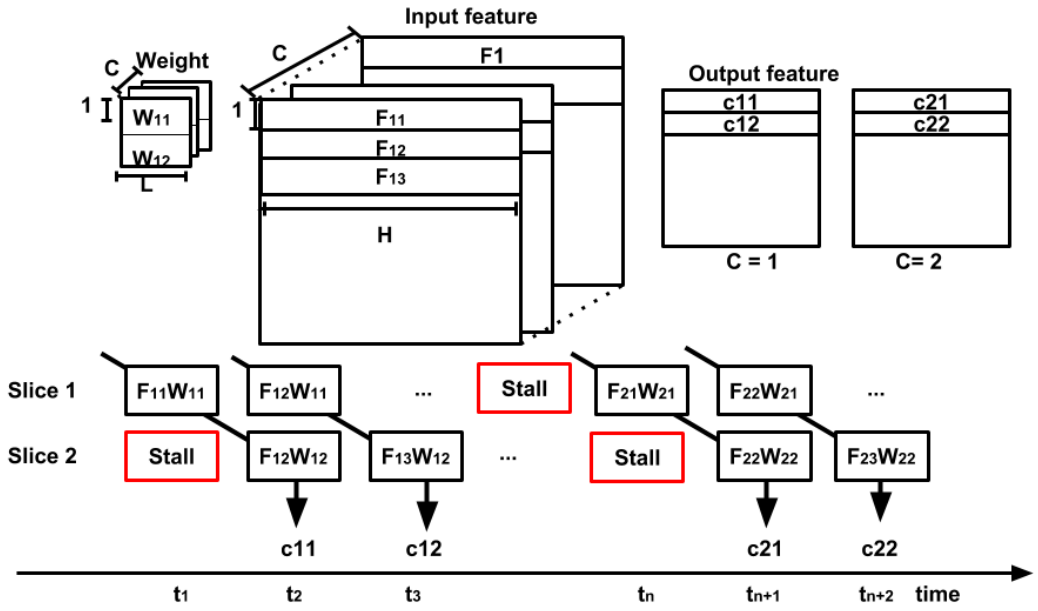



Figure 3.4: Row Stationary on the iFPNA architecture.

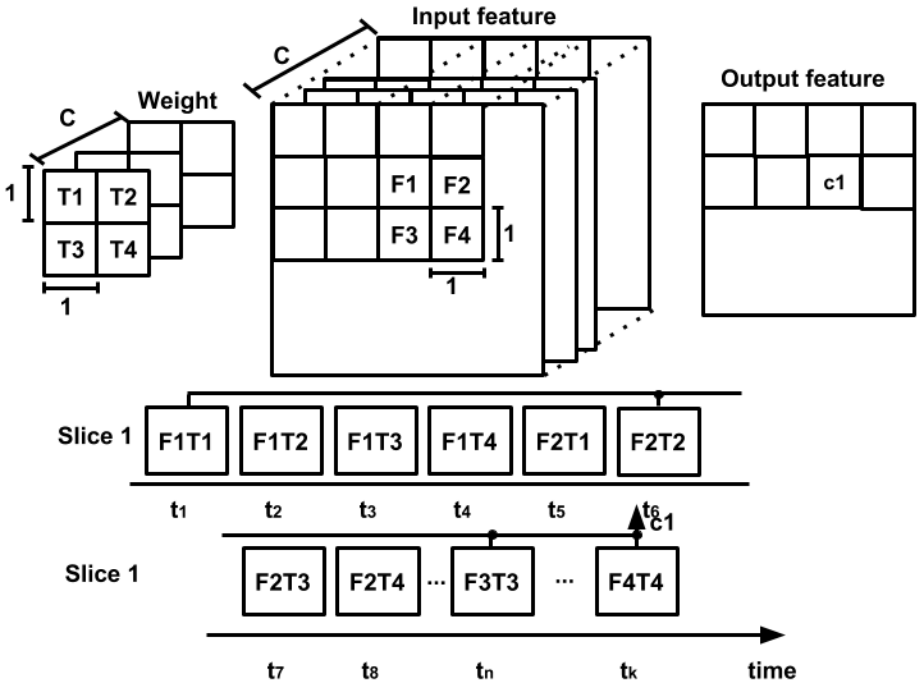



Figure 3.5: Tunnel Stationary on the iFPNA architecture.

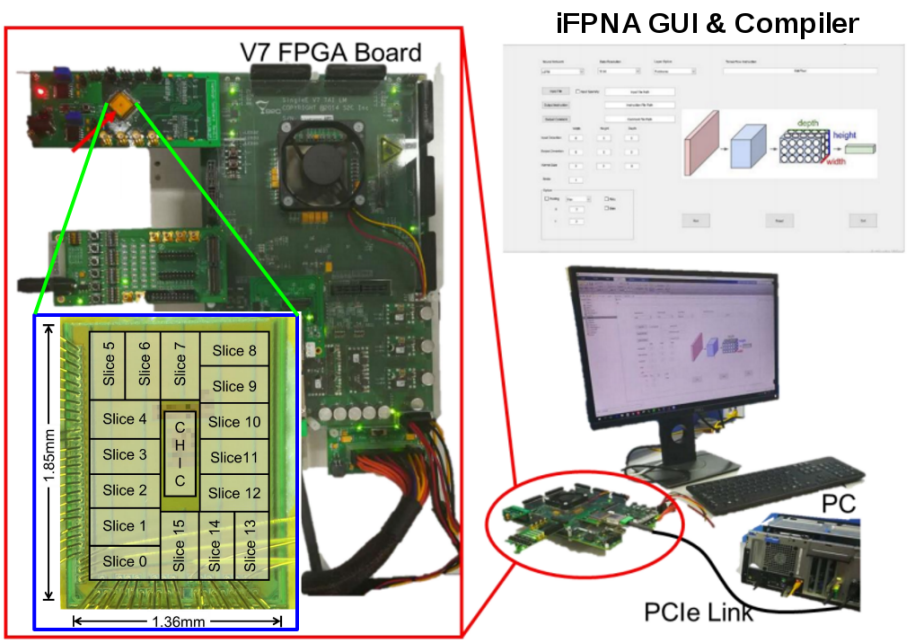



Figure 3.6: iFPNA chip prototype and demonstration system.

while being highly flexible and programmable, achieves performance comparable to dedicated CNN/RNN accelerators.

The experience with iFPNA highlighted the importance of architectural flexibility, adaptable dataflows, and compiler support in efficiently mapping evolving AI workloads onto hardware.

## *3.2 F|B|C: Optimized Deep Q-Learning with the Filter-Batch-Channel Dataflow*

Deep Reinforcement Learning (DRL), particularly value-based methods like Deep Q-Learning (DQN) and its successors (e.g., Rainbow [72]), represents another crucial AI paradigm. DRL agents learn optimal policies through interaction with an environment, making them suitable for applications requiring real-time decision-making, such as game playing and robotics. However, accelerating DRL presents unique challenges distinct from supervised learning.

| | Eyeriss[28] | OCEAN[24] | iFPNA |
|---|---|---|---|
| Technology | 65nm LP I | 65nm | 28nm HPC |
| VDD (V) | 0.82-1.17 | 0.8-1.2 | 0.65-0.9 |
| Clock (MHz) | 100-200 | 20-400 | 20-200 |
| Application | CNN | RNN | CNN/RNN |
| Data flow | RS | / | WS/IS/RS/TS |
| Bit width | 16b | 16b | 4b-16b |
| Power | 278 mW | 6.6-155.8 mW | 33.3 mW |
| Peak Performance | 56 GOPS | 311.6 GOPS | 53.4 GOPS |
| Peak Energy Efficiency | 0.35 TOPS/W | 2.0 TOP/W | 1.6 TOPS/W |

Table 3.2: iFPNA measured performance and comparison.

### *3.2.1 Motivation and Challenges in DRL Acceleration*

Similar to conventional DNNs, DRL needs many training samples to train the DNN parameters. The main difference is that DRL is unsupervised learning. It generates the datasets for training while the training itself is happening. Simply scaling up performance by using massive numbers of GPUs for a large number of training data samples is often not possible in DRL training because of the evolving environment. To generate the training data and target outputs, there are often numerous network inferences with different inputs and parameters involved.

We profile the end-to-end latency of one popular Q-learning algorithm Rainbow [72] on a NVIDIA T4 GPU with different training batch sizes and different number of DNN layers. As shown in Figure 3.7, DNN inferences (FP) account for up to 47.6% of the total end-to-end latency, followed by the backward propagation, weight gradient computation, and replay memory sampling. Thus, an accelerator with the optimization only on inference or training will not be sufficient for an end-to-end DRL system. It is necessary to carry out software/hardware co-optimization from a higher level.

Figure 3.8 shows a study of popular DRL algorithms. In contrast to DNNs, DRL typically

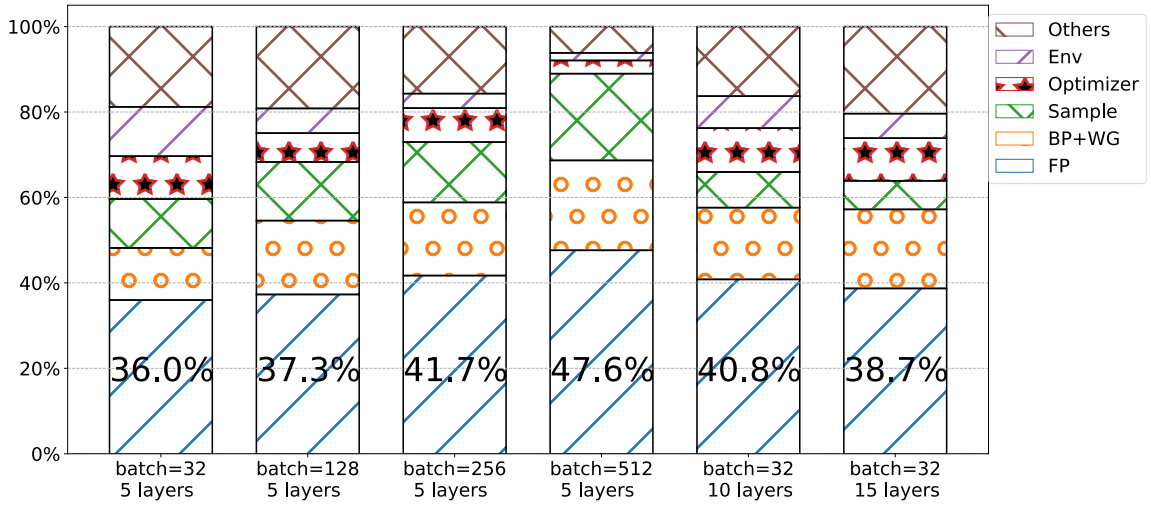



Figure 3.7: End-to-end of Rainbow DRL latency breakdown on a GPU. DNN inferences (FP) account for 36.0% to 47.6% of latency.

take millions of training iterations, while its model size is often much smaller, and the dimensions of layers vary greatly. For hardware designers, the various shapes of DNN layers are challenging because they affect the key design point of DNN accelerators: *dataflow*. Many existing DNN dataflow works [111, 214, 77] aim to optimize a single layer. They rely on a set of pre-selected data dimensions to exploit parallelism across an array of processing elements (PEs), which usually works fine for DNNs where most layers has the similar shape, such as ResNet-50 [71]. However, when the set of pre-selected data dimensions change across layers, the array utilization may decrease (i.e., fewer PEs are used), resulting in a decrease in both performance and energy efficiency. Furthermore, these works lack consideration of actual physical implementation. Some dataflows require high data bandwidth of the network-on-chip (NoC) to keep the PEs fully utilized, and custom-designed datapaths for inputs and outputs data, leading to a high design and control overhead. As a result, these dataflows and architecture designs are often difficult to scale, which makes it difficult to accelerate future large-scale DRLs (more actors/learners, large training batch size, potential large networks, etc). Therefore, a dataflow that can maintain its advantages across all layers of DRL training and an accelerator architecture with good scalability are crucial.

| DRL Algorithms | DQN | A2C | PPO | DDPG |
|---|---|---|---|---|
| # Parameter | 1.69M | 0.82M | 10.00K | 30.39K |
| Training Iteration | 10M~50M | 2M~10M | 0.5M~1M | 1M~2.5M |

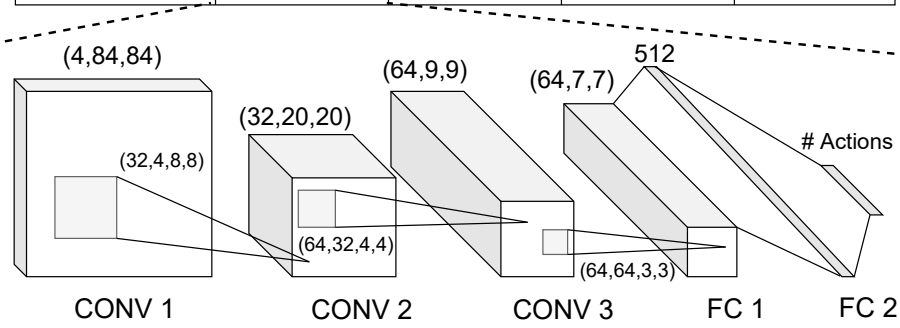



Figure 3.8: A study of popular DRL algorithms. With the large number of training iterations, DRL often uses a shallow DNN and the layer shape varies greatly.

Prior research in accelerator architectures for deep reinforcement learning is relatively uncommon. FA3C [32] is an FPGA-based DRL platform for a certain class of algorithms, Asynchronous Advantage Actor-Critic (A3C) [127]. It exploits standard DNN hardware acceleration techniques, like the dynamic data layout optimized for inference and training of the neural networks in A3C. Implemented on FPGAs, FA3C achieves 1.28× better performance and 1.62× better energy efficiency than using GPUs. We focus on Q-learning based DRL training such as Google DeepMind's Rainbow [72] and R2D2 [94], which enjoys several desirable properties over A3C. Since A3C is asynchronous, the training data is generated in real time, so the training batch size is limited, usually set as 5. With the help of the experience replay mechanism [125, 139], training data in Q-learning DRL comes from a large replay memory, and the batch size can be as large as 32 in most cases. Compared with policy-based methods, Q-learning methods are more sample efficient and achieve state-of-the-art performance on the Atari benchmarks [94]. To the best of our knowledge, our work is the first work that aims to accelerate Q-learning-based DRL training.

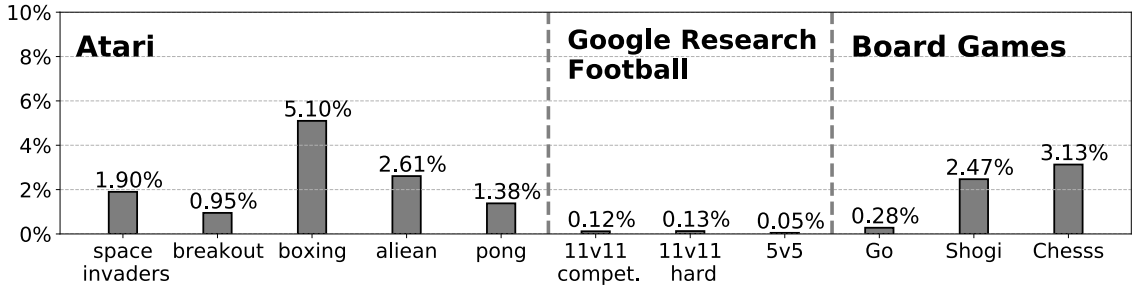



Figure 3.9: Change rate of adjacent frames.

### *3.2.2 Hardware/Software Co-Optimization Strategies*

To tackle the unique performance hurdles of Deep Q-Learning, two key hardware-software co-optimization strategies were developed.

*Exploiting State Similarity*

In many simulation-based or game environments, consecutive states $(S_t, S_{t+1})$ exhibit high temporal locality. To better quantify the similarity between states (frames), we analyze the difference of neighboring frames from 5 different Atari 2600 games, 3 scenarios from Google Research Football [110], and 3 popular board games, as shown in Figure 3.9. For the Atari games, we observe that only around 0.95% to 5.15% of pixels are changed per frame.

Inspired by the observation, we propose a method to **use the similarity between states to reduce the computation of FP in training.** In current implementations, $q_\theta(S_i)$ and $q_\theta(S_{i+1})$ are executed sequentially as two forward propagation. However, there are a lot of repeat operations in those two FPs, because their inputs are similar. Since the DNN topology is known, given the index of different elements of two input tensors, one can easily calculate the different indexes of output of each layer. If we did one NN FP on one of the input tensor, to get the results of another input tensor, we just need to recalculate those different indexes of each layer.

Figure 3.10 shows the example of a CONV layer with a $2 \times 2$ filter and the stride is 1. If two inputs have different element at idex $(x = 2, y = 1)$, according to the operation

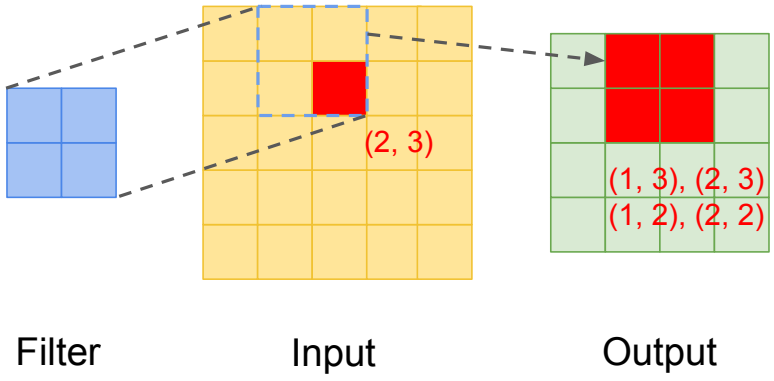



Figure 3.10: The different output indexes can be derived based on different input indexes.

pattern of a CONV layer, we can easily find out the different output elements, which are located at $(x = 1, y = 3), (x = 2, y = 3), (x = 1, y = 2)$, and $(x = 2, y = 2)$. The input different element involves the calculation of all of them. Note that different output channels have the same different indexes since they come from the same inputs. Using this method, when adding a new frame $f_{i+1}$ to the replay memory, we will also add a frame difference $\Delta f_{i+1}$. $\Delta f_{i+1}$ includes a set of tuple $(index, f_{i+1}[index])$ that records the location and value of different elements in frame $f_{i+1}$ compared with the previous one $f_i$.

*Target Pre-sampling*

The target network $q_{\overline{\theta}}$ in DQN, as shown in Equation 2.4, is a periodic copy of the online network $q_\theta$. It helps stabilize learning but doubles the network parameter storage requirement on the accelerator. Given that the large replay memory (e.g., 1 million transitions) changes very slowly relative to the training frequency (e.g., only 0.8% change after 8k steps), the target pre-sampling technique was proposed. Instead of storing and using $q_{\overline{\theta}}$ directly, the system periodically (e.g., every $m$ steps) pre-computes the target values needed for the loss function ($q_\theta(S_{j+1})$ for a representative sample of next states $S_{j+1}$ likely to be drawn from the replay memory soon) using the **current** online network parameters $\theta$. These pre-computed target values are then used for the subsequent training steps until the next pre-sampling phase. This effectively eliminates the need to store or transfer the target network parameters $\overline{\theta}$ on

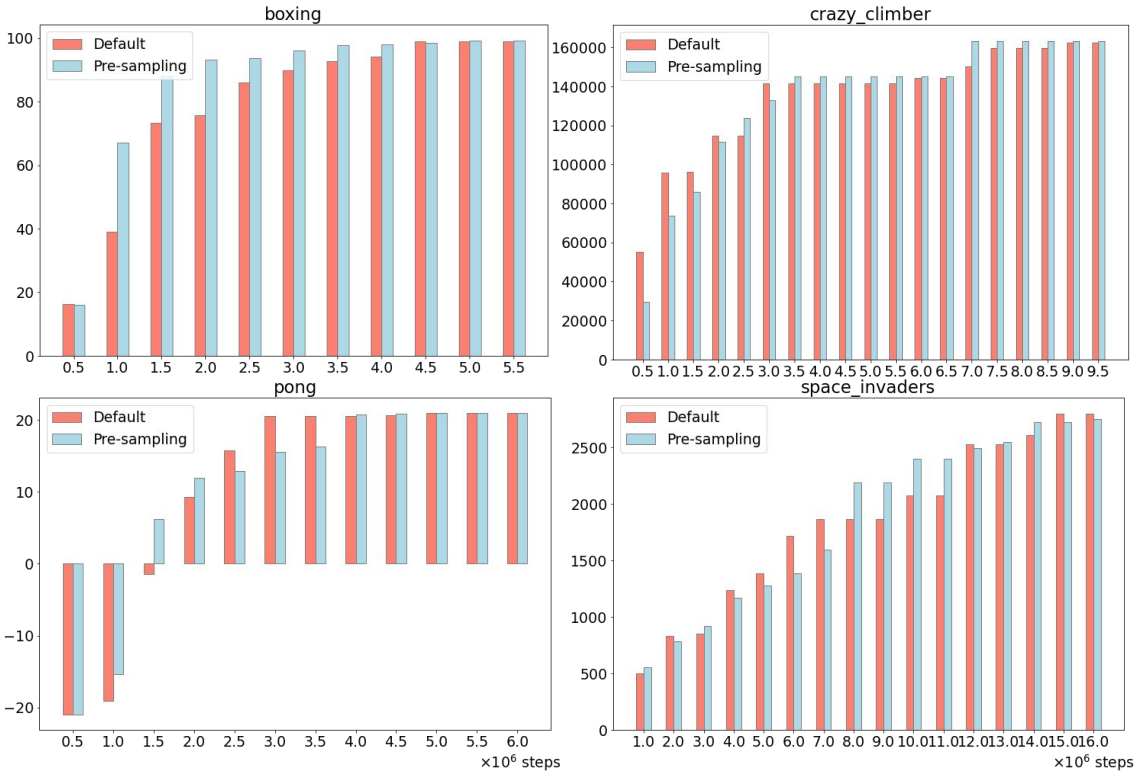



Figure 3.11: The training speed differences when applying different sampling scheme. X-axis represents training steps, Y-axis represents achieved scores during the training.

the accelerator, halving the on-chip parameter memory requirements.

Figure 3.11 compares the highest evaluation scores achieved of the pre-sampling enhanced Rainbow [72] with the original version on four different Atari tasks. Compare with the baseline, our method performs the same in pong, improving by 0.3% and 0.7% in boxing and crazy climber, respectively, and is 1.8% worse in space invaders. This demonstrates that the pre-sampling will not compromise training accuracy.

These co-optimizations target the major bottlenecks identified in DRL Q-learning: excessive forward passes and replay memory access, significantly improving end-to-end efficiency.

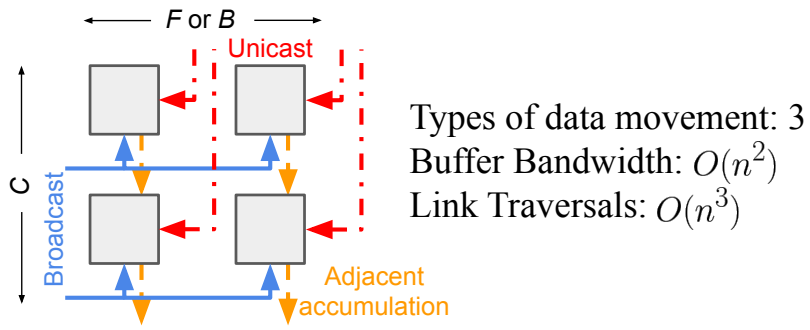



Figure 3.12: $F|C$ and $B|C$ dataflows require a variety of complex data movements, leading to high design and control overhead.

### *3.2.3 The $F|B|C$ Dataflow for Efficient DRL Training*

To address the challenge of low PE utilization caused by varying layer shapes in DRL networks and the high overhead of complex dataflows in some prior accelerators, a novel dataflow named Filter-Batch-Channel ($F|B|C$) was proposed. This notation was originally introduced in Interstellar [214] to represent which loops are spatially unrolled on a 2 dimensional processing elements (PE) array. Specifically, $U|V$ means that tensor dimension $U$ and $V$ are processed in parallel across the vertical and horizontal PEs. All tensor dimension notation refer to Figure 2.1.

Existing dataflows often parallelize across spatial dimensions or use complex schemes like $F|C$ or $B|C$. Spatial parallelization can lead to underutilization when output feature map dimensions $(E_x, E_y)$ or filter counts (F) don't match the PE array size. Complex schemes like $F|C$ or $B|C$ can require multiple types of data movement (unicast, broadcast, neighbor accumulation), leading to high NoC bandwidth requirements ($O(n^2)$ buffer bandwidth, $O(n^3)$ link traversals for an $n \times n$ array), complex control, and poor scalability, as shown in Figure 3.12.

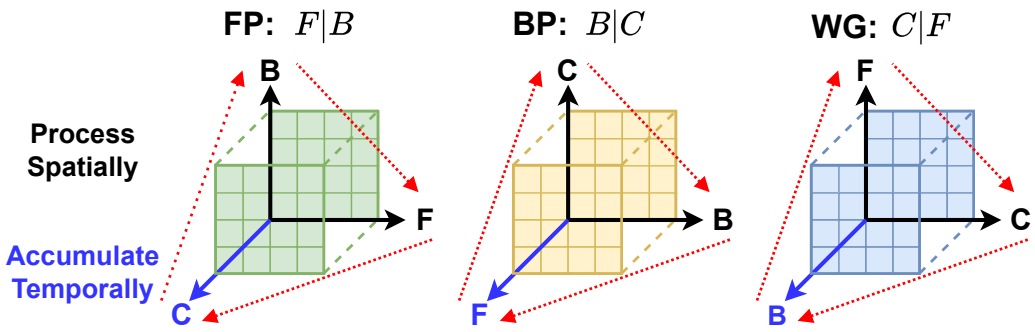



Figure 3.13: In F|B|C, at either phase of FP, BP, or WG, the two dimensions from $F$, $B$, and $C$ are processed in parallel, and the other dimension is accumulated temporally.

*$F|B|C$ Dataflow*

The $F|B|C$ dataflow exploits spatial parallelism across the dimensions that are typically more regular and often multiples of common values like 32 in DNN training: Filter count (F), Batch size (B), and input Channel count (C). The specific dimensions parallelized spatially versus accumulated temporally adapt to the phase of training to maximize data reuse and efficiency (Figure 3.13):

- **Forward Propagation (FP):** Employs $F|B$ spatial parallelism. Each PE row processes a different sample from the batch (B dimension), and each PE column processes a different output filter (F dimension). Input activations are broadcast vertically (reused across filters F), weights are broadcast horizontally (reused across batch samples B), and the computation across input channels (C dimension) is accumulated temporally within each PE.

- **Backward Propagation (BP):** Employs $B|C$ spatial parallelism. PE rows handle batch samples (B), columns handle input channels (C). Weights are reused within rows, output gradients within columns. Filter dimension (F) is accumulated temporally.

- **Weight Gradient (WG):** Employs $C|F$ spatial parallelism. PE rows handle input

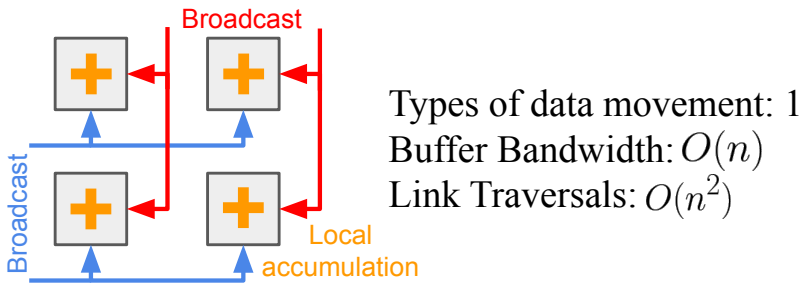



Figure 3.14: $F|B|C$ has less and simple data movements, leading to low design and control overhead, so it has good scalability.

channels (C), columns handle output filters (F). Output gradients are reused within rows, input activations within columns. Batch dimension (B) is accumulated temporally (integrating gradient aggregation, as shown in Figure 2.1).

The architecture design can be simplified and easy to scale by adopting our F|B|C. With F|B|C dataflow, partial sum output element can be accommodated onto the PE array. Thus, data locality for partial sum can be enhanced and all data reuse happens on-chip, significantly reducing the overall data movement. Additionally, according to our above discussion, our F|B|C dataflow require only one-dimensional broadcast. Compared with the three different types of data movement needed in previous works (unicast, 1D broadcast and the adjacent addition), our dataflow substantially decreases the design and control cost. For example, the peak buffer bandwidth needed for a $n \times n$ PE array is $O(n)$, and the link traversals of the dataflow is $O(n^2)$, as shown in Figure. 3.14.

In Figure 3.15, we compare $F|B|C$ with three common dataflows: $R_y|E_y$ (i.e., row-stationary) from [28], $F|C$ from [99], and $E_x|E_y$ from [33] on a $32 \times 32$ PE array. For dataflows cannot fill the whole array, we add a third-level parallelism dimension, which can utilize as many PEs as possible.

By parallelizing across F, B, and C, which are often larger and more regular than spatial dimensions, $F|B|C$ achieves near 100% PE utilization across different layer types and training phases. $F|B|C$ only requires efficient 1D broadcasts (row-wise or column-wise).

This drastically simplifies the NoC design compared to schemes needing unicast or complex routing (Figure 3.14 vs Figure 3.12). It reduces NoC traffic and energy consumption. It enables spatial reuse of both input tensors (activations/gradients and weights) in each phase, reducing memory access. It can flexibly prioritize activation or weight reuse temporally based on the phase, which is beneficial as DRL training involves significant activation data. The $F|B|C$ dataflow provides a robust foundation for building efficient and scalable DRL training accelerators.

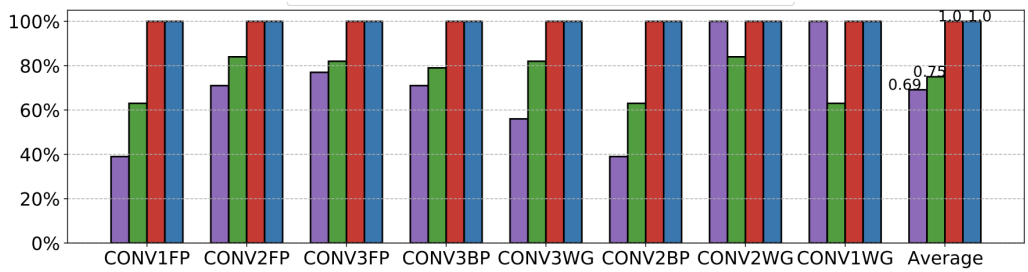
The principle of achieving hardware efficiency through flexible, compiler-aware mapping of tensor operations, rather than relying on fixed-size matrix multiplication units, is a growing area of investigation in AI accelerator design. For instance, recent work on general tensor contraction processors, such as the TCP architecture by Kim et al. [100], also emphasizes flexible hardware (e.g., PEs divisible into slices) and a compiler-driven approach to explore a broad space of *lowered shapes* and *tactics* for optimizing various tensor contractions, primarily demonstrated for LLM inference. While TCP targets a broader range of AI workloads and general tensor contractions with a more extensive compiler framework, the $F|B|C$ dataflow represents a specific, structured strategy for adapting parallelism in the context of DNN training phases.

### *3.2.4 DRLP: A Tiled Accelerator Architecture for $F|B|C$*

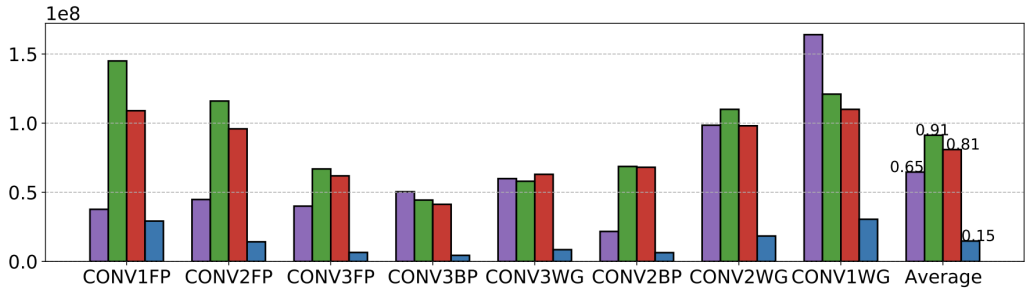
Leveraging the $F|B|C$ dataflow, the DRLP (Deep Reinforcement Learning Processor) architecture was designed with simplicity and scalability as primary goals.

DRLP utilizes a 2D tiled architecture, as shown in Figure 3.16. A $32\times32$ tile configuration was chosen based on typical DRL batch sizes (often 32 or multiples thereof). The tiles (PEs) are connected by a novel, efficient 1D broadcast Network-on-Chip (NoC). This NoC consists of two independent sets of links (horizontal and vertical) capable of broadcasting data from any tile to all other tiles in its row or column in a single hop, perfectly matching the requirements of $F|B|C$ without needing complex routing logic. IO modules are placed at the periphery for communication with the host CPU and off-chip DRAM.

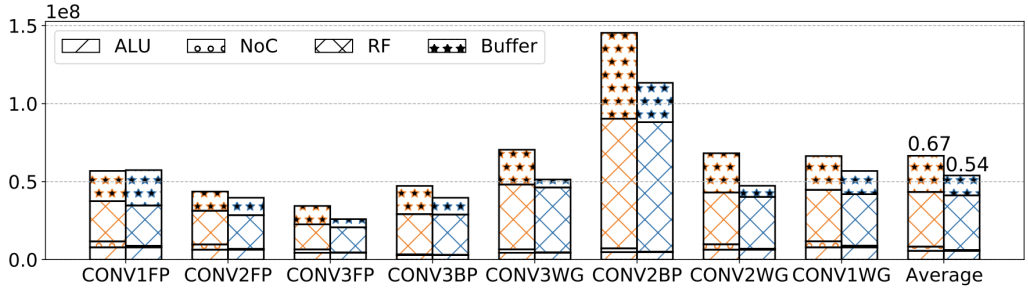
Each tile acts as a PE. It includes small, local register-file-based scratchpads (Input SPad,




(a) PE Array Utilization (%)




(b) Total NoC Traffics (Hops)




(c) Energy Breakdown (pJ) of $F|C$ (red) and $F|B|C$ (blue)

Figure 3.15: Compared with $R_y|E_y$, $F|C$, and $E_x|E_y$ on a $32 \times 32$ PE Array, $F|B|C$ achieves 100% utilization, needs less NoC traffics and costs less energy in most cases.

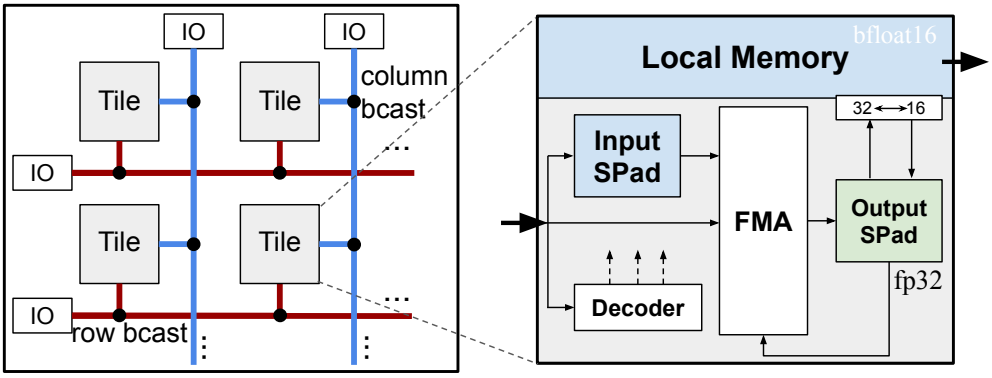



Figure 3.16: A high-level DRLP diagram and the tile with mixed-precision MACs. Tiles are connected by two sets of 1D broadcast NoC to enable the $F|B|C$ dataflow.

Output SPad) for buffering inputs and outputs, maximizing temporal data reuse within the tile. A local memory (larger than scratchpads, e.g., 1KB) stores partial sums during temporal accumulation (across $C$, $F$, or $B$ depending on the phase) and can buffer activations/weights for inter-layer or inter-phase reuse. The compute core is a mixed-precision MAC unit: multiplication uses 16-bit bfloat16 format (reducing area and energy for multipliers and memory storage), while accumulation uses 32-bit single-precision floating-point (fp32) to maintain accuracy during the sum reductions, which is critical for DRL training stability. Converters handle the bfloat16-to-fp32 transition where needed.

The NoC routers (Figure 3.17) implement simple routing rules: data entering from a tile is broadcast in the specified direction (horizontal or vertical); data entering from a direction continues in the opposite direction and is also delivered to the tile. This design eliminates the need for destination addresses, simplifying control and minimizing latency and traffic. We design the router based on a recent NoC design, Ruche Network [90], from the open-source hardware library BaseJump STL [192].

### *3.2.5 Evaluation*

To evaluate the improvements brought from our high-level co-optimization for deep Q-learning, we use 5 games from the Atari benchmark. To evaluate the dataflow and ar-

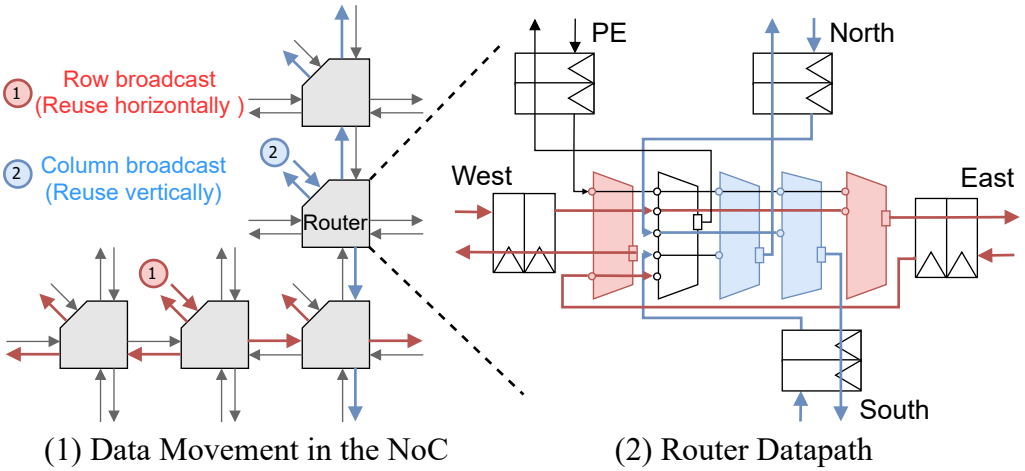



Figure 3.17: (1) In most cases, data read from a tile will be broadcast to the whole row (red arrows) to reuse horizontally or the whole column (blue arrows) to reuse vertically. (2) shows the datapath inside each 1D broadcast router.

chitecture, we use different DNNs from the canonical DQN [128], data-efficient DQN [199], and A3C [127], etc. We use Timeloop[141] to evaluate and compare the performance of the DRLP and other designs. The evaluation is based on the dataflow, and the accelerator architecture, etc. We compare a $32 \times 32$ DRLP against the following state-of-the-art DRL and DNN systems and accelerators: a FPGA-based accelerator for A3C (FA3C) [32], a multi GPUs-CPUs DRL system (iSwitch) [119], a recent DNN training accelerator [33] and two RL accelerators [99] and [206].

Compared to FA3C [32], a prior FPGA-based accelerator for A3C, the projected ASIC implementation of DRLP achieved a 43× speedup in inference throughput. Compared to iSwitch [119], a distributed multi-GPU system optimized for DRL training, DRLP achieved 677 training iterations per second, representing a 15× speedup. Figure 3.18 shows the breakdown that attributed these gains.

DRLP was also compared against optimistically scaled versions of a prior CNN training accelerator [33] and other DRL accelerators [206, 99]. Across various DQN and A3C network configurations, DRLP demonstrated speedups ranging from 1.14× to 2.18× (Figure 3.19),

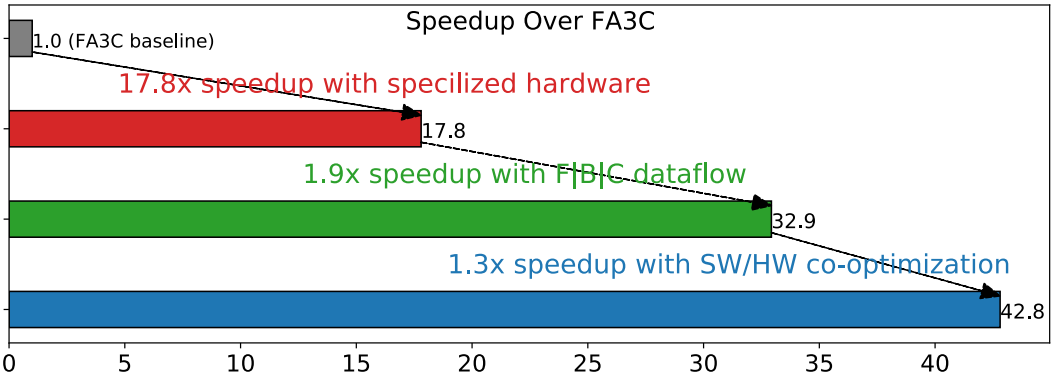



Figure 3.18: Speedup breakdown of DRLP over FA3C. $F|B|C$ dataflow and co-optimization bring 1.9× and 1.3× speedup, respectively.

underscoring the benefits of its tailored dataflow and co-design approach even against other specialized designs.

These results validate the effectiveness of the proposed co-optimizations, the $F|B|C$ dataflow, and the DRLP architecture in efficiently accelerating demanding Deep Q-Learning workloads.

### *3.2.6 FPGA Emulation and Real-Time Demonstration on AWS F1*

Given the complexity and long runtimes associated with simulating large DRL workloads on accelerators, and the cost of traditional hardware emulation systems, an FPGA emulation approach using the Amazon Web Services (AWS) EC2 F1 cloud platform is adopted for validation and testing.

A multi-stage flow was used to migrate the DRLP design to the FPGA:

1. **Local RTL Simulation (VCS):** Initial functional verification of the core accelerator logic using simple memory models.

2. **Local AWS RTL Simulation:** Integration of the accelerator core with the AWS Shell components (AXI interfaces, DMA, DDR controller models) using the AWS-provided simulation environment. This step included adding AXI protocol checkers

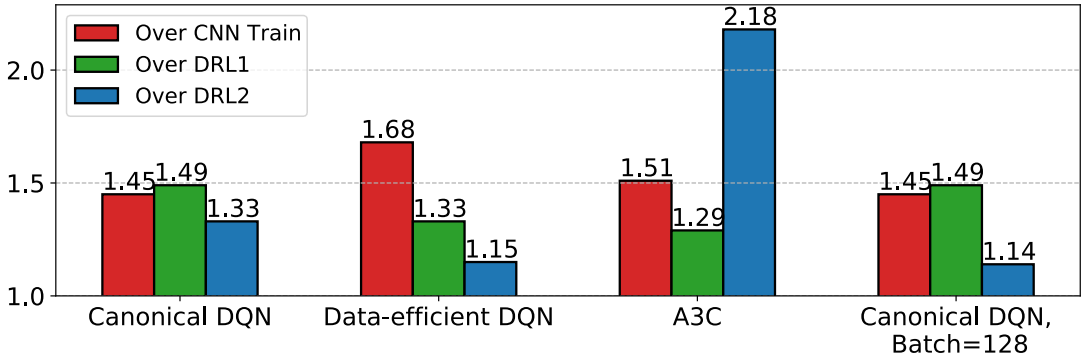



Figure 3.19: Speedup of DRLP over optimistically scaled CNN training accelerator (CNN Train) [33], DRL accelerators [206] (DRL1) and [99] (DRL2) on DNNs training in different DRL algorithms. DRLP achieves speedups of up to 1.68×, 1.49× and 2.18×.

and using a BaseJump-to-AXI bridge (Figure 3.20) to connect the accelerator's native interface to the AXI-based shell.

3. **Local Co-Simulation (C/C++/SystemVerilog DPI):** Running C/C++ testbenches that interact with the RTL simulation via the DPI, allowing for more complex verification scenarios and initial software driver development. Modifications were made to the AWS co-simulation API for better usability (e.g., handling binary DMA data).

4. **FPGA Compilation (Vivado):** Compiling the design locally and then uploading it to AWS. The compile flag instantiates the Integrated Logic Analyzer (ILA) on the DDR, DMA, and Bridge AXI interfaces to provide debug hooks. The compile process can take hours, depending on the design. However, the speed advantage after compiling is significant.

5. **Cloud Runtime Testing:** Launching an F1 instance, loading the compiled design (AFI), and running the actual DRL application using software drivers interacting with the FPGA via PCIe.

This emulation setup enables the creation of a system where the DRLP design running on

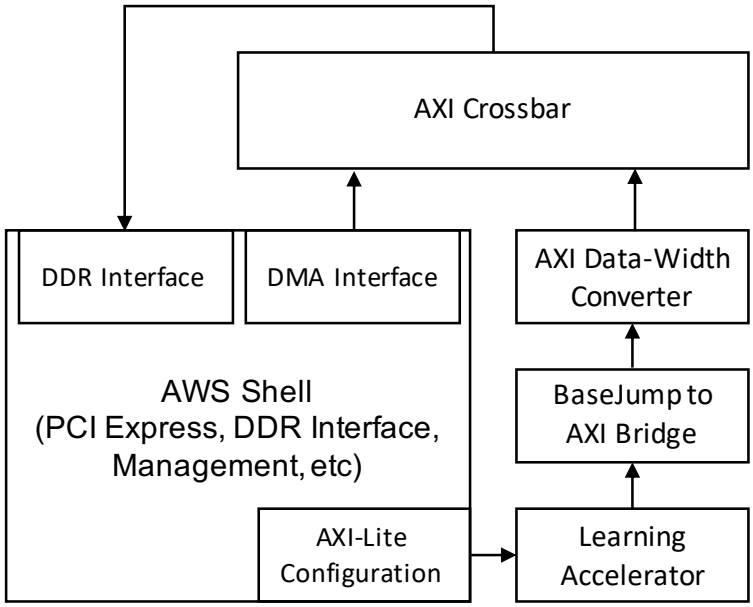



Figure 3.20: Block diagram of accelerator integrated with the AWS EC2 F1 shell.

the F1 FPGA controlled an agent in the OpenAI Gym environment. Specifically, the system successfully demonstrates real-time gameplay of the Atari Breakout game, with the FPGA performing the neural network inferences required for the agent's actions. This serves as a crucial end-to-end validation of the accelerator's functionality within a complete application loop.

While RTL simulation takes hundreds of seconds per game frame, the FPGA execution time per frame is negligible. To make a fair comparison, we define the break-even point as the number of frames after which the total time spent simulating exceeds the one-time FPGA compilation cost plus the near-instantaneous emulation runtime. This break-even point was found to be as low as 6 to 12 frames, as shown in Table 3.2.6. This demonstrates that for workloads such as DRL, cloud FPGA emulation offers a dramatic acceleration in development and testing cycles compared to traditional RTL simulation.

| Design Size | Per Frame RTL Simulation | FPGA Compilation | Break-Even |
|---|---|---|---|
| 16 PEs | 784 s | 9403 s | 12 frames |
| 8 PEs | 836 s | 6823 s | 9 frames |
| 4 PEs | 1065 s | 5663 s | 6 frames |

Table 3.3: Break-even point for RTL simulation v.s. FPGA emulation

### *3.3 Conclusion*

The foundational work described in this chapter, spanning the flexible iFPNA and the co-designed DRLP, provided valuable lessons for tackling larger and more demanding AI workloads. Designing iFPNA underscored the need for programmability and adaptable dataflows to handle diverse network structures. Developing DRLP highlighted the critical importance of analyzing the entire application loop (including data generation and memory access patterns unique to DRL), the benefits of hardware-software co-optimization (state similarity, pre-sampling), and the impact of designing scalable, efficient dataflows ($F|B|C$) and corresponding architectures tailored to specific computational characteristics. Challenges related to memory bandwidth, managing different types of data reuse (across layers, phases, and within operations), and achieving high utilization despite workload variability, encountered in accelerating these earlier paradigms, directly informed the subsequent research focus on the significantly larger scale and distinct architectural demands presented by Large Language Models.

Chapter 4

# REALLM: A HOLISTIC HARDWARE SYSTEM SIMULATION FRAMEWORK FOR LLM SERVING

## *4.1 Introduction*

Generative Large Language Models (LLMs) have rapidly become one of the most exciting and disruptive technologies in the machine learning space, revolutionizing natural language processing tasks and driving advancements in conversational AI [136, 44], code generation [59], and even multimodal content creation [138, 174]. As these language models continue to grow in complexity, capability, and scale, following established scaling laws [93], the hardware required to support them becomes increasingly elaborate and expensive. This exponential growth in computational resource demands raises significant concerns about the scalability, cost-efficiency, and environmental sustainability of deploying such advanced AI systems at scale. Consequently, optimizing LLM inference deployments through effective hardware and system co-design is increasingly essential.

Despite a growing interest and significant progress in designing hardware accelerators for LLMs, a substantial gap remains between the theoretical peak performance of hardware components and the realized system-level efficiency in production environments. Traditional accelerator studies often focus on chip-level metrics such as Floating Point Operations Per Second (FLOPS) and DRAM bandwidth, sometimes neglecting important system-level factors that directly influence service-level objectives (SLOs) like time-to-first-token (TTFT) and time-between-tokens (TBT). Achieving high throughput and low latency in large-scale LLM inference requires the sophisticated orchestration of multiple elements, including diverse parallelism strategies (e.g., data, tensor, pipeline, context, expert parallelism [170, 78, 118, 51]), system-level optimizations (such as mixed continuous batching [74, 3]), efficient inter-device communication, and optimized chip-level kernel execution. Understanding the intricate relationship between hardware architectures, software mapping strategies,

workloads, and optimization targets remains a significant challenge. Accurately bridging the gap between chip-level performance and system-level SLOs is therefore essential for designing next-generation AI accelerators and scalable LLM-serving architectures.

Existing LLM hardware simulators and performance evaluation frameworks often focus either on detailed chip-level simulations or on more abstract system-level modeling, frequently failing to capture the complex interplay between both levels. For instance, while some tools offer detailed accelerator performance modeling [219], they may lack comprehensive system-level execution analysis. Other system-level simulators [15, 108] might approximate the impact of compute, memory, and interconnect bandwidth but often lack kernel-level accuracy, relying on simpler linear models or focusing predominantly on matrix multiplication kernels while neglecting other significant operations like attention mechanisms, normalization, or activation functions. Furthermore, a comprehensive simulation of the full execution graph, integrating aspects like dynamic batching with mixed continuous batching [74], sophisticated request scheduling, and realistic workload traces, is often missing or limited. The sheer scale of the design space, encompassing hardware parameters, parallelism choices, and scheduling algorithms, also makes exhaustive brute-force evaluation computationally prohibitive with many existing tools. This dissertation addresses these critical gaps by presenting a more holistic solution.

To overcome these challenges, we introduce ReaLLM, a holistic and multi-level hardware system simulation framework specifically designed for the comprehensive evaluation and analysis of large-scale LLM inference. ReaLLM aims to bridge the gap between detailed accelerator design and system-wide performance, providing a unified platform for researchers and engineers to understand, evaluate, and make informed optimization decisions for LLM serving systems. The motivation behind ReaLLM is to provide a tool that can accurately model the complex interactions within LLM inference environments, thereby facilitating the development of more efficient and cost-effective hardware and software solutions, as highlighted by its comprehensive feature set in Table 4.1.

The key contributions and pillars of the ReaLLM framework are:

- **Multi-Level Fidelity:** ReaLLM incorporates a multi-level simulation strategy, allow-

| Feature | LLMCompass [219] | GenZ [15] | Optimus [108] | ReaLLM |
|---|---|---|---|---|
| Micro-arch Level Kernel Simulation | ✓ | ✗ | ✗ | ✓ |
| Non-Linear/Complex Kernel | Partial | ✗ | ✗ | ✓ |
| System-Level Parallelism | Partial | ✓ | ✓ | ✓ |
| System-Level Scheduling | ✗ | ✗ | ✗ | ✓ |
| Trace Generation | ✗ | ✗ | ✗ | ✓ |
| Trace-Driven System Simulation | ✗ | ✗ | ✗ | ✓ |
| Comprehensive SLO Analysis | ✗ | ✗ | ✗ | ✓ |
| Interactive Visualization (GUI) | ✗ | ✗ | ✗ | ✓ |

Table 4.1: Comparison of ReaLLM with existing LLM performance evaluation frameworks.

ing users to balance simulation speed and accuracy. This ranges from initial high-level performance estimations to detailed, cycle-aware kernel analysis and full system-level simulation.

- **Accelerated Kernel Analysis:** To tackle the high computational cost of simulating numerous kernel variations arising from different model configurations, batch sizes, and context lengths, ReaLLM introduces the concept of a hypothesis-derived precomputed kernel library. This library, populated through optimized micro-architectural simulations and leveraging interpolation techniques, drastically reduces the overhead of repeated kernel evaluations during system simulation.

- **Trace-Driven System Simulation:** ReaLLM employs trace-driven simulation to capture realistic workload dynamics. By using traces derived from real-world applications or representative synthetic workloads, it can accurately model system behavior under various conditions, including fluctuating request rates and diverse prompt characteristics. This allows for the precise evaluation of parallelism strategies, dynamic batching techniques (e.g., mixed continuous batching), and scheduling policies.

- **Interactive Visualization:** ReaLLM features a web-based graphical user interface (GUI) that allows users to visualize LLM network structures, operator details (such

as types and tensor sizes), and overlay simulated latency information. This interactive component aids in model analysis, debugging, and intuitively understanding performance bottlenecks.

ReaLLM supports flexible hardware architecture specifications, diverse software mapping strategies, and various workload configurations. These inputs can be easily adjusted, enabling users to explore the impact of different design choices on performance and cost. By providing critical metrics such as latency (TTFT, TBT, E2E), throughput, power consumption, hardware cost, and TCO, ReaLLM facilitates the discovery of both high-performance and cost-effective solutions for LLM serving.

This chapter details the architecture, methodologies, and capabilities of the ReaLLM simulation framework. The primary goal is to present ReaLLM as a comprehensive tool that addresses the critical need for accurate and efficient modeling of LLM inference systems, from individual kernel execution up to full-scale, multi-device deployments. We will demonstrate how its integrated approach allows for a deeper understanding of performance bottlenecks and facilitates informed decision-making in the co-design of hardware and software for LLM serving. The remainder of this chapter is structured as follows:

Section 4.2 describes the overall architecture of ReaLLM, its design philosophy, core components, and the interactive visualization GUI. Section 4.3 delves into device-level modeling and the novel accelerated kernel analysis methodology, including the generation and utilization of the precomputed kernel library. Section 4.4 explains the system-level modeling capabilities, focusing on trace-driven simulation, support for parallelism and batching strategies, and inter-device communication. Section 4.5 presents the validation of ReaLLM against real hardware systems, evaluates its simulation efficiency, and showcases illustrative use cases. Finally, Section 4.6 concludes the chapter.

Through this comprehensive exposition, we aim to establish ReaLLM as a valuable tool for advancing the design and optimization of next-generation hardware systems for the ever-evolving landscape of large language models.

## *4.2 Framework Architecture and Core Components*

### *4.2.1 Design Philosophy*

The design of ReaLLM is guided by three core principles aimed at addressing the complexities of LLM inference simulation:

- **Holistic Approach:** ReaLLM is engineered to provide a comprehensive view of LLM inference, bridging the gap between detailed, low-level hardware execution and high-level system-wide performance. It captures the intricate interplay between chip-level kernel behavior, inter-device communication, parallelism strategies, and system-level scheduling and batching policies. This holistic perspective is crucial for accurately identifying true performance bottlenecks and understanding the end-to-end implications of design choices.

- **Multi-Level Simulation Strategy:** Recognizing that different stages of design and analysis require varying levels of detail and simulation speed, ReaLLM incorporates a multi-level simulation strategy. Users can leverage rapid analytical models (such as roofline models) for initial exploration and bottleneck identification. As designs mature and more precise results are needed, ReaLLM integrates detailed micro-architectural kernel simulations and comprehensive system-level simulations. This tiered approach allows for an effective balance between simulation accuracy and computational cost.

- **Scalability and Efficiency:** Simulating large-scale LLM deployments across a vast design space is computationally intensive. A primary design goal for ReaLLM is to achieve scalability and efficiency in its simulation process. This is realized through innovative techniques such as the precomputation of kernel performance data into a reusable library and the use of efficient trace-driven simulation methodologies. These features enable rapid evaluation of numerous configurations, making extensive design space exploration feasible.

### *4.2.2 Overall Simulation Pipeline*

ReaLLM operates through a structured simulation pipeline, designed to systematically evaluate LLM inference performance from individual kernels to the complete system. The framework takes as input an LLM model description (typically in ONNX format for broad compatibility), an abstract hardware description detailing the target system architecture, and workload characteristics, which can be defined by user-specified parameters or through execution traces.

The simulation process in ReaLLM can be conceptualized in two primary phases, as illustrated in Figure 4.1:

1. **Kernel Library Construction:** This initial phase focuses on characterizing the performance of all unique computational kernels within the target LLM(s) on the specified hardware.

    - *Hypothesis-Driven Kernel Generation:* ReaLLM first parses the LLM computational graph (e.g., from ONNX) and systematically identifies all unique kernels (e.g., matrix multiplications, attention operations, normalizations, activations). It then hypothesizes all feasible kernel variants based on factors like batch sizes, input/context lengths, and potential parallelism splits (tensor, data, etc.).
    - *Optimized Kernel Profiling:* Each unique, hypothesized kernel is then simulated using a detailed, micro-architecture-aware kernel simulator (ReaLLM builds upon and extends capabilities found in simulators like LLMCompass [219]). This step determines the optimal mapping strategy (e.g., tiling, loop ordering, dataflow) for each kernel on the target hardware core and records its execution latency. To manage the vast number of potential kernel dimension variants (especially due to varying context lengths), ReaLLM employs interpolation techniques, simulating a subset of key points and interpolating for intermediate values, significantly reducing simulation overhead.
    - *Precomputed Kernel Library:* The outcome of this phase is a precomputed kernel library, which stores the optimal latency and mapping details for each profiled

kernel variant. This library serves as a fast lookup table during the subsequent system simulation phase.

2. **Trace-Driven System Simulation:** Once the kernel library is established, this phase simulates the end-to-end execution of LLM inference tasks across the entire hardware system.

    - *Workload Trace Processing:* The system simulator ingests workload traces, which define sequences of requests with arrival times, prompt lengths, and generation requirements. ReaLLM can utilize user-provided traces or generate synthetic traces representative of real-world applications (e.g., conversational AI, code generation, based on datasets like the Azure LLM Inference Dataset [126]).
    - *System-Level Scheduling and Execution:* A built-in scheduler models various batching (e.g., continuous batching, mixed continuous batching with prefill chunking) and scheduling strategies. It manages request queues and dispatches execution tasks to the simulated hardware.
    - *Hardware Simulation:* For each step in the LLM execution graph, the simulator retrieves kernel latencies from the precomputed kernel library. It also models inter-device communication overhead based on the hardware's network topology, bandwidth, latency, and the chosen collective communication algorithms (e.g., Ring, 2D-Ring, Tree-based allreduce [215, 164, 83]).

The final outputs of the ReaLLM pipeline include detailed performance metrics such as Service Level Objectives (SLOs) like Time-To-First-Token (TTFT), Time-Between-Tokens (TBT), and end-to-end request latency. Additionally, the framework provides data for TCO analysis, identifies system bottlenecks, and can feed information into the visualization GUI. This comprehensive pipeline allows for robust evaluation of how different hardware designs, software strategies, and workload conditions impact overall LLM serving performance and cost.

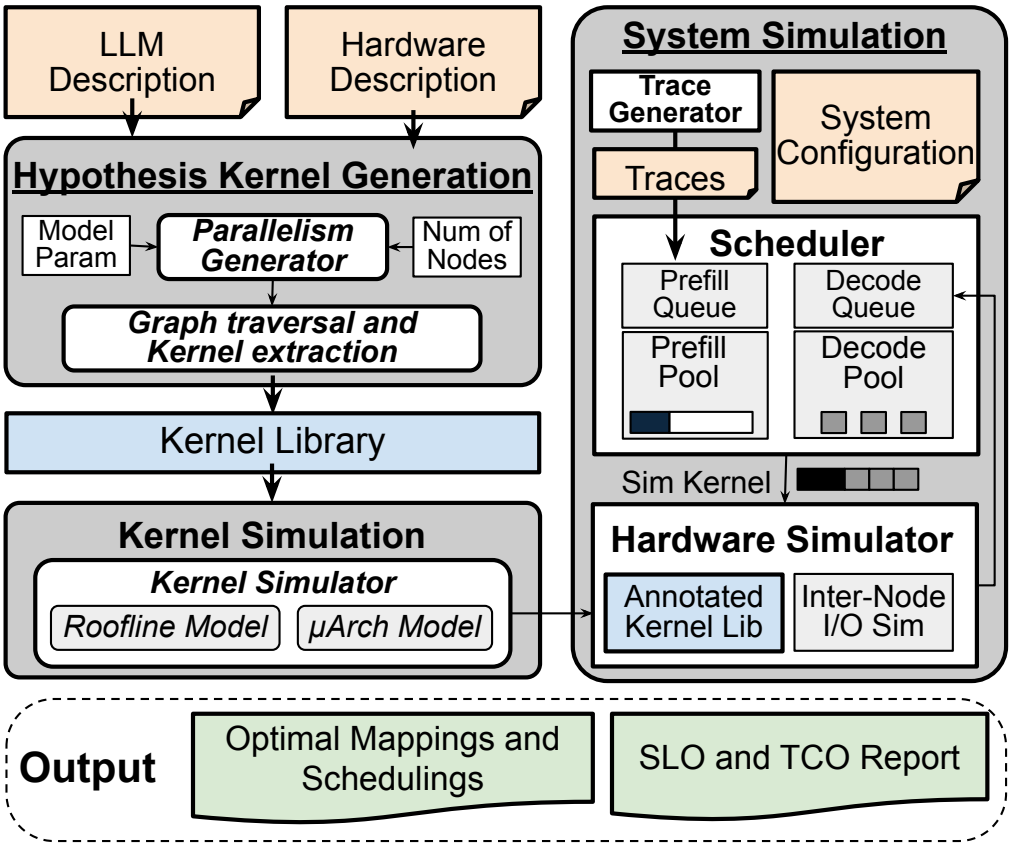



Figure 4.1: High-level overview of the ReaLLM simulator pipeline.

### *4.2.3 Abstract Hardware Representation*

To support a wide variety of existing and future hardware architectures, ReaLLM employs a flexible and parameterizable abstract hardware description. Users define the hardware system using a YAML-based configuration file, which specifies the architectural details across multiple hierarchical levels: the chip, the package, and the server.

An example of this hierarchical representation is shown in Figure 4.2:

- **Chip Level:** Defines the characteristics of a single processing die. This includes parameters such as the technology node (e.g., 7nm, 5nm), the organization and capacity of on-chip memory hierarchies (e.g., global SRAM or L2 cache, local shared memory or

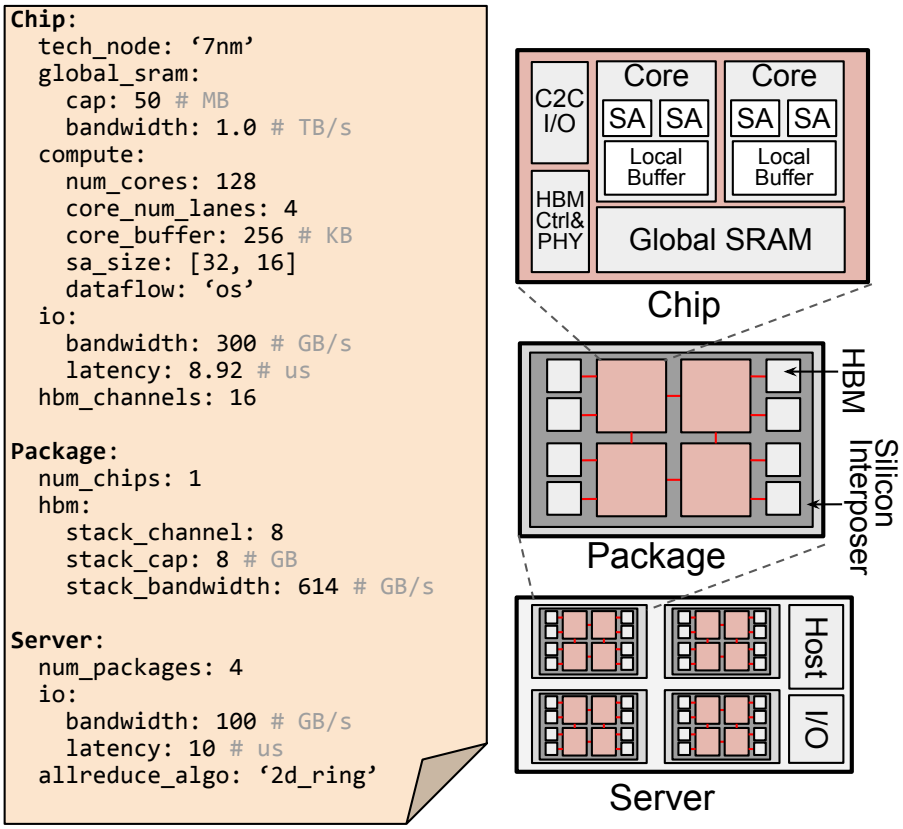



Figure 4.2: Example of ReaLLM's abstract hardware description hierarchy, depicting chip, package, and server level components and their parameterization.

L1 cache per core), and the details of the compute units. Compute units can be specified with attributes like the number of cores, the number and size of systolic arrays (SAs) or tensor cores, vector processing capabilities, and their operational frequency. Dataflow within compute units (e.g., weight-stationary, output-stationary for SAs) can also be configured.

- **Package Level:** Describes how one or more chips are integrated. This includes the number of chips per package, the type and configuration of off-chip memory (e.g., High Bandwidth Memory - HBM stacks, specifying capacity, channels, and bandwidth), and chip-to-chip (C2C) interconnect details if multiple dies are on the same package (e.g., via silicon interposer).

- **Server Level:** Defines the system-level integration of packages. This includes the number of packages (devices) per server, device-to-device interconnect specifications (e.g., NVLink, TPU Interconnect, PCIe, Ethernet), including bandwidth and latency, and the topology of this interconnect.

This hierarchical and parameterized approach allows users to easily model existing hardware like GPUs and TPUs, as well as explore hypothetical future architectures by modifying these parameters. The configuration can specify single values for fixed designs or lists of values to facilitate exploration of different hardware design points, although automated design space sweeping is a future work direction.

### *4.2.4 Interactive LLM Network and Performance Visualizer*

To aid in the understanding of LLM computational graphs and the interpretation of simulation results, ReaLLM includes a web-based Graphical User Interface (GUI), an example view of which is presented in Figure 4.3. This interactive tool is designed to provide users with an intuitive way to analyze model structures, configure simulation parameters, and pinpoint performance characteristics.

The GUI facilitates the following workflow and capabilities:

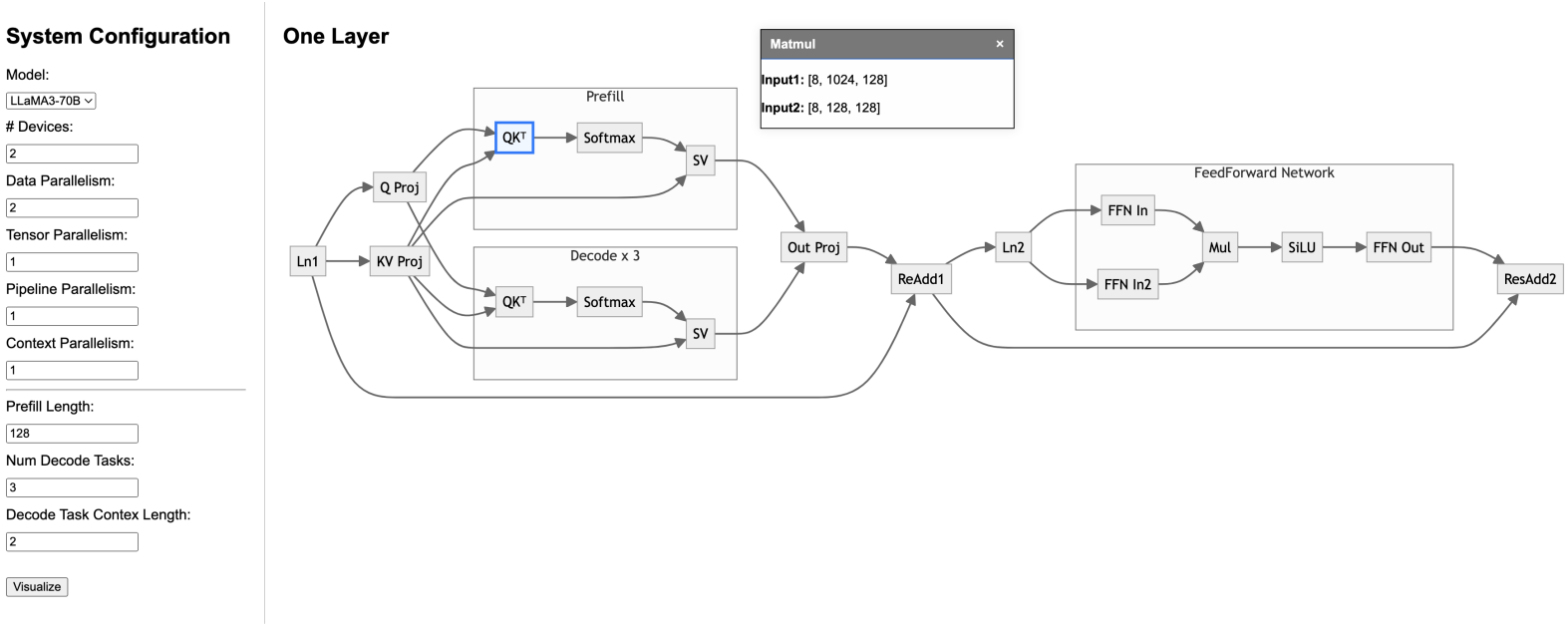



Figure 4.3: The web-based Graphical User Interface of ReaLLM.

**Configuration Input:** Users begin by selecting the target LLM (e.g., Llama, DeepSeek) from a predefined list. They can then choose from a set of predefined hardware configurations or provide their own abstract hardware description (as detailed in Section 4.2.3). Users can also specify system-level configurations, such as the desired parallelism strategies (e.g., tensor parallelism degree, pipeline depth).

**Simulation-Driven Data Generation:** Based on these user inputs, the ReaLLM backend (as described in Section 4.2.2) calculates the effective size of each computational kernel as it would be mapped onto each device under the specified parallelism. It then retrieves or computes the corresponding latency for these mapped kernels using the precomputed kernel library.

**Computational Graph Visualization with Performance Annotation:** The GUI renders the computational graph of the selected LLM. The generated data, including the per-device mapped kernel sizes and their simulated latencies, are then annotated directly onto this visual representation of the network. This allows users to see, for instance, how different parts of the model contribute to overall latency on specific devices.

**Bottleneck Identification and Comparative Analysis:** By visualizing the latencies across the graph, users can easily identify performance-critical operators and potential bottlenecks within the LLM execution flow for a given hardware and system configuration. The GUI enables users to iteratively change hardware or system configurations (e.g., try a different parallelism scheme, or simulate on a hypothetical faster hardware) and observe the impact on the annotated graph, facilitating direct comparison and aiding in design decisions.

By providing this interactive loop of configuration, simulation-driven data generation, and visual feedback, the ReaLLM GUI significantly enhances the user's ability to understand model performance, debug system configurations, and explore the impact of different hardware and software choices. It transforms raw simulation data into actionable visual insights, making the complex task of LLM system optimization more accessible and intuitive.

## *4.3 Device-level Modeling and Kernel Profiling*

Effective simulation of LLM inference systems requires accurate and efficient modeling of computational kernels at the device level. This section details ReaLLM's approach to this

critical aspect, covering the systematic identification and characterization of kernels, the development of a precomputed kernel library to accelerate simulation, and the use of foundational performance models for initial estimations. This device-level understanding forms the bedrock upon which system-level simulations are built.

### *4.3.1 Kernel Identification and Hypothesis Generation*

The first step in device-level modeling within ReaLLM is to identify and characterize all unique computational kernels executed by an LLM. This process is essential for understanding the computational workload and for building the precomputed kernel library.

#### *Systematic Kernel Extraction from LLM Graphs*

ReaLLM ingests LLM models, typically provided in the Open Neural Network Exchange (ONNX) format, which offers a standardized graph-based representation of neural networks. This compatibility allows ReaLLM to support a wide range of LLM architectures and their constituent operators. The framework parses the ONNX graph, systematically traversing it to identify and count occurrences of each unique kernel type. Common kernels include matrix multiplications (MatMul) for fully connected layers and attention projections, attention mechanisms themselves (e.g., scaled dot-product attention, including its components like Softmax and element-wise operations), normalization layers (e.g., LayerNorm), activation functions (e.g., GELU, SiLU, SwiGLU), and various element-wise operations. For each identified kernel, ReaLLM extracts its fundamental properties, such as its type and the static shapes of its weight tensors.

#### *Impact of Parallelism, Batching, and Context Lengths on Kernel Dimensions*

The actual dimensions of a kernel at runtime are highly dynamic and depend on several factors, including the batch size of incoming requests, the input sequence length (particularly for the prefill phase), the accumulated context length (for the decode phase), and the chosen multi-device parallelism strategy. ReaLLM incorporates a sophisticated shape inference engine that propagates these dynamic dimensions throughout the LLM graph.

LLM inference systems often distribute the workload across multiple hardware devices (nodes) using various parallelism techniques to manage the substantial memory and computational requirements. ReaLLM supports commonly adopted parallelism strategies, including data, tensor, pipeline, context, and expert parallelism. The choice of parallelism strategy and the degree of parallelism directly influence the dimensions of the kernels executed on each device. For example, tensor parallelism will reduce the dimension of a matrix multiplication along which it is sharded. ReaLLM's parallelism generator considers the model hyperparameters (e.g., number of attention heads, number of layers) and system constraints (e.g., number of available devices) to enumerate valid parallelism configurations.

Given batch size, input/context lengths, and parallelism configurations, ReaLLM calculates and hypothesizes all possible sizes for each kernel. The top part of Table 4.2 lists MatMul kernels for a Llama-like [122] LLM, which uses group-query attention and gated linear units. Each MatMul operation is represented as $(B_1, B_2, M, K) \times (B_2, K, N) = (B_1, B_2, K, N)$. $l_{in}$ denotes the input sequence length, which is the prompt length for prefill and 1 for decode. $l_{ctx}$ denotes the context length, which is the prompt length for prefill and past context length for decode. All divisions in Table 4.2 use ceiling division to ensure the identification of the system's critical path. The table also lists collective operations required for certain parallelism strategies. Context parallelism requires SendRecv operations for `q_k` and `s_v` since each node must receive the complete query and scores. Tensor parallelism requires AllReduce operations for `o_proj` and `mlp_dn` to aggregate partial results. The bottom section of Table 4.2 presents MatMul kernels for multi-latent attention, which introduces additional smaller kernels. Low-rank adaptation is applied to key and value projections, compressing them into a lower-dimensional space $d_c$.

By iterating over all input factors, ReaLLM constructs a complete kernel library for further simulation. Pipeline and expert parallelism do not change kernel sizes but affect kernel execution times, which is accounted for in system simulation.

| MatMul | $\mathbf{B}_1$ | $\mathbf{B}_2$ | **M** | **K** | **N** | **Collective** |
|---|---|---|---|---|---|---|
| q_proj | $\frac{batch}{D}$ | 1 | $\frac{l_{in}}{C}$ | $d_m$ | $d_h\frac{n_h}{T}$ | |
| kv_proj | $\frac{batch}{D}$ | 1 | $\frac{l_{in}}{C}$ | $d_m$ | $d_h\frac{n_{kv}}{T}$ | |
| q_k | $\frac{batch}{D}$ | $\frac{n_{kv}}{T}$ | $\frac{n_h}{n_{kv}}\frac{l_{in}}{C}$ | $d_h$ | $l_{ctx}$ | SR($\frac{batchn_hl_{in}d_h}{DTC}$) |
| s_v | $\frac{batch}{D}$ | $\frac{n_{kv}}{T}$ | $\frac{n_h}{n_{kv}}\frac{l_{in}}{C}$ | $l_{ctx}$ | $d_h$ | SR($\frac{batchn_hl_{in}l_{ctx}}{DTC}$) |
| o_proj | $\frac{batch}{D}$ | 1 | $\frac{l_{in}}{C}$ | $d_h\frac{n_h}{T}$ | $d_m$ | AR($\frac{batchl_{in}d_m}{DC}$) |
| mlp_gate | $\frac{batch}{D}$ | 1 | $\frac{l_{in}}{C}$ | $d_m$ | $\frac{d_{ffn}}{T}$ | |
| mlp_up | $\frac{batch}{D}$ | 1 | $\frac{l_{in}}{C}$ | $\frac{d_{ffn}}{T}$ | $d_m$ | |
| mlp_dn | $\frac{batch}{D}$ | 1 | $\frac{l_{in}}{C}$ | $\frac{d_{ffn}}{T}$ | $d_m$ | AR($\frac{batchl_{in}d_m}{DC}$) |
| q_k_1 | $\frac{batch}{D}$ | $\frac{n_h}{T}$ | $\frac{l_{in}}{C}$ | $d_h$ | $d_c$ | SR($\frac{batchn_hl_{in}d_h}{DTC}$) |
| q_k_2 | $\frac{batch}{D}$ | 1 | $\frac{n_h}{T}\frac{l_{in}}{C}$ | $d_c$ | $l_{ctx}$ | SR$\frac{batchn_hl_{in}d_c}{DTC}$ |
| q_k_pe | $\frac{batch}{D}$ | 1 | $\frac{n_h}{T}\frac{l_{in}}{C}$ | $d_h^R$ | $l_{ctx}$ | SR($\frac{batchn_hl_{in}d_h^R}{DTC}$) |
| s_v_1 | $\frac{batch}{D}$ | 1 | $\frac{n_h}{T}\frac{l_{in}}{C}$ | $l_{ctx}$ | $d_c$ | SR($\frac{batchn_hl_{in}l_{ctx}}{DTC}$) |
| s_v_2 | $\frac{batch}{D}$ | $\frac{n_h}{T}$ | $\frac{l_{in}}{C}$ | $d_c$ | $d_h$ | SR($\frac{batchn_hl_{in}d_c}{DTC}$) |

Table 4.2: MatMul kernel size for Llama-like LLM (top) and multi-latent attention (bottom). D, T, C are the sizes of data, tensor, and context parallelism. AR=AllReduce, SR=SendRecv.

| In a LLM | Batch Sizes | Parallelisms | Context Lengths | Total |
|---|---|---|---|---|
| 10 | 10 | $10^2$ | $10^5$ | $10^9$ |

Table 4.3: Order of magnitude of Matmul kernel variations given different input factors.

### *4.3.2 The Precomputed Kernel Library: Enabling Rapid Simulation*

A major challenge in applying accurate kernel simulation to end-to-end LLM system modeling is slow speed of simulation. For example, simulating a single inference pass with LLMCompass [219] can take several minutes, mainly due to the long simulation time of *Matmul* kernels. Accurate Matmul simulation requires exploring a vast mapping and scheduling space, including L2 and L1 tiling, loop ordering, and systolic array dataflows, etc. The number of possible mapping strategies for a single Matmul operation can reach millions. While LLMCompass [219] applies heuristics to reduce this search space, simulating each Matmul still takes a minute.

This speed is impractical for system-level simulation, as it dramatically increases the number of required kernel evaluations. Table 4.3 highlights the order of magnitude of Matmul kernels that need to be simulated. An LLM contains approximately 10 distinct Matmul kernels. Considering variations in input request rates, batching strategies, and different parallelism configurations (data, tensor, pipeline, context, expert, etc.), the number of Matmul kernels grows exponentially. Furthermore, with dynamic prompt lengths during prefill and context lengths during decode, modern LLMs with context lengths up to 128K introduce over $10^5$ variations. As a result, the total number of Matmul simulations required for a complete system evaluation can reach $10^9$, which is computationally prohibitive given that each simulation takes minutes.

The core motivation behind ReaLLM's precomputed kernel library is to decouple the time-consuming process of detailed kernel profiling from the system-level simulation. By simulating each unique, hypothesized kernel instance *once* and storing its performance characteristics, ReaLLM avoids redundant computations during the dynamic system simulation phase. This approach significantly accelerates the overall evaluation process, making it fea-

sible to explore a wide range of system configurations and workload scenarios.

ReaLLM's kernel simulator builds upon the foundations of existing open-source hardware evaluation frameworks like LLMCompass [219]. It extends these capabilities by incorporating support for a wider range of attention mechanisms (e.g., grouped-query attention, multi-latent attention), additional operators in modern LLMs (e.g., SiLU activation, element-wise operations for gated linear units), and optimizations such as multiprocessing to parallelize the evaluation of different kernel mappings.

For a given kernel and target hardware core (defined by the abstract hardware description, see Section 4.2.3), the kernel simulator explores a vast space of possible mapping strategies. This includes optimizing tiling sizes for data movement across multiple memory hierarchies (e.g., off-chip memory to L2 cache/global SRAM, L2 to L1 cache/local shared memory, L1 to registers/LO), determining optimal loop ordering at different cache levels, considering techniques like L2 double buffering, and selecting appropriate dataflows for specialized compute units like systolic arrays (e.g., weight-stationary, output-stationary). The goal is to find the mapping that minimizes execution latency for each kernel on the specified hardware.

*Latency Interpolation*

Even with optimized profiling, simulating every single dimensional variant of a kernel, especially those affected by continuous variables like context length ($l_{ctx}$), can be excessively time-consuming. As noted in Table 4.3, context lengths can introduce a large number of variations. To mitigate this, ReaLLM employs latency interpolation.

Instead of simulating every possible context length for affected kernels (e.g., `q_k` and `s_v` MatMuls during prefill), ReaLLM samples a subset of key points (e.g., logarithmically spaced context lengths). Full, detailed simulations are run for these sampled points. For intermediate, unsampled context lengths, the latency is then interpolated from the nearest simulated points. As demonstrated in Figure 4.4, linear interpolation between these logarithmically spaced points provides high accuracy (e.g., average errors of 0.90% to 3.63% for MatMul dimension sweeps) while drastically reducing the number of full simulations

required. This technique is particularly effective because kernel latency often exhibits predictable, piece-wise linear trends with respect to changes in a single dimension, once other dimensions and mapping strategies are fixed.

*Initial Performance Estimation (Optional Stand-in for Early Exploration)*

While the precomputed kernel library provides detailed and accurate latency information, its construction can still be a time-intensive prerequisite for full system simulation. For very early-stage design exploration, or when a quick assessment of potential hardware bottlenecks is needed without the full detail of micro-architectural simulation, ReaLLM can also incorporate simpler, analytical roofline models.

The roofline model [207] is a well-established analytical tool that provides an insightful visual representation of the achievable performance of a given hardware architecture based on its peak computational throughput and peak memory bandwidth. ReaLLM uses the roofline model to estimate the performance of individual kernels. By comparing a kernel's operational intensity (FLOPs per byte of data moved) against the hardware's roofline, one can quickly determine if the kernel is likely to be compute-bound or memory-bound. This model is particularly advantageous during the initial phases of hardware design space exploration, enabling designers to rapidly assess a massive design space and identify high-level bottlenecks for a given workload before committing to more detailed simulations.

These initial estimation techniques serve as optional, faster stand-ins when the full detail of the precomputed kernel library is not immediately required or available. They complement the more detailed simulation capabilities, aligning with ReaLLM's multi-level fidelity design philosophy.

## *4.4 System-level Modeling and Trace-Driven Simulation*

Building upon the device-level kernel characterization and the precomputed kernel library detailed in the previous section, ReaLLM employs a sophisticated system-level simulation engine. This engine is designed to model the end-to-end execution of LLMs in a multi-device environment, capturing the complexities of real-world workloads and system dynamics. This section elaborates on ReaLLM's approach to trace generation and utilization, its dynamic

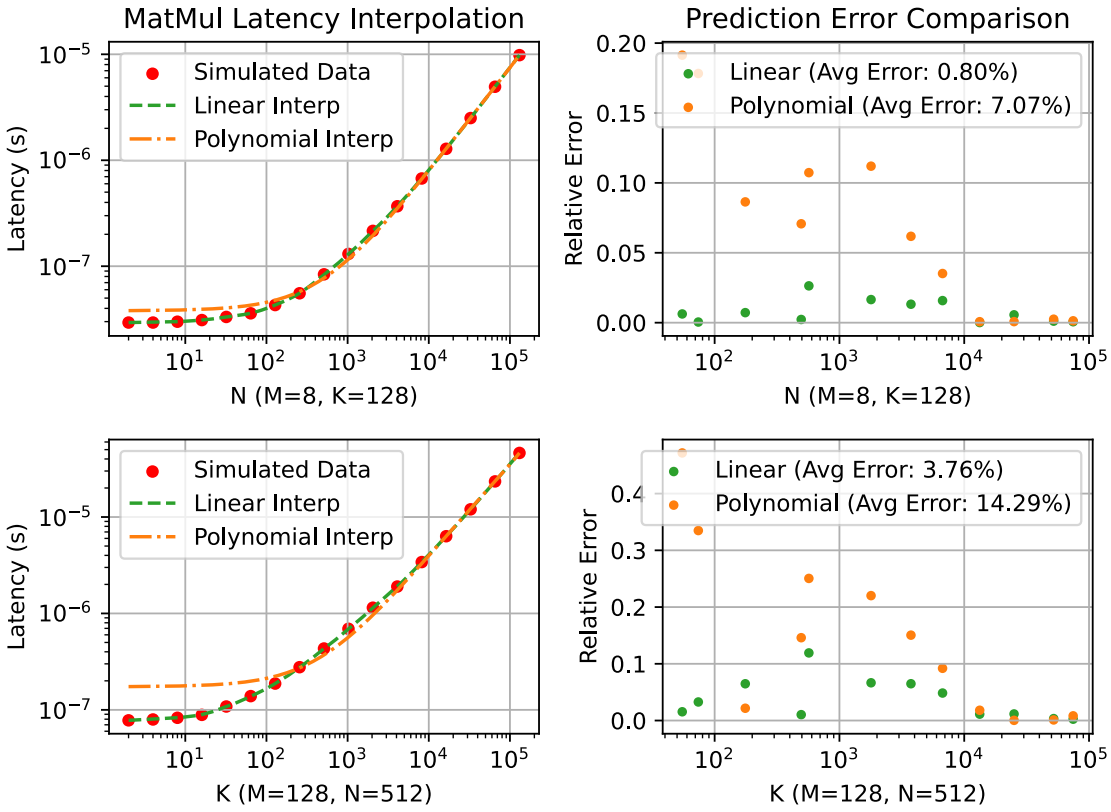



Figure 4.4: Comparison of MatMul latency interpolation methods. (Left) Simulated latency data points (red dots) with linear and polynomial interpolation. (Right) Relative prediction error for linear and polynomial interpolation. Linear interpolation generally achieves lower error rates.

task scheduler, the mechanisms for simulating hardware execution including inter-device communication, and the generation of key performance results.

### *4.4.1 Capturing Realistic Workloads: Trace Generation and Utilization*

The performance of an LLM inference system is heavily influenced by the characteristics of the incoming workload. To ensure realistic and relevant evaluations, ReaLLM incorporates a robust trace-driven simulation methodology. Traces define the sequence of user requests, their arrival patterns, and the specifics of each request, such as prompt length and the number of tokens to be generated.

ReaLLM offers flexibility in how these traces are provided:

- **User-Provided Traces:** Users can supply their own custom traces, allowing them to model specific, known workload patterns relevant to their applications or research questions.

- **Built-in Trace Generator:** For users who may not have access to custom traces or wish to explore standardized workload scenarios, ReaLLM includes a built-in trace generator. This generator can synthesize traces that mimic real-world applications. To achieve this, it leverages insights from publicly available production trace datasets, such as the Azure LLM Inference Dataset 2023 [126]. By analyzing such datasets, ReaLLM can model key workload characteristics, including:

  - **Request Arrival Patterns:** Simulating different request rates to stress-test the system and evaluate its performance under varying loads.
  - **Context Length Distributions:** Capturing the typical prompt lengths and the number of generated tokens for different applications, such as conversational AI versus code generation tasks. As observed in Figure 4.5, conversational tasks often have shorter input prompts but require longer generated sequences (lower input-to-output token ratio) compared to coding tasks, which might have longer initial contexts but shorter completions.

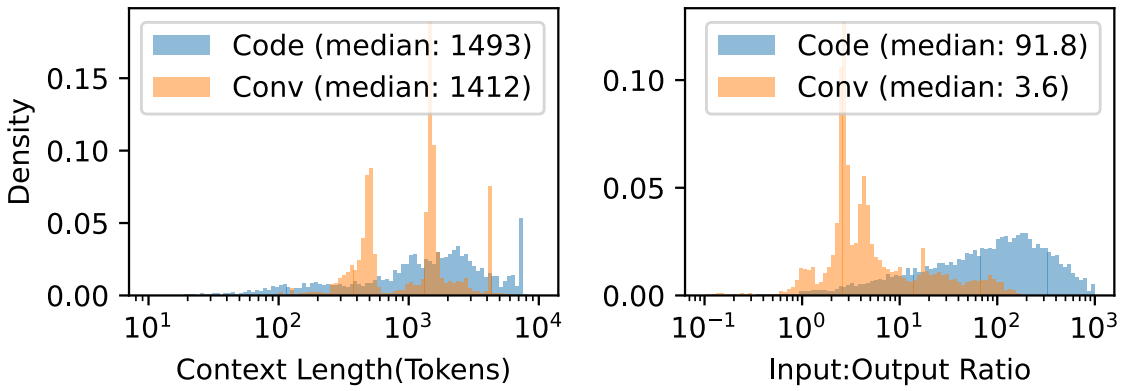



Figure 4.5: Distribution of context lengths and input-to-output token ratios for coding and conversational tasks, derived from the Azure LLM Inference Dataset [126].

– **Input-to-Output Token Ratios:** Modeling the relationship between the length of the input prompt and the expected length of the generated response.

By utilizing realistic traces, whether user-defined or synthesized, ReaLLM can simulate system behavior under conditions that closely mirror production environments, leading to more accurate and actionable performance insights.

### *4.4.2 Simulating System Dynamics and Execution*

As shown in Figure 4.1, at the heart of ReaLLM's system-level simulation is a dynamic task scheduler that interacts with a hardware simulation model. This combination allows for the modeling of complex execution flows, parallelism strategies, and communication patterns.

The ReaLLM task scheduler is responsible for managing incoming user requests from the processed trace and orchestrating their execution on the simulated hardware. Its key functions include:

**Request Queue Management:** Incoming requests are initially placed into appropriate queues (e.g., a prefill queue for new requests, a decode queue for ongoing generation tasks) based on their arrival times and current processing state.

**Dynamic Batching Strategies:** The scheduler implements various dynamic batching techniques (continuous batching, mixed continuous batching, etc) to optimize resource utilization and throughput.

**Execution Task Generation:** Based on the selected batching algorithms, the scheduler groups requests (or parts of requests, in the case of chunked prefill) into execution tasks. Each execution task is typically represented by an integer prefill length (if any prefill operations are part of the batch) and an array of integers denoting the current context lengths of all decode tasks included in that batch.

Once an execution task is processed by the hardware simulator, the scheduler updates the status of all associated requests. Unfinished requests (i.e., those requiring further token generation) are placed back into the decode queue for subsequent iterations.

### *4.4.3 End-to-End System Performance Simulation and Result Output*

The hardware simulator processes the execution tasks generated by the task scheduler. For each task, which represents a batch of operations (prefill and/or decode) for one pass through the relevant part of the LLM:

1. It traverses the LLM's execution graph (or the relevant sub-graph for the current pipeline stage).

2. For each computational kernel in the graph, it determines the effective dimensions based on the current batch composition and parallelism strategy.

3. It queries the precomputed kernel library (Section 4.3.2) using these effective dimensions to retrieve the pre-profiled execution latency for the target hardware. If an exact match is not found (e.g., for an un-sampled context length), latency interpolation is applied as described in Section 4.3.2.

4. It calculates the latency of any necessary inter-device communication operations associated with the current computational step (e.g., AllReduce after a distributed MatMul).

5. The computation and communication latencies are summed to determine the total time for the current step or layer. This process is repeated for all operations in the execution task.

This cycle-accurate (for kernels) and model-based (for communication) approach allows ReaLLM to simulate the time progression of each request through the system.

*Inter-Device Communication Modeling*

In distributed LLM inference, inter-device communication can be a significant performance factor. ReaLLM incorporates a detailed communication model to capture these overheads. The time required to transmit an N-byte message between any two directly connected nodes is modeled using the standard linear model: $T_{comm} = \alpha + N\beta$, where $\alpha$ represents the per-message latency (independent of message size, capturing software overheads and network interface delays) and $\beta$ denotes the reciprocal of the bandwidth (i.e., per-byte transmission time). These parameters ($\alpha$ and link bandwidth) are specified in the abstract hardware description (Section 4.2.3).

For collective communication operations, which are common in parallelism strategies like tensor parallelism (e.g., AllReduce for aggregating partial results) or pipeline parallelism (e.g., point-to-point Send/Receive between stages), ReaLLM supports several widely adopted algorithms. The time complexity for these operations depends on the algorithm, the number of participating nodes ($p$), message size ($N$), and the underlying point-to-point communication parameters. Supported algorithms and their approximate time complexities are listed in Table 4.4.3.

The choice of communication algorithm can be configured by the user or determined by ReaLLM based on heuristics. The latencies calculated from these models are added to the computation latencies to provide an end-to-end estimate for each step in the inference process.

| **Algorithm** | **Approximate Time Complexity** |
|---|---|
| Ring AllReduce (AR) | $2(p-1)\alpha + 2\frac{p-1}{p}N\beta$ |
| 2-D Ring AllReduce [215] | $4(\sqrt{p}-1)\alpha + 2\frac{\sqrt{p}-1}{\sqrt{p}}N\beta$ |
| Two Tree AllReduce [164] | $4\log_2(p)\alpha + 2N\beta + 4\sqrt{2\log_2(p)\alpha N\beta}$ |
| Two Tree Broadcast (BC) [164] | $2\log_2(p)\alpha + N\beta + 2\sqrt{2\log_2(p)\alpha N\beta}$ |
| Hierarchical AllReduce [83] | Time(LocalAR) + Time(GlobalAR) + Time(LocalBC) |

Table 4.4: Supported collective communication algorithms in ReaLLM and their approximate time complexities for $N$-byte tensors among $p$ nodes.

*Outputting Performance Metrics and Service Level Objectives (SLOs)*

Throughout the simulation, ReaLLM meticulously records the timing information for each request, including its arrival time and the generation time of each output token. From this raw data, it calculates and reports key performance metrics, particularly Service Level Objectives (SLOs), which are critical for evaluating the quality of service of an LLM inference system. The primary SLOs measured by ReaLLM include:

**Time-To-First-Token (TTFT):** The latency from when a user request arrives at the system until the first output token is generated and available. This is a crucial metric for user-perceived responsiveness in interactive applications.

**Time-Between-Tokens (TBT)** (also known as Time Per Output Token - TPOT): The average latency to generate each subsequent token after the first one. A low and consistent TBT is important for a smooth user experience during streaming output.

**End-to-End (E2E) Latency:** The total time taken from the arrival of a request until the complete response (or a specified number of tokens) has been generated.

ReaLLM can report these SLOs as distributions or specific percentile values (e.g., P50 median, P90, P99) to capture the system's performance consistency under varying conditions and SLA (Service Level Agreement) thresholds.

In addition to these primary SLOs, the simulation results can be used to derive other metrics like system throughput (e.g., tokens per second, requests per second). The detailed

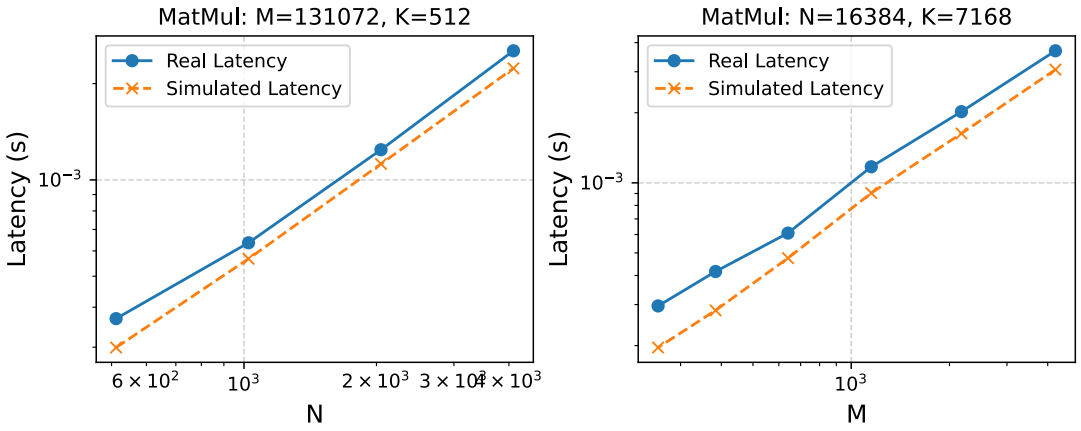



Figure 4.6: Validation of kernel latency predictions on A100. Each subfigure compares real and simulated latencies for MatMul at different input sizes.

data generated by ReaLLM, including kernel execution times and communication overheads, also allows for in-depth bottleneck analysis. The framework aims to not only provide these metrics but also to help identify the optimal chip-level kernel mappings (discovered during kernel library construction) and the system-level scheduling and parallelism strategies that lead to the best SLO performance for a given scenario.

## *4.5 Validation and Evaluation*

### *4.5.1 Validation Against Real Hardware*

To validate ReaLLM's accuracy, we compare the predicted kernel latencies and end-to-end request latencies against real measurements on NVIDIA A100 and H100 systems.

**Kernel-Level Validation:** To assess the accuracy of ReaLLM at the kernel level, we compare predicted versus measured latencies for key LLM inference operations on a NVIDIA A100 GPU. Figure 4.6 shows the latency of MatMul operations across different input sizes, demonstrating that ReaLLM's predictions align closely with real execution times.

This high fidelity ensures that kernel-level estimations in ReaLLM provide precise performance insights, making it a reliable tool for evaluating large-scale LLM inference systems.

**End-to-End Latency Validation:** Beyond kernel-level validation, we assess end-to-end inference accuracy by comparing ReaLLM's simulated latencies for LLaMA-70B running on four H100 GPUs against real-world traces (Figure 4.7). The results indicate that ReaLLM predicts the end-to-end time (E2E) with an average error of 9.07% across 90 test traces. Notably, most of the early differences arise from transient system warm-up effects and variations in initial scheduling, while later traces have improved accuracy. This strong alignment with real hardware confirms the robustness and reliability of ReaLLM's system-level simulation. Furthermore, ReaLLM's trace-driven scheduling and dynamic batching models effectively adapt to fluctuating workloads, accurately reflecting real-world deployment scenarios. By incorporating execution-aware scheduling strategies, ReaLLM ensures that its predictions remain highly relevant for large-scale LLM inference studies.

### *4.5.2 Bottleneck Analysis with Fixed Workloads*

To better understand the limitations of serving LLMs on modern platforms, we use ReaLLM to simulate 64 TPU v5p chips running an LLM sized equivalently to GPT-3 [21]. Each v5p chip achieves a peak BF16 performance of 459 TFLOPS, alongside 95 GB of HBM2 offering a total 2765 GB/s bandwidth. Inter-chip communication bandwidth peaks at 300 GB/s per direction. All 64 chips are connected by a $4 \times 4 \times 4$ 3-D torus network, as described in [84].

Figure 4.8 shows the latency breakdown for two tasks with different input and output lengths. The left task is input-dominated, with 256 input tokens and 64 output tokens. The right task is output-dominated, with 64 input tokens and 256 output tokens. We identified some insights listed below.

**IO-bound in the prefill stage -** During prefill, as shown in the top row in Figure 4.8, I/O accounts for the majority of the latency. Since all tokens in the prompt have to be sent to the model at the same time, the activation tensors for the allreduce is large. Two allreduce operations are required per layer, one in the self-attention network (SA) and one in the feed-forward network (FFN). In GPT-3, the activation tensor size with 256 input tokens is 5.86 MB in the SA and 23.44 MB in the FFN. Across 64 chips, the I/O time becomes much longer than the compute time. Batch sizes do not affect the ratio of I/O time to computation

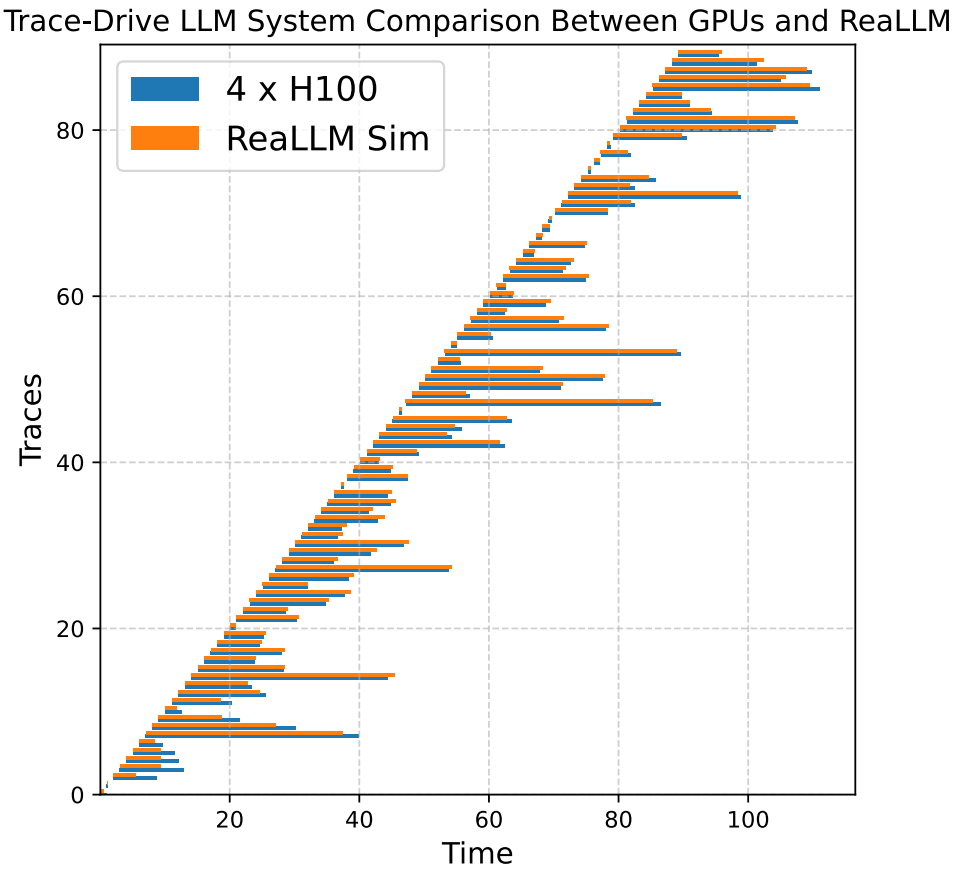



Figure 4.7: Comparison of simulated and real end-to-end request latencies for LLaMA-70B inference on a four-H100 system.

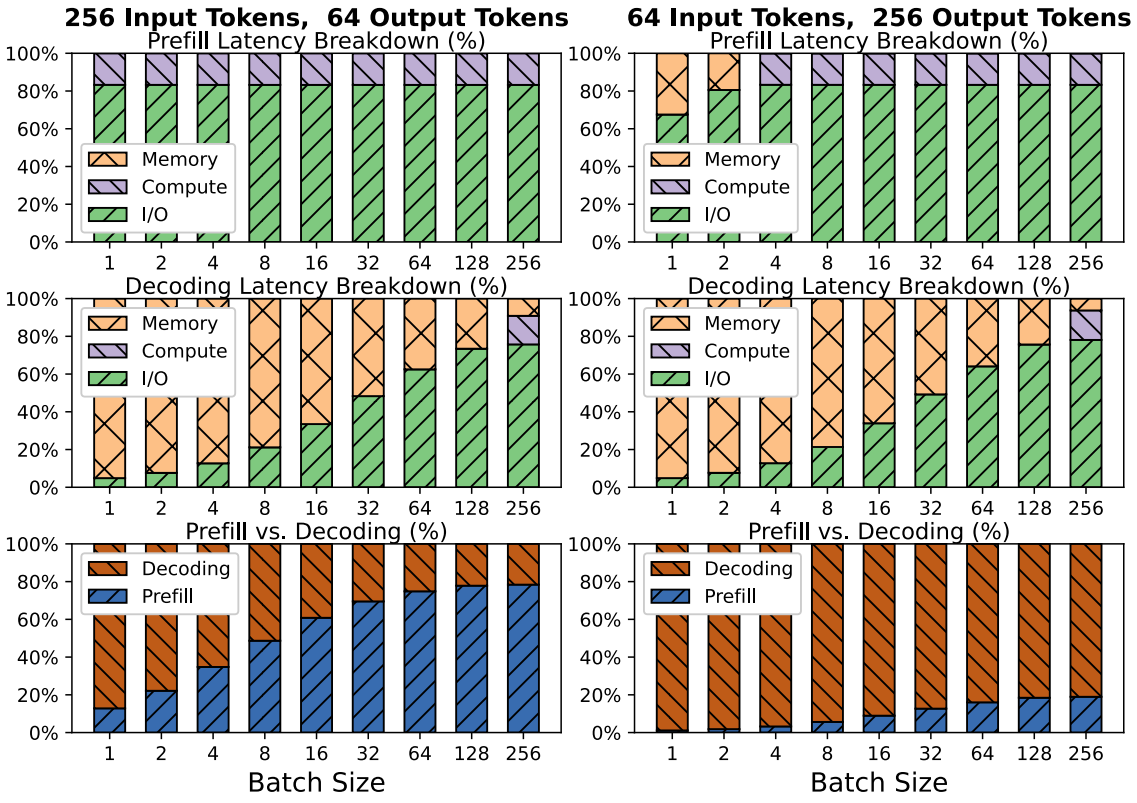



Figure 4.8: Latency breakdown of 64 TPU v5p chips. The system is IO-bound in prefill (top row), and memory-bound for most batch sizes while decoding (second row). Decoding often has longer latency than the prefill (bottom row).

time because both activation size and number of computations increase linearly with batch sizes.

**(Mostly) Memory-bound in the decoding stage:** During decoding, the activation becomes a 1D vector. The system becomes memory-bounded when the batch size is small, as shown in the middle row of Figure 4.8. Due to the low operational intensity, the memory access takes much longer time than the computation. As the batch size increases to hundreds, the activation tensor becomes larger and the system is I/O-bound again, similar to the prefill stage.

**Decoding often has longer latency than prefill:** The bottom row of Figure 4.8 shows the prefill versus decoding latency breakdown. Notably, the system only spends more time in the prefill stage than the decode stage when the batch size is equal to or greater than 16. This is because prefill only takes one iteration while decoding requires $n$ iterations to generate $n$ tokens.

**Hardware has excessive compute capacity:** When operational intensity is high, such as during the prefill or decoding stages with large batch sizes, communication overhead starts to dominate making I/O the primary bottleneck. The best compute utilization achieved by state-of-the-art GPU [9] and TPU [150] implementations are around 50% and 40%, respectively. This suggests a surplus of computing units on current hardware platforms for serving LLMs.

Since the system is primarily memory-bound in the decoding stage, which has longer latency than the prefill stage in most cases, the system favors memory architecture with a higher bandwidth. With the latest HBM3E, NVIDIA's B200 GPU [134] achieves a 8 TB/s of bandwidth per package. It also has a peak compute performance of 2250 TFLOPS. Despite significant advancements in both memory and compute capabilities, the ratio of peak bandwidth to compute performance remains relatively small. However, to improve prefill latency, optimization should mainly focus on I/O. In summary, to optimize end-to-end performance for serving LLMs on existing systems, we need **memory** with higher bandwidth, potentially coupled with fewer compute units to enhance the bandwidth-to-compute ratio, and optimized **I/O** mapping and communication strategies.

| Workload | TTFT | TBT | E2E |
|---|---|---|---|
| **Code** | 400 ms | 50 ms | 12.9 s (250 tokens generated) |
| **Conversation** | 200 ms | 50 ms | 25.2 s (500 tokens generate) |

Table 4.5: SLOs for evaluation. E2E is set to be TTFT plus the time to generate a number of tokens while meeting TBT.

### *4.5.3 Performance Evaluation with Real-World Traces*

To identify performance bottlenecks and potential optimizations, we analyze two models: Llama3-70B [122] on an 8-node system and DeepSeek v3 [45] on a 32-node system. The baseline system models H100 style GPUs, while alternative configurations explore increased DRAM bandwidth, greater systolic array (tensor core) height, and additional compute cores (SMs). All configurations maintain consistent system-level settings, including node count, interconnect links and topology, and the dynamic batching strategy. We specifically leverage chunked mixed continuous batching with a prefill block size of 2048, which improves operational intensity for decode tasks. Llama3 employs tensor parallelism, whereas DeepSeek v3 uses expert parallelism.

As input loads to an LLM system fluctuate over time, a crucial metric is whether the system can maintain SLOs under high request rates. To explore this, our trace generator produced traces for both coding and conversation applications at various input request rates, sampling from Figure 4.5.

Figure 4.9 presents P50 and P90 end-to-end latencies across different input loads for LLaMA3-70B (left) and DeepSeek v3 (right) inference. The x-axis represents input load, while the y-axis shows E2E latency normalized to the SLO thresholds in Table 4.5. Results indicate that increasing tensor core height or core count significantly improves performance, whereas boosting HBM bandwidth only provides limited benefits. This suggests that modern LLM inference systems are increasingly compute-bound rather than memory-bandwidth-bound, largely due to the effectiveness of advanced batching techniques.

Additionally, we evaluate Llama3-70B on conversation workload in Figure 4.10. We ob-

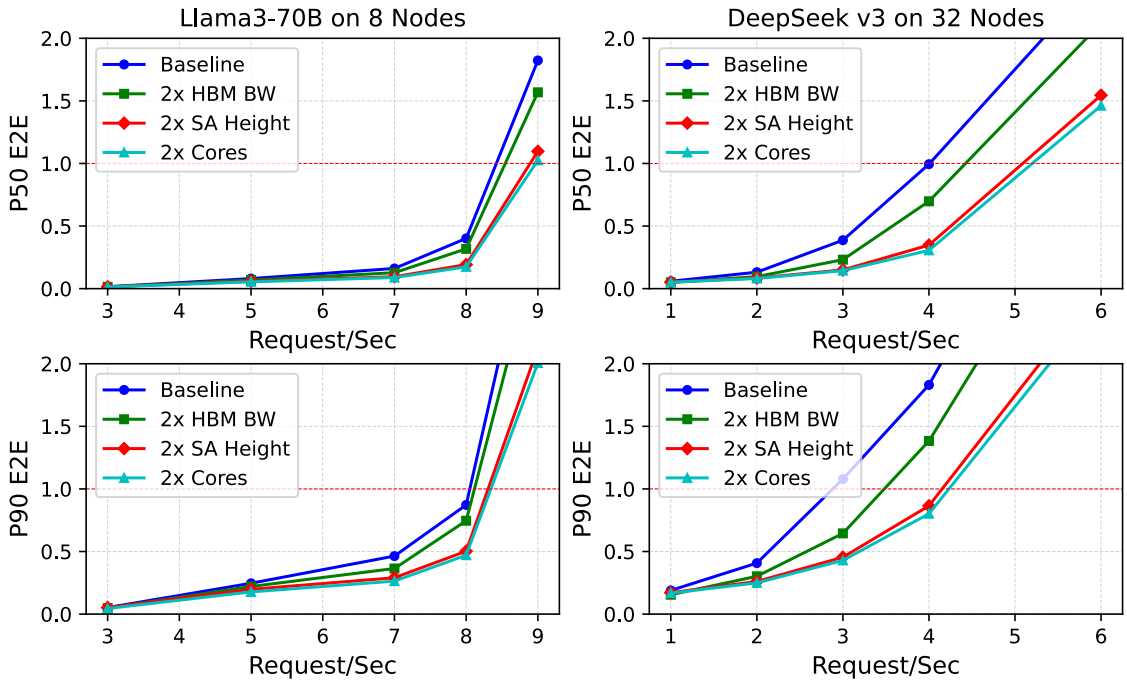



Figure 4.9: Latency metrics across input loads of Llama3-70B on 8 nodes (left) and DeepSeek v3 on 32 nodes (right) systems with different architectures.

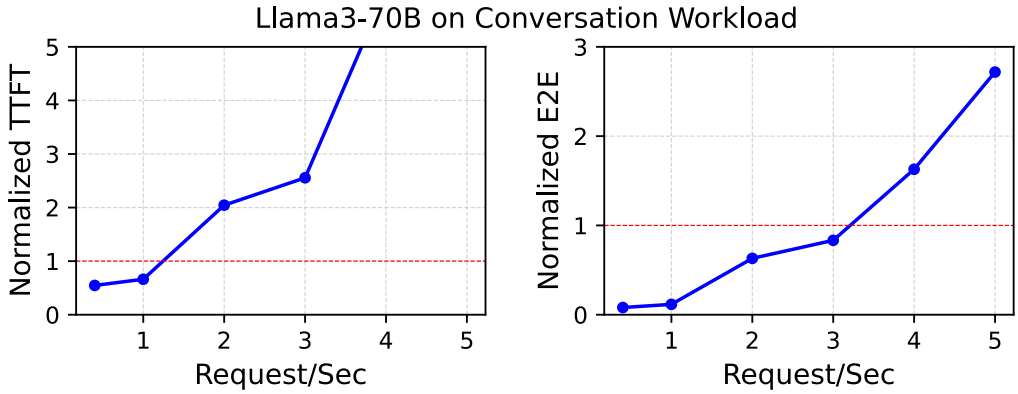



Figure 4.10: TTFT and E2E across input loads of Llama3-70B for conversation applications on a 32-node system with different architectures.

serve that the conversation workload experiences an earlier latency increase as input request rates grow. This is because conversation-based tasks typically require generating more tokens per request, causing requests to remain in the system for longer durations. Consequently, conversation applications may require greater hardware resources compared to coding applications to maintain similar SLOs.

### *4.5.4 Scalability and Efficiency Gains*

We evaluate ReaLLM's impact on simulation efficiency by comparing its performance against a baseline approach that relies exclusively on a kernel simulator like LLMCompass. As shown in Figure 4.11, simulating traces with hundreds of requests and context lengths extending to thousands of tokens requires the baseline to perform approximately $10^4$ MatMul simulations, resulting in an estimated runtime of 4,570 minutes. In contrast, ReaLLM drastically reduces this overhead by identifying 1,600 key kernels and precomputing their latencies in 729.6 minutes. Once the kernel library is constructed, trace-driven simulation takes only 27.9 minutes, leading to a 164× speedup in trace execution. Since kernel construction is a one-time process, this optimization significantly accelerates design space exploration while maintaining high fidelity in performance modeling.

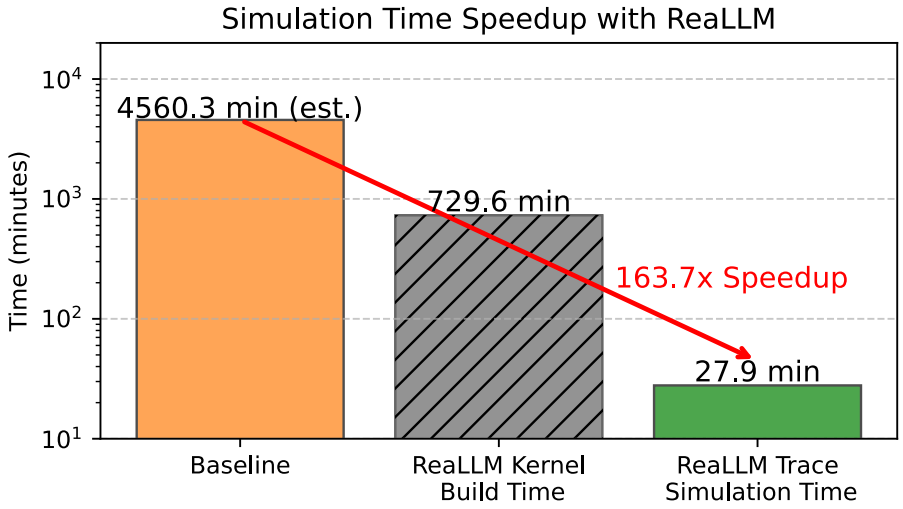



Figure 4.11: ReaLLM achieves a 164× speedup in trace simulation time compared to the baseline kernel simulator by leveraging precomputed kernel reuse.

By leveraging precomputed kernel latencies and an efficient trace-driven simulation methodology, ReaLLM transforms large-scale LLM system evaluation from an intractable computational problem into a practical and scalable process, enabling rapid architectural exploration and optimization.

### *4.6 Discussion and Conclusion*

The rapid evolution and increasing scale of Large Language Models (LLMs) have presented significant challenges in designing and deploying efficient inference systems. Optimizing these systems requires a deep understanding of the complex interplay between hardware architectures, software strategies, and workload characteristics. This chapter has introduced ReaLLM, a holistic simulation framework developed to address these challenges by providing a comprehensive platform for evaluating, analyzing, and gaining insights into LLM serving performance and cost.

ReaLLM distinguishes itself by integrating detailed, device-level kernel modeling with system-wide, trace-driven simulation, bridging a critical gap in existing evaluation method-

ologies. Its core design philosophy emphasizes a holistic view, multi-level simulation fidelity, and computational efficiency. Key contributions include its multi-level simulation approach balancing accuracy and speed, the significant runtime reduction (up to 164× in trace execution) achieved via the hypothesis-driven precomputed kernel library, and its realistic system behavior modeling using trace-driven simulation with advanced scheduling and batching. Furthermore, ReaLLM incorporates comprehensive hardware parameterization, integrated Total Cost of Ownership (TCO) modeling, and an intuitive interactive visualization GUI to provide a well-rounded analysis platform. Validation against real NVIDIA A100 and H100 GPU systems has demonstrated ReaLLM's accuracy, with an average end-to-end latency prediction error of approximately 9.07% for complex workloads like LLaMA-70B on a multi-GPU setup.

The application of ReaLLM to analyze various LLM inference scenarios, as presented in Section 4.5, has yielded several important insights. Simulations highlighted dynamic bottleneck shifts between I/O, memory bandwidth, and compute, depending on the workload phase and batch size. Notably, evaluations of modern GPU-style systems with advanced batching suggest an increasing trend towards being compute-bound rather than purely memory-bandwidth-bound at scale. ReaLLM also underscored the significant impact of workload characteristics (e.g., conversational AI vs. code generation) on system performance and the efficacy of system-level optimizations like mixed continuous batching. These findings reinforce the necessity of hardware-software co-design, for which ReaLLM provides a robust analytical platform.

While ReaLLM provides a robust and versatile simulation framework, it also has limitations that open avenues for future research. Currently, ReaLLM facilitates manual exploration of the design space; a significant future enhancement would be the integration of automated Design Space Exploration (DSE) methodologies. Continuous expansion of model and hardware support, alongside more detailed dynamic power modeling, also represent important future directions. Further enhancements to the GUI's analytical capabilities and modeling of emerging hardware technologies and LLM serving techniques will continue to increase ReaLLM's utility.

The challenge of efficiently serving ever-larger LLMs demands sophisticated tools for

deep performance and cost analysis. This chapter presented ReaLLM as a holistic simulation framework that successfully integrates detailed kernel analysis with comprehensive, trace-driven system modeling. Its unique combination of multi-level fidelity, accelerated simulation, TCO analysis, and interactive visualization empowers researchers and engineers to navigate the intricate LLM inference landscape effectively.

Validation against real hardware underscores ReaLLM's accuracy, and its application in case studies has proven its ability to reveal critical system bottlenecks and evaluate diverse architectural and software strategies. The significant simulation speedup makes ReaLLM a practical tool for extensive design exploration. As LLMs continue to reshape AI, frameworks like ReaLLM are instrumental for driving the co-design of powerful, cost-effective, and sustainable next-generation systems. ReaLLM is available as an open-source project at (TODO: GitHub Link), aiming to further democratize research and development in this critical domain and help realize the full potential of large language models.

Chapter 5

# CHIPLET CLOUD: A TCO-OPTIMIZED LLM HARDWARE ARCHITECTURE

The discussion, figures, and tables related to the Chiplet Cloud architecture presented in this chapter are based on and reprinted from our prior work published in [145].

## *5.1 Introduction*

A major contributing factor to the increase in ML capabilities comes from the unprecedented scale of the LLMs being deployed. Most LLMs used today have billions [21, 34, 197] or even trillions of parameters [51]. Serving modern generative LLMs on commodity hardware, like GPUs, is already hitting a *scalability wall*. For example, Google Search is estimated to process over 99,000 queries [129] per second while state-of-the-art GPT-3 throughput on GPUs is 18 tokens/sec per A100 [9]. If GPT-3 is embedded into every query and each query generates 500 tokens, Google would need 340,750 NVIDIA DGX servers (2,726,000 A100 GPUs) to keep up. Assuming every GPU was able to sustain 50% utilization, the average power would be over 1 Gigawatt which is enough energy to power 750,000 homes [39]. To address these scalability issues, we must design hardware systems that attain significantly better ***total-cost-of-ownership (TCO) per token*** served.

This chapter proposes *Chiplet Cloud*, a highly parameterizable chiplet-based ASIC LLM-supercomputer architecture which aims to reduce TCO per generated token [145]. The main insights behind the Chiplet Cloud architecture are shown in Figure 5.1. To address the potential bandwidth bottlenecks of LLM inference, the Chiplet Cloud architecture allows for all model parameters and KV values to be stored in a memory system called CC-MEM, a scalable on-chip memory system for Chiplet Cloud architectures. We use a finely tuned replicated chiplet accelerator module to reduce the fabrication cost as we scale the system to meet performance demands. To support models that leverage sparsity, we use a compression decoder unit which lives within the CC-MEM network to implement a *Store-as-Compressed,*

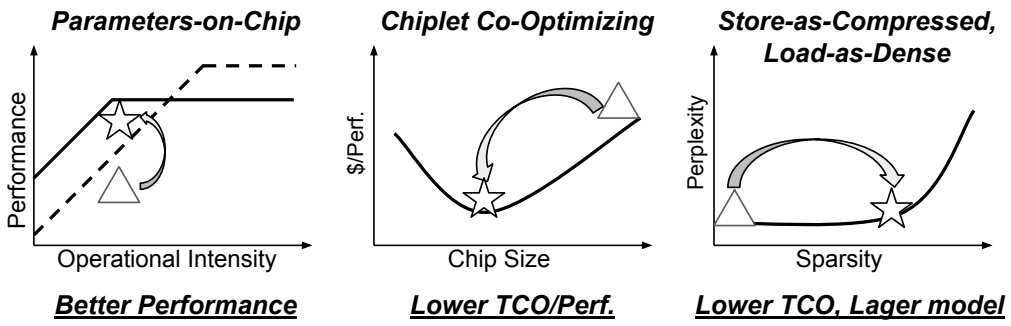



Figure 5.1: Compared to conventional systems, Chiplet Cloud (1) fits all model parameters inside the on-chip CC-MEM, greatly improving the performance; (2) co-optimizes the chip size with software mapping to reduce TCO/Perf; (3) exploits sparsity to reduce TCO and support larger models.

*Load-as-Dense* mechanism. We show these design choices win in the competition of TCO per token for serving generative LLMs but requires careful consideration with respect to the chiplet die size, chiplet memory capacity and bandwidth, and total number of chiplets to balance the fabrication cost and model performance.

To explore the massive hardware-software co-design space of Chiplet Cloud and find TCO per token optimal parameterizations, we propose a two-phase design-search methodology that fine tunes the architecture across a collection of LLM workloads. The hardware exploration phase conducts a bottom-up design space exploration of Chiplet Cloud hardware architecture from a flexible accelerator architecture up to a 1U rack mounted server architecture taking power budget, floorplan, and thermal constraints into account. The software evaluation phase then performs a detailed performance and TCO analysis of the server designs given a specific workloads while simultaneously searching for a software mapping strategy that complements the server architecture. While software mapping strategies for LLMs are now considered standard techniques for improving performance on existing hardware platforms, our design methodology flips the order and allows us to explore mapping strategies across all possible Chiplet Cloud hardware configurations for a software-hardware co-design methodology.

This chapter will first delve into the architectural details of Chiplet Cloud, followed by case studies derived from the design methodology, an evaluation of its performance and cost-effectiveness, and finally, concluding remarks.

## *5.2 Chiplet Cloud Architecture*

Specialized chip designs often focus on raw hardware performance, however this is not always aligned with cloud hardware designers whose systems are optimized for TCO per performance [85]. TCO includes both the the capital expenditure ($CapEx$) plus the operation expenditure ($OpEx$) over the lifetime expectancy of the system ($Life$), giving us the equation $TCO = CapEx + Life \times OpEx$. Optimizing TCO is therefore a balance of how much you are willing to pay for the additional performance. Aiming for improved TCO per performance, we propose a chiplet-based cloud-scale system design for LLM inference, called ***Chiplet Cloud***. The architectural breakdown of Chiplet Cloud is shown in Figure 5.2, which includes the high-level abstract architecture at different levels from the memory system up to the chiplet module, server, and cloud.

*Chiplet Cloud Memory Architecture*

The heart of Chiplet Cloud is the Chiplet Cloud Memory architecture CC-MEM (Figure 5.2 (a)). CC-MEM is a scalable on-chip memory system with the ability to sustain high-bandwidth, low-latency read and write operations. This is the main memory for each chiplet in the chiplet-cloud system which stores the model parameters, KV cache and activations.

The CC-MEM is designed to act as a drop-in replacement for DRAM memory but leverages SRAM to give us opportunities to take advantage of higher-bandwidth and lower-latency memory access for significantly better performance for the low operational intensity kernels of LLMs. SRAMs are clustered into bank groups with each bank group acting as a virtual single-port memory. Each bank group also contains a compression decode unit including a sparse tile memory.

Bank groups are interconnected using a pipelined crossbar switching network. The decision to use a crossbar network comes from the low-latency and low-global communication

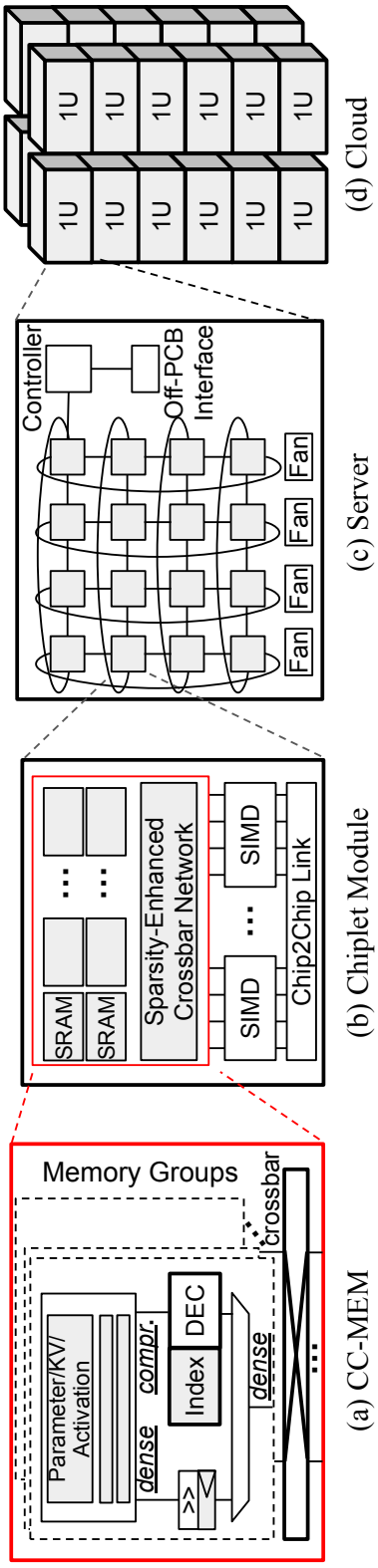



Figure 5.2: Chiplet Cloud architecture from the CC-MEM to the cloud.

power overhead while being able to achieve a 100% saturated throughput with reasonable network scheduling. The biggest downside of utilizing a crossbar network comes from their area scalability as the network scales quadratically with the radices of the network. As with many networks, this area is routing dominated. The CC-MEM is mostly SRAM; thus, there is an abundance of routing tracks available above the SRAM devices severely lessening the area overhead of crossbar network, a concept known as *NoC symbiosis* [149]. Crossbar networks also have the benefits of being simple to model both in terms of latency (pipeline depth) and congestion (bank conflicts). This allows our hardware-software co-design search space to take into account memory scheduling to ensure that we can achieve the memory access performance that is required in order to hit our target TCO/performance metrics.

The CC-MEM supports burst mode operations. Each bank group contains a simple control unit to facilitate in bursting multiple sequential read/write commands within a bank group. This control unit is programmed using simple memory mapped control status registers. Due to the highly structured nature of GEMM kernels, burst mode operations will make up a majority of the memory operations during moments of computation and will greatly reduce the burden on the compute unit to keep the memory system bandwidth at near-peak throughput.

*CC-MEM for Sparsity*

There is a growing interest in reducing LLM inference costs via model compression. Recent work [52] has shown that large models are more compressible and have significantly less accuracy drop off than small models under compression. OPT-175B [220], which has the same model architecture as GPT-3, can reach 60% unstructured sparsity with negligible increase in perplexity while requiring no fine-tuning effort. Supporting unstructured sparse model on ASIC can be challenging since the highly irregular sparsity can lead to unpredictable data assess and compute patterns. Simultaneously, sophisticated decoder and on-chip network architecture for sparse data dispatching can add significant area overhead. To address these issues, we implement a *Store-as-Compressed, Load-as-Dense* mechanism into the CC-MEM architecture. Models are compressed using a tile-based compressed sparse row format [131]

and stored in the CC-MEM in this sparse format. However, load access patterns and data appear as if the data was stored dense. The methodology is based on the insight that TCO/Token of our proposed system will be primarily limited by the on-chip memory size, rather than memory bandwidth and compute unit utilization. Reducing the required memory size will be the first priority when supporting sparse models. Using this mechanism, the compute units are *sparsity-agnostic* and do not require any special design, reducing area overhead and increasing flexibility.

To support this methodology, each bank group within the CC-MEM contains a compression decode unit. Data in CC-MEM can be in raw dense formats or sparse compressed formats. The decode units are controlled using a simple set of memory mapped CSRs similar to the burst mode CSRs. Data sent over the network is always in dense formats, allowing any network attached compute units to be completely agnostic to the format the data is stored in. Compressed data ultimately has a lower bandwidth than dense data. This is because dense data and sparse data are both stored in the same SRAM banks which have the same peak bandwidth but sparse data has additional bits per word.

Figure 5.3 shows an example design of the compression decoder unit. In this example, the sparse matrix is divided into tiles of shape (32, 8). Like the standard compressed sparse row format, non-zero values (NZV, 16 bits) in a tile are encoded using a 5-bit row index ($r$) and a 3-bit column index ($c$), forming a 24-bit sparse word stored in data memory. Tile indexes are stored in a separate index memory, which is placed together with crossbar routing tracks to minimize area overhead. To read sparse data, the decoder sends a tile read request to the index memory and receives the initial address and end address of the NZVs in a tile. The decoder then reads data memory at a rate of up to 8 sparse words per cycle and writes them to a double-buffer. Depending on the row index and column index, zeros are inserted accordingly to form the original dense tile. The unit can constantly output 8 dense words per cycle.

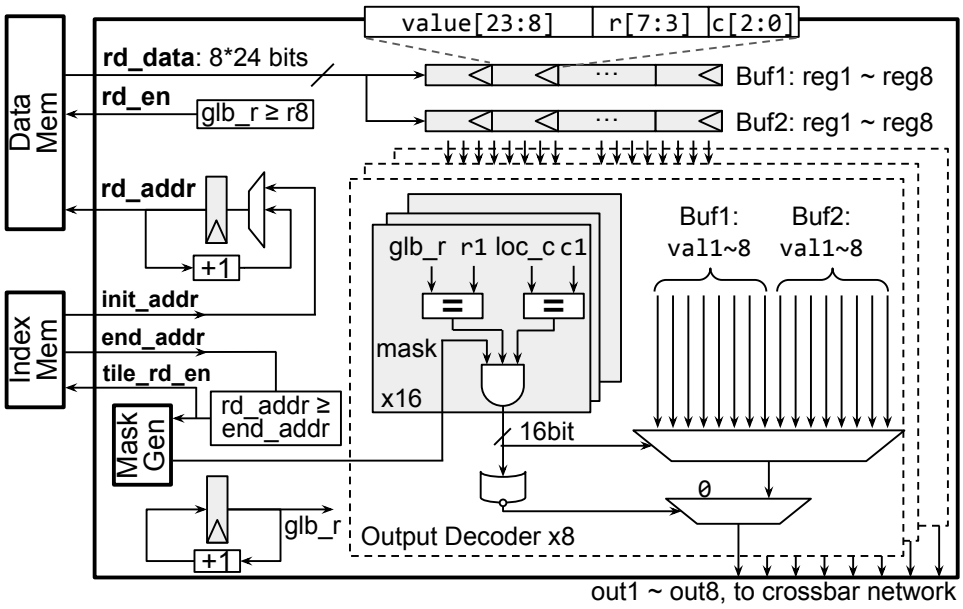



Figure 5.3: Compression decoder unit in CC-MEM.

*From Chiplet to Cloud*

Figure 5.2 (b) shows a LLM accelerator chiplet module. Inside the chiplet, multiple SIMD cores are attached to a CC-MEM. Compared to a fully custom compute units, the SIMD cores are more flexible with very few limitations on the types of kernels that can be efficiently supported, which is essential for supporting the various activation functions and embeddings found in modern LLMs.

In Chiplet Cloud, a single chiplet module functions as a discrete package, with multiple chiplets interlinked across the board. Advanced package-level solutions such the silicon interposers [92] can provide higher signal density for high bandwidths in-package communication. However, it has a limited reach and adds more cost. In contrast, our Chiplet Cloud design adopts a *board-level* organic substrate chiplet approach, aligning with specific communication requirements. Given the large scale of modern LLMs, running the model within a single package of chiplets is often impractical. Partitioning into multiple packages or even across servers becomes a necessity. This partitioning mandates collective operations, such as

all-reduce, to occur across packages. Since the conventional ring all-reduce implementation is limited by the slowest link among nodes, the in-package high-speed links do not provide much help in this case. Compared to conventional package-level chiplet, the board-level chiplet architecture eliminates cost of advanced packaging.

Each Chiplet Cloud server (Figure 5.2 (c)) contains a printed circuit board (PCB) with multiple chiplets, a controller and an off-PCB network interface. The controller, which can be an FPGA or a microcontroller, dispatches remote procedure calls from off-PCB interface to all chiplets. Chiplets are connected together via a 2D torus on-PCB network, which is able to accommodate the many different mapping strategies that we might need to implement to efficiently run different models. Candidates for chip-to-chip interfaces can be custom-designed links such as NVIDIA ground-referenced signaling GRS links [198, 151], Google TPU's Inter-Core Interconnect [86], Graphcore's IPU-links [102], or high-speed PCI-e which has been widely used as interconnects for many deep learning chips [172, 152, 121]. Off-PCB interfaces could be 10/100 Gigbit Ethernet or InfiniBand, enabling communication between adjacent servers.

*Design Space Discussion*

The design space of Chiplet Cloud is a balancing act that includes many different architectural parameters across the entire system that greatly impact the resulting TCO/Token. Some aspects include (1) *Chiplet Module Size:* small chips benefit from higher yields while incurring more per-chip overhead; (2) *Per Chiplet Memory Size:* more memory on chips means few chips required but few FLOPS per chip; (3) *Per Chiplet FLOPS:* more FLOPS increases performance while requiring higher memory bandwidth, resulting in a larger memory crossbar; and (4) *Software Mapping:* the trade-off between different parallelisms affects utilization and interconnect data communication. Since all of these aspects are tightly coupled, a comprehensive design methodology is critical to optimize the end-to-end performance and TCO.

## *5.3 Case Studies*

To evaluate Chiplet Cloud, we performed a case study on eight language models, including GPT-2 [153], Megatron-LM [170], GPT-3 [21], Gopher [154], MT-NLG [173], BLOOM [19], PaLM [34] and Llama-2 [197]. Details about these models are shown in Table 5.3. All studies were conducted on publicly released data, such as model architecture hyper-parameters, and do not use actual weights.

*Design Space Exploration*

We performed a thorough design exploration under 3 different context length scenarios (1024, 2048 and 4096) and on batch sizes from 1 to 1024. This exploration results in over 2 million valid design points for each model. Each design point combines the result from both hardware exploration and software evaluation, which includes hardware design (chip and server), software mapping (tensor parallelism size, pipeline parallelism size, batch size and micro-batch size), cost (OpEx and CapEx) and performance (latency and throughput), etc.

Table 5.3 shows the TCO/Token optimal Chiplet Cloud designs for each model in our case study. We found that all TCO-optimal designs are targeting batch sizes greater than or equal to 32. Large batch sizes are good for utilization in FC layers but will require additional silicon for memory to account for a larger KV cache. This means we either need bigger chips which greatly increase CapEx, or more chips which generate more inter-node traffic and hurt throughput. This will either results in larger chips which will sharply increase the CapEx as our silicon per chip gets larger and yield gets worse, or it will generate systems with a larger number of chips increasing the amount of inter-chip communication and diminishing the end-to-end performance. Finding batch sizes that balance each factor is essential to achieve good TCO/Token but is challenging to find. Each optimal design points across our 8 models all have different chip, server designs, and mapping strategies demonstrating the importance of our design methodology—every aspect of the system affects performance and cost and are sensitive to the requirements of the workload.

While the workload does impact the optimal Chiplet Cloud configuration, this doesn't mean that a Chiplet Cloud instance can only run a single model. Additional discussion on

| Model | **GPT-2** | **Megatron-LM** | **GPT-3** | **Gopher** | **MT-NLG** | **BLOOM** | **PaLM** | **Llama-2** |
|---|---|---|---|---|---|---|---|---|
| Parameters (B) | 1.5 | 8.3 | 175 | 280 | 530 | 176 | 540 | 70 |
| $d_{model}$ | 1,600 | 3,072 | 12,288 | 16,384 | 20,480 | 14,336 | 18,432 | 8,192 |
| Layers | 48 | 72 | 96 | 80 | 105 | 70 | 118 | 80 |
| Die Size ($mm^2$) | 60 | 40 | 140 | 100 | 160 | 120 | 100 | 80 |
| MB per Chip | 32.8 | 27.0 | 225.8 | 151.0 | 198.0 | 137.5 | 95.0 | 82.5 |
| TFLOPS per Chip | 5.60 | 2.87 | 5.50 | 4.83 | 6.32 | 7.02 | 12.07 | 7.62 |
| BW per Chip (TB/s) | 2.80 | 2.29 | 2.75 | 2.41 | 4.21 | 3.51 | 1.51 | 1.90 |
| Chips per Server | 128 | 144 | 136 | 160 | 160 | 152 | 120 | 72 |
| Number of Servers | 24 | 8 | 96 | 80 | 105 | 70 | 118 | 80 |
| Tensor Parall. Size | 64 | 144 | 136 | 160 | 160 | 152 | 120 | 72 |
| Pipeline Parall. Size | 48 | 8 | 96 | 80 | 105 | 70 | 118 | 80 |
| Batch Size | 128 | 8 | 256 | 128 | 128 | 128 | 1024 | 512 |
| Micro-Batch Size | 2 | 1 | 2 | 2 | 1 | 2 | 8 | 4 |
| Max Context Length | 16K | 256K | 8K | 16K | 16K | 16K | 2K | 4K |
| Tokens/Sec per Chip | 473.3 | 69.7 | 8.1 | 4.3 | 2.7 | 8.6 | 7.0 | 26.5 |
| TCO/1M Tokens ($) | 0.001 | 0.008 | 0.161 | 0.228 | 0.521 | 0.141 | 0.245 | 0.046 |

Table 5.1: TCO/Token optimal Chiplet Cloud systems for different language models.

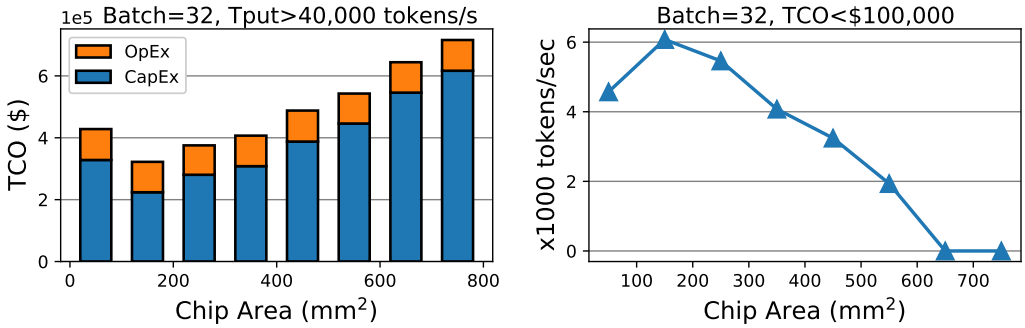



Figure 5.4: Proper chip size can reduce the fabrication costs (CapEx) without compromising performance as much. Left: For a given throughput requirement, chips with a size of less then 200 $mm^2$ have lowest TCO. Right: For a given TCO budget, chips with a size between 100 $mm^2$ to 200 $mm^2$ achieve the best throughput.

the impact of running non-optimized models and how a multi-model objective optimization perform can be found in the next section.

*Design Insights*

**How chip size affects TCO and performance.** Figure 5.4 shows the results of GPT-3 in two different scenarios. On the left is how we should choose the die size to lower TCO for a given minimum throughput requirement. Compared to chips over 700 $mm^2$, which is the size of many traditional large monolithic chips, a chip around 200 $mm^2$ reduces TCO by about 2.2× and still meets the throughput constraint. We also find the CapEx exceeds 80% of TCO for most designs. The right side of Figure 5.4 shows chips with a size between 200 mm2 to 300 mm2 achieve the best throughput for a given TCO budget. This shows that proper chip sizing can effectively reduce TCO without compromising performance.

**How the batch size affects TCO/Token.** Figure 5.5 shows the TCO/1K Tokens versus batch size across 4 models and 3 context lengths. When the batch size is increased from 1, TCO/Tokens improves due to increases in compute utilization by providing more opportunities to exploit pipeline parallelism. As the batch size continues to increase, the utilization will reach a peak. For the traditional multi-head model, more silicon is required

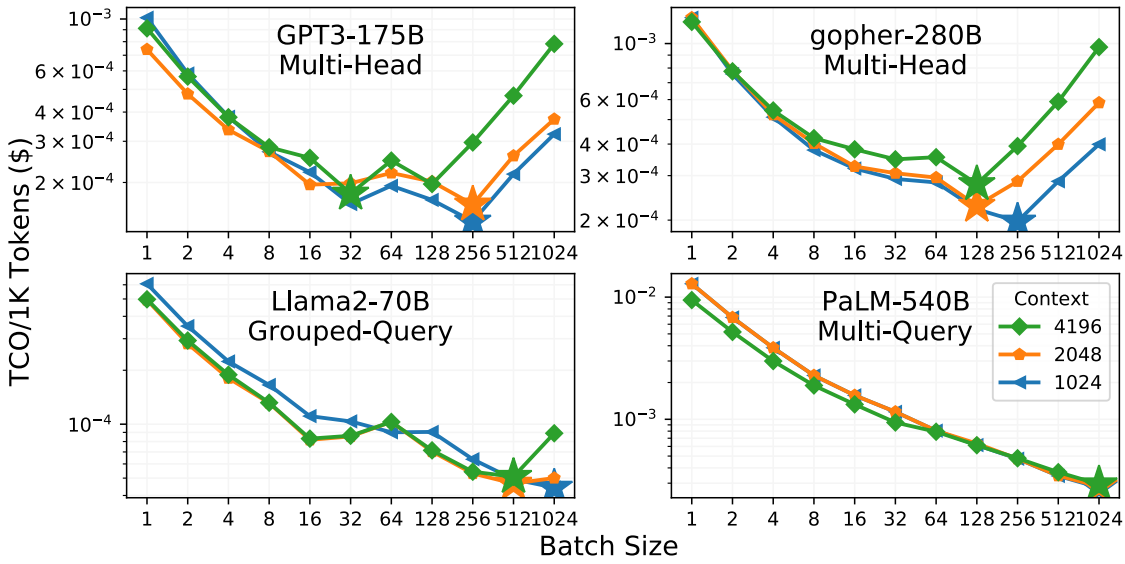



Figure 5.5: The optimal TCO/Token under different batch sizes. Small batch requires less silicon, and large batch benefits weight reuse. The optimal batch size for multi-head models is between 32 to 256, while the multi-query and grouped-query models are able to maintain a near-optimal TCO/Token at batch size 1024.

for KV cache in large batch size and long contexts, which significantly increases TCO/Token. Chiplet Cloud supports batch sizes up to 128 with near-optimal TCO/Token for these models. PaLM adopts multi-query attention [167] and Llama-2 adopts grouped-query attention [6], where key and value are shared across all or some groups of attention heads, which reduces the size of the KV cache by a factor of number of heads. For these models, Chiplet Cloud supports batch sizes up to 1024 with near-optimal TCO/Token. The cost of longer contexts is negligible, especially when the batch size is not too large.

**How the mapping strategy affects TCO/Token for a given batch size.** Figure 5.6 shows that when the number of pipeline stages $p$ (i.e. the pipeline parallelism size) is close to the batch size, the system utilization is the largest and TCO/Token is optimal. When these two numbers are similar, the system can take full advantage of pipeline parallelism with a micro-batch size of 1, so the number of micro-batches is also close (if not equal)

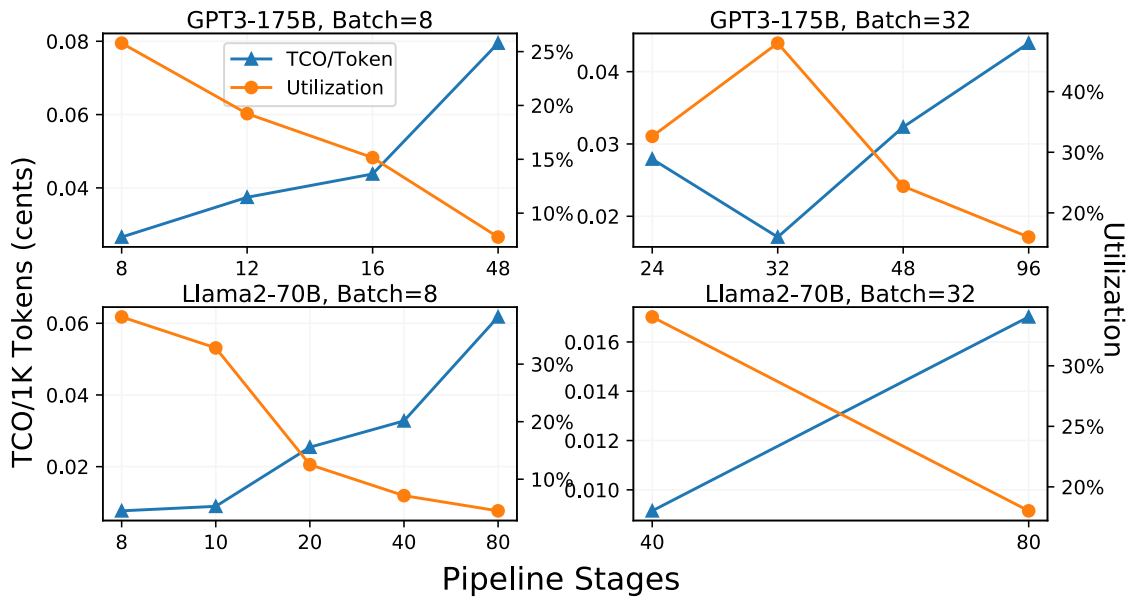



Figure 5.6: Pipeline stages sweeping for different models and batch sizes. The number of pipeline stages close to the batch size usually achieves the highest utilization, resulting in the optimal TCO/Token.

to the pipeline stage [9]. This helps balance the latency of micro-batches passing through all pipeline stages and pipeline stages completing all micro-batches. Specifically, when $p$ is too small, performance is hindered by pipeline stage latency, whereas an excessively large $p$ results in limitation by microbatch latency.

### *5.4 Evaluation*

In this section, we evaluate the performance and cost of Chiplet Cloud for serving large language models. The key metric we are targeting is *TCO/Token*. TCO/Token is measured as cost per token generated and is the key factor in the ability to democratize LLMs. One of the most popular business models for generative LLMs is also to charge users per generated token. Lower TCO/Token not only adds more profit margins, but also makes LLMs more approachable. We compare Chiplet Cloud to state-of-the-art GPU and TPU cloud implementations. We also evaluate the sparsity support and flexibility of Chiplet Cloud architectures.

*Comparison with GPUs and TPUs.*

We compare optimal Chiplet Cloud designs from the previous section to state-of-the-art A100 GPU [9] and TPUv4 [150] implementations. Neither work is specifically optimized for TCO/Token. For our comparison, we choose the throughput optimal result for GPU, and the utilization optimal result for TPU, which are key indicators that you are close to TCO/Token optimal. Compared to GPU and TPU clouds, our design achieves up to 106.0× and 19.9× TCO/Token improvement on GPT-3 and PaLM 540B respectively. TCO for GPUs and TPUs are based on the best cloud rental price we could find [37, 113].

Adding the NRE of Chiplet Cloud ($35M, estimated based on the NRE model from Moonwalk [98]), we show the actual cost improvement in Figure 5.7. As the number of tokens expected to be generated (x-axis) grow, NRE is greatly amortized and Chiplet Cloud gains more improvement over GPU and TPU. Compared to A100 GPU and TPUv4 clouds, at the scale of Google search (99,000 queries per second [129], and assuming 500 tokens per query), Chiplet Cloud achieves 97× and 18× improvement on (TCO+NRE)/Token, respectively. We also add variance to 2 inputs that are difficult to accurately estimate,

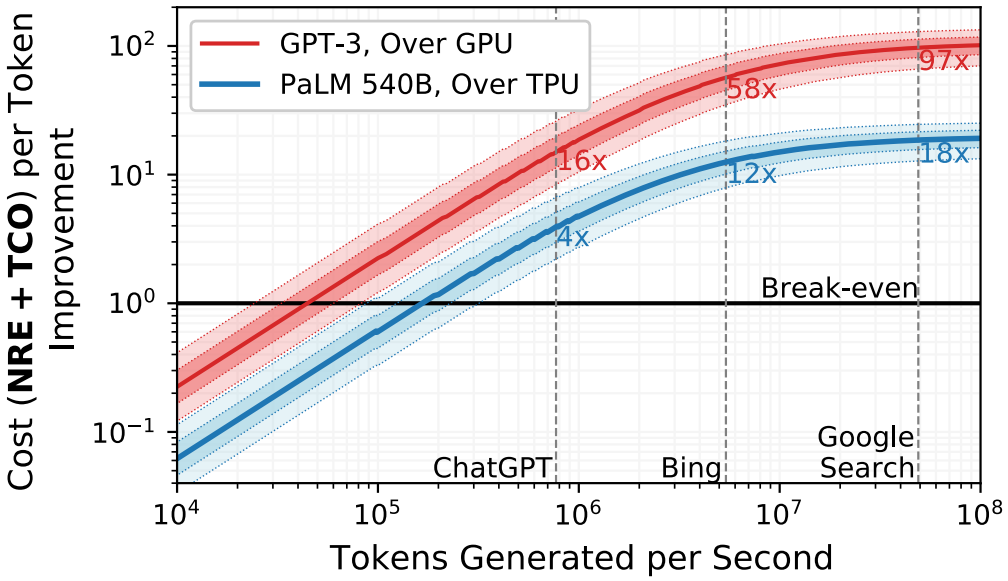



Figure 5.7: Compared to A100 GPU and TPUv4, Chiplet Cloud can achieve over 97× and 18× improvement in (NRE+TCO)/Token on GPT-3 and PaLM 540B, respectively. The light and dark shaded regions represent the results under ±30% and ±15% input variance.

those being the TCO of GPU and TPU clouds, and the NRE of Chiplet Cloud. With a ±30% variance of these inputs, Chiplet Cloud is still expected to maintain a 66× to 129× improvement over GPU, and 12× to 24× improvement over TPU.

Figure 5.8 shows the breakdown of TCO/Token of Chiplet Cloud over GPU and TPU. Some of the improvement in TCO comes from building the silicon instead of renting it. To analyze the impact of owning a chip, we feed the chip and server specifications of A100 and TPU v4 into our TCO model. The results show that owing the chip saves 12.7× and 12.4× in TCO/Token. Note that the actual savings should be less than this, as our model does not include the cost of liquid cooling and advanced packaging, which are critical for TPUs and GPUs but not required for Chiplet Cloud. We see that our specialized memory system improves TCO/Token by 5.1× and 1.5× over GPUs and TPUs, while die sizing improves it by an additional 1.3× and 1.1×. Compared to GPUs, the 2D weight-stationary layout in feed-forward network and the larger batch sizes lead to a 1.1× and 1.2× improvement

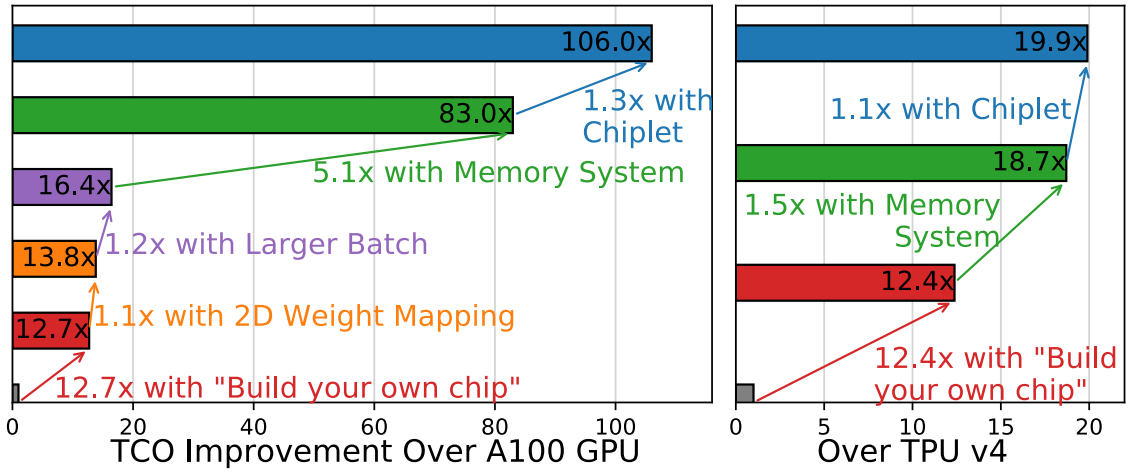



Figure 5.8: TCO/Token improvement breakdown over GPU and TPU.

respectively. Both of these optimizations are supported in the TPU implementation.

In Figure 5.9, we compare the architectural benefits of Chiplet Cloud versus TPU v4 [150] using our model for the TPU's TCO. Chiplet Cloud is more efficient at most batch sizes and achieves a TCO/Token improvement of up to 3.7× at batch size 4 as the high-bandwidth CC-MEM benefits from low operational intensity.

*Sparse Models Evaluation*

We evaluate the sparse models in Figure 5.10. The top plot compares TCO and perplexity of OPT-175B [220] under different weight sparsities. The perplexity values are from SparseGPT [52]. The blue bars show the change in TCO/Token compared to using the non-compressed dense model. At low sparsity (such as 10% and 20%), TCO/Token increases because it requires more memory to store compressed format encoding overhead. 60% sparsity represents a sweet spot where the perplexity of the model is only marginally above that of the dense model while attaining a 7.4% improvement in TCO/Token. Additional sparsity continues to give additional improvements in TCO but the model perplexity starts to increase rapidly. Chiplet Cloud also supports larger models with sparsity. The bottom of Figure 5.10 shows that under the same system configuration, Chiplet Cloud is able to

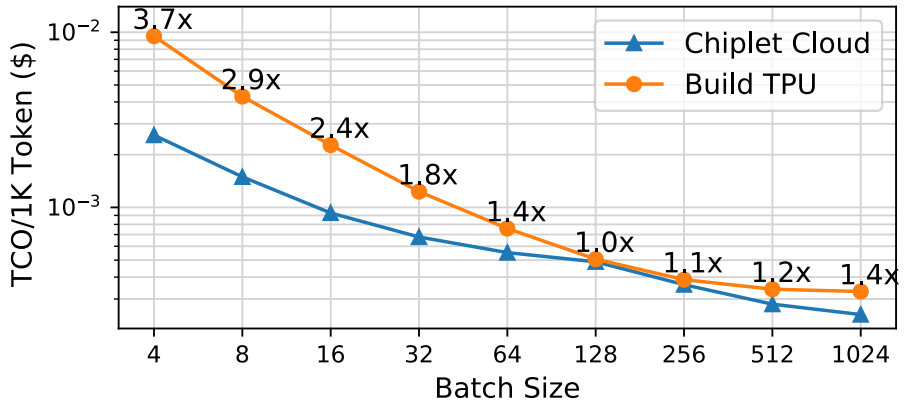



Figure 5.9: Chiplet Cloud is more efficient than TPU v4 at most batch sizes, especially for small batch sizes. TPU performance is from [150] with and TCO is from our model.

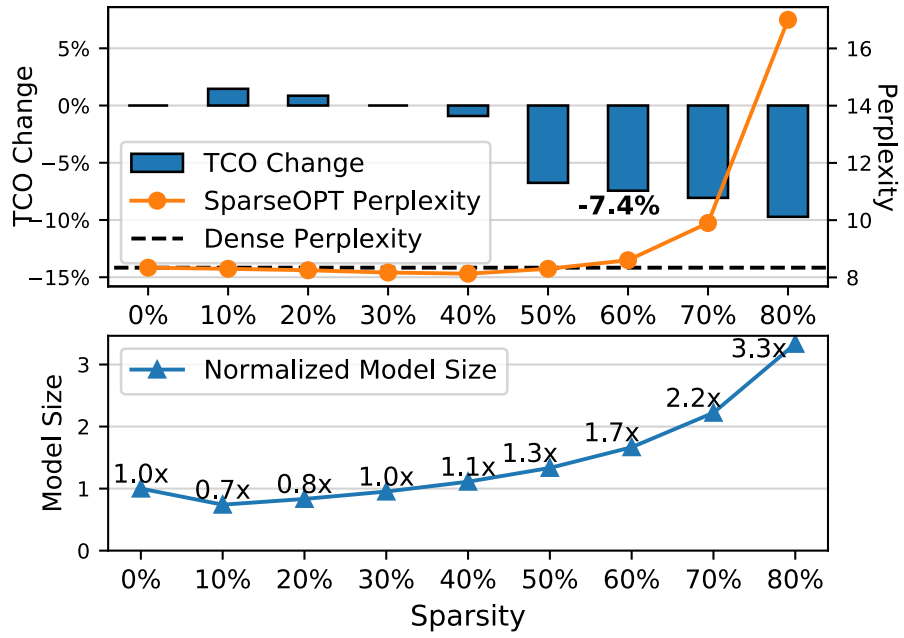



Figure 5.10: Top: TCO/Token and perplexity (from SparseGPT [52], lower is better) of OPT-175B under different sparsity. Chiplet Cloud can further reduce 7.4% of TCO/Token at 60% sparsity with negligible increase in perplexity. Bottom: Chiplet Cloud supports a 1.7× larger model with a sparsity of 60%.

support models with 1.7× parameters at a sparsity of 60%.

*Chiplet Cloud Flexibility*

Flexibility is one of the main limiting factors for large-scale deployment of ASIC supercomputers. ASIC designs with higher flexibility are believed to have longer lifetimes and thus easier to amortize the NRE costs. The main flexibility of Chiplet Cloud depends on the flexibility of chip design, which usually dominates in NRE. It is feasible to redesign servers and software mapping for different generative language models using the same chip. Since LLM scaling changes the number of parameters, while the operational intensity usually remains the same, Chiplet Cloud is able to support LLMs of larger sizes by adding chips. LLMs may also have different element-wise operations, such as different activation functions and positional embeddings, our highly programmable SIMD cores are able to support all of these variations.

By adjusting the number of chips and optimizing the server and mapping, one chip design can to run models of different sizes without sacrificing too much TCO/Token. In Figure 5.11, we show the impact on TCO/Token when mapping a chip to different models. We first show 3 model-optimized chip designs in blue, orange and green bars for Llama2, Gopher, and GPT-3, respectively. When running different models, it only increases TCO/Token by 1.1× to 1.5× compared to the corresponding model-optimized design. When flexibility comes as the first priority, one can also set a multi-model optimization for the chip design. The red dashed box shows a design optimized for the geometric mean of TCO/Token on all 8 models, achieving an average overhead of only 0.16× compared to the 8 single-model optimized designs. The red dots represent the number of chips used for each model. This demonstrates Chiplet Cloud has the flexibility to support various LLMs.

## *5.5 Conclusion*

This chapter presented Chiplet Cloud [145], a chiplet-based ASIC LLM-supercomputer architecture designed to achieve unprecedented TCO/Token for serving large generative language models. The architecture's core tenets include the CC-MEM on-chip memory system to eliminate memory bandwidth limitations, an integrated compression decoder supporting sparse

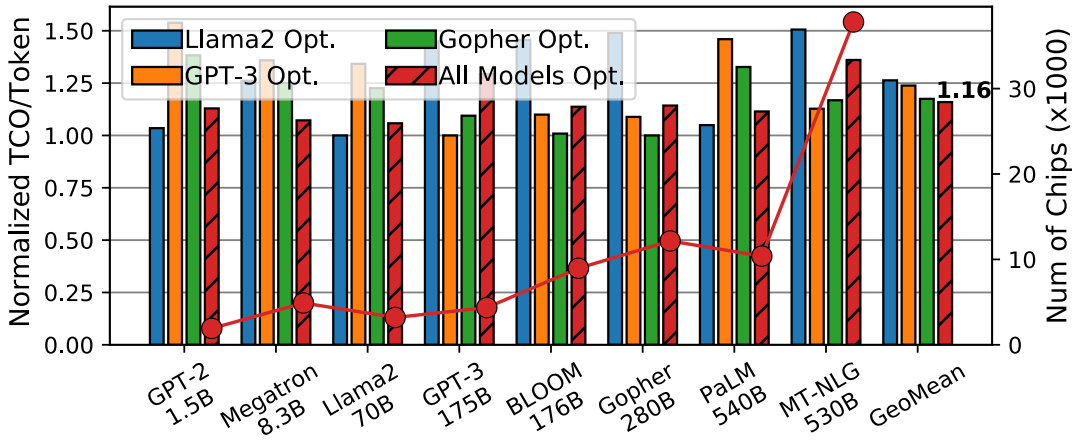



Figure 5.11: A Chiplet Cloud chip design is flexible to run models of different sizes via scale-up. Comparing to the model-optimized design, chip optimized for other models has TCO/TOken of 1.1× to 1.5× (blue, orange and green bar). One can also optimizes the chip for multi-model (dashed red bars) at only 1.16× TCO/Token on average, and the number of chips used is shown in red dots.

models via a Store-as-Compressed, Load-as-Dense mechanism, and careful moderation of chiplet die size to optimize system costs.

Case studies on eight diverse LLMs demonstrated the effectiveness of this approach. The resulting Chiplet Cloud systems achieved remarkable TCO/Token improvements, up to 97× better than rented GPU clouds and 18× better than rented TPU clouds for comparable workloads. The architecture also showcased robust support for sparse models and inherent flexibility to run various LLMs efficiently with a single chiplet design.

The significant NRE associated with ASIC development is shown to be justifiable given the enormous scale and operational costs of current LLM deployments. By substantially reducing the TCO per generated token, Chiplet Cloud offers a viable path towards making advanced AI capabilities more accessible and economically sustainable. We believe architectures like Chiplet Cloud, born from a holistic TCO-driven design philosophy, represent the future for democratizing modern and future large generative language models.

# Chapter 6

# CHRONOSTACK: A 3D-MEMORY ARCHITECTURE FOR LONG-CONTEXT LLM

## *6.1 Introduction*

The ability of Large Language Models (LLMs) to process increasingly long context lengths, now reaching up to 1 million tokens in models like Google's Gemini 1.5 [195], is crucial for tasks requiring deep contextual understanding. However, extending context length significantly strains hardware due to the attention mechanism's computational and memory demands. Beyond context length, the ratio of input to output tokens also critically affects hardware efficiency. *High-ratio* prompts (large input, small output, e.g., summarization) are generally more compute-bound, while *low-ratio* prompts (small input, large output, e.g., content generation) become more memory-bandwidth-bound as more tokens are generated.

Traditional GPU architectures, with 2.5D integrated High Bandwidth Memory (HBM) [133], face the *memory wall*: compute throughput (FLOPs) outpaces memory bandwidth growth, making them inherently more efficient for high-ratio requests. To better serve memory-bound low-ratio workloads, alternative architectures are needed. SRAM-only systems like Chiplet Cloud [145], Groq LPU [1], and Taalas [48] offer higher bandwidth by keeping parameters and KV-cache on-chip. However, SRAM's lower density necessitates many nodes for large LLMs, leading to significant inter-node communication overheads and concerns about long-term scalability as LLM and context sizes grow against slowing SRAM scaling.

Vertically stacked 3D memory, particularly with direct bond interconnects (hybrid bonding) [54], presents a promising path to overcome the memory wall by offering substantially higher bandwidth through wider parallel data lines between stacked memory and logic dies. Nevertheless, integrating 3D memory introduces challenges: limited stack capacity, increased power density leading to thermal concerns, and area overhead from Through-Silicon Vias (TSVs).

We argue that despite these obstacles, 3D DRAM can be effectively exploited in LLM inference accelerators to enhance the end-to-end latency of long-context, low-ratio workloads without compromising the performance of short-context or high-ratio tasks. This chapter will first introduce Time-Multiplexed KV-Prefetching, a novel KV-cache prefetching technique designed to maximize the utility of 3D memory bandwidth, especially in capacity-constrained scenarios, by time-multiplexing the 3D memory. It then proposes ChronoStack, a hybrid-bonded 3D memory integrated architecture tailored for LLM inference. Last, it evaluates ChronoStack employing Time-Multiplexed KV-Prefetching across a diverse range of context lengths and workload ratios.

## *6.2 KV-Prefetching to Address Memory Bottlenecks*

Generative large language models (LLMs) have been pushing the high performance computing (HPC) community in many directions. The amount of computation, memory capacity, and memory bandwidth requirements are continuously being pushed by ever growing models. Various batching techniques, such as mixed continuous batching, have improved system utilization, often making inference systems compute-bound during most operations. However, certain operations, such as the attention mechanism in decode tasks, exhibit low operational intensity and rely heavily on memory bandwidth. This dependency grows more pronounced as context lengths increase, significantly impacting overall performance. With the emergence of 3D die stacking in commercial products, vertically integrated memory systems, such as 3D-stacked DRAM, present a promising solution to address these memory bottlenecks, particularly for workloads with longer context lengths. However, adopting 3D-stacked architectures introduces trade-offs that may limit their effectiveness in certain scenarios. The remainder of this section examines the memory bottlenecks in LLM inference systems utilizing mixed continuous batching and outlines our approach to leveraging 3D-stacked memory with Time-Multiplexed KV-Prefetching to enhance performance.

### *6.2.1 Bandwidth Bottleneck: Batching Does Not Help Attention, Bandwidth Does*

The idea of batching is to group multiple requests together and compute them simultaneously while keeping their results separate. This allows systems to exploit data reuse,

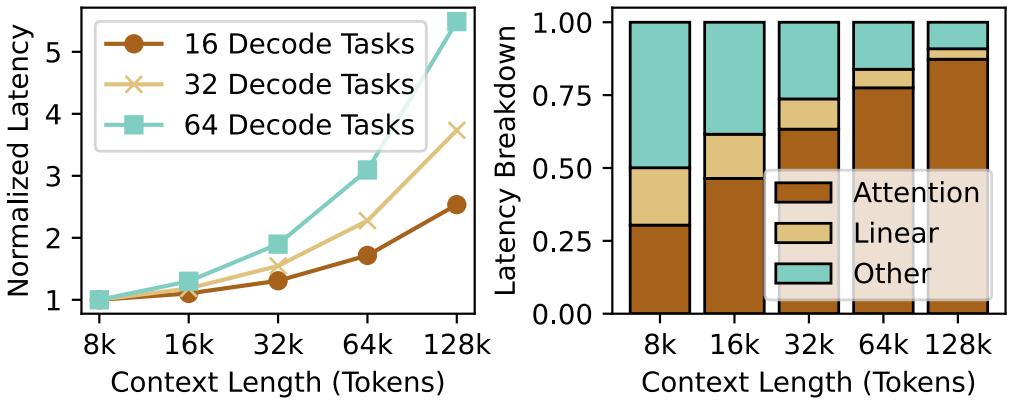



Figure 6.1: The normalized latency and breakdown of a chunked mixed continuous batching kernel. Prefill chunk size set to 512. Left: As the context length increases, benefits of batching decode tasks diminishes even with a prefill chunk helping the operational intensity of linear operations. Right: Breakdown for 32 decode tasks. As the context length grows the attention layer.

particularly with the learned model parameters since all requests use the same weights increasing the operational intensity of the computation and making the system more compute bounded during the *linear* operations. However, this advantage does not extend to *attention* operations, where the query, key, and value matrices in attention are unique for each user, preventing data sharing across batches.

Figure 6.1 shows the latency of an LLM inference iteration using chunked mixed continuous batching with a prefill chunk of 512 and various number of simultaneous decoding tasks. During the linear layers, the prefill chunk size acts as additional batching thus we are simultaneously computing $512 + D$ tokens where $D$ is the current number of active decode tasks being processed. We can see that as the context length grows, the amount of time we spend in the attention operations starts to completely dominate the computation latency. This is why batching has little to impact on the throughput of the machine (doubling the batch size nearly doubles the latency).

This motivates the need for higher bandwidth memory solutions. Currently, systems

improve their memory bandwidth by scaling out the number of devices working on the same operation, sharding the workload and then combining the results through all-reduction or all-gather operations. While effective, there are limitations to how far a system can scale-out a single operation without the data communication of the all-reduction or all-gather operations starting to dominate the runtime. Furthermore, scaling-out has a massive capital cost.

**Insight 1:** *While faster memory bandwidth will not eliminate the need for scaling-out, there is always going to be a want for faster single device memory bandwidth to address the attention operator overheads.*

### *6.2.2 Capacity Bottleneck: KV Cache Grows Super Linear With Context Length*

Thanks to chunked mixed continuous batching (C-MCB, refer to Section 2.3.2), the system is typically compute-bound during linear operations when the request rate is sufficiently high. This is because we can mix a prefill chunk with the current decode tasks. As a result, we focus on using 3D die-stacked memory specifically for the KV cache. In LLMs, the KV cache size for a given user scales linearly with the current context length. The following equation calculates the size of the KV cache for a given user:

$$KV_{tokens} = 2 * N_{layers} * D_{head} * N_{kvhead} * L_{ctx} \tag{6.1}$$

Where $D_{head}$ is the dimension per head, $N_{kvhead}$ is the number of KV heads, and $N_{layers}$ is the number of layers all of which are model specific parameters. $L_{ctx}$ is the current context length of the user.

As shown by the equation, the number of tokens in the KV cache is linear with respect to $L_{ctx}$. However, Figure 6.2 presents a different trend. The left side of Figure 6.2 demonstrates that, for a constant request rate, the peak number of active requests being processed increases as the context length grows. This occurs because requests with longer context lengths require more iterations to complete. The right side of Figure 6.2 illustrates that this phenomenon, combined with the larger KV cache size needed to support these longer context lengths, results in a rapid increase in KV cache size as the context length expands.

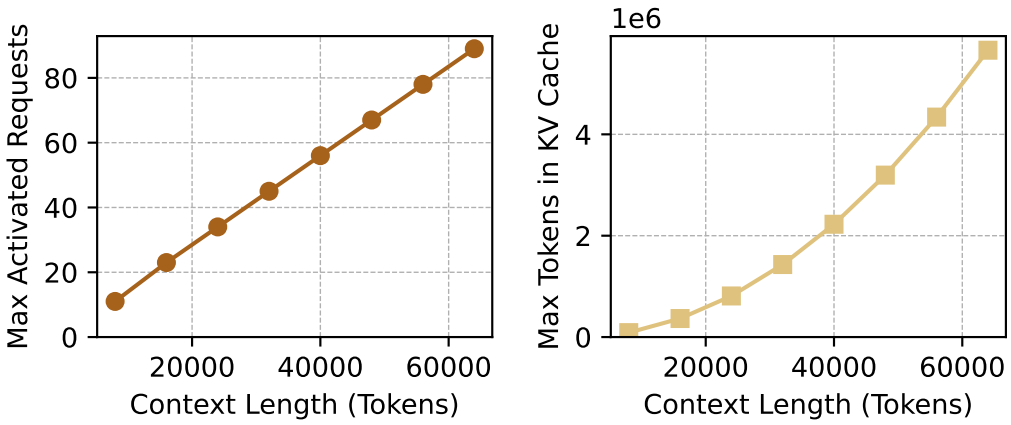



Figure 6.2: Left: For a constant request rate, as the context length grows so does the peak number of active requests being processed, because longer request it takes more iterations to finish. Right: KV Cache grows super linear with context length since there more active requests and each request is longer.

Currently, products that use memory-on-compute vertically stacked memories [209] only use a single memory layer on top of the compute die. Using a memory density of 11.7 MB/mm$^2$ [217] a near full reticle chip of approximately 800 $mm^2$ has a max capacity of 9.3 GB. Ideally, the KV cache can simply reside in this 3D memory, however this can be very limiting. Take a system running OPT-175B on 24 chips each with a 9 GB 3D memory for the KV cache. If there are 64 users mapped to this system each user can have an average context length of just 768 tokens before running out of that 9 GB memory.

This motivates the need for larger memory capacity for the KV cache. Especially since in practice, the KV cache size grows super linearly with respect to the context length. But the capacity of 3D memory systems is currently limited as only single die layer for memory on top of compute are currently commercially available, and exceeding that capacity is relatively simple.

**Inisght 2:** *The benefits that we get from the improved bandwidth of 3D memory is limited by its max capacity, leading to a relatively small number of workload scenarios where we can take advantage of the improved bandwidth..*

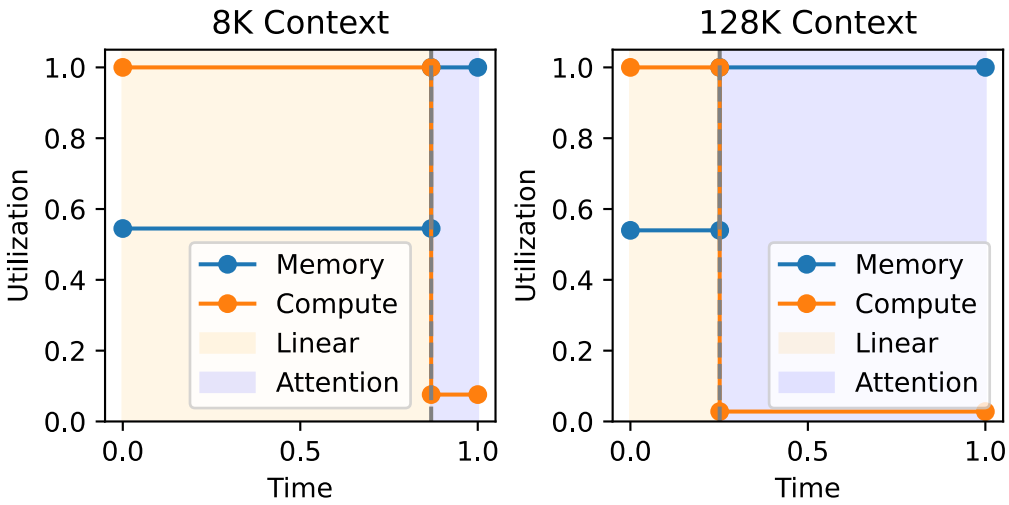



Figure 6.3: In linear operations, LLM inference is often compute-bound and underutilizes memory bandwidth, while in attention operations, it is memory-bound and underutilizes compute cores. As the context length increases, attention operations take longer time. Figure generated based on roofline model with a chunk size of 512.

### *6.2.3 Time-Multiplexed KV-Prefetching*

Figure 6.3 shows a roofline analysis of the memory and compute utilization during linear and attention operations for different context lengths on a H100 GPUs system. We can see quite clearly that during the linear operations, we are fully compute bound with the memory utilization sitting around 57%, while during attention operations we are fully memory bound. In order to leverage both the improved bandwidth that 3D memory systems promise, while overcoming the max capacity limitation, we propose a new technique called ***Time-Multiplexed KV-Prefetching***. Time-Multiplexed KV-Prefetching attempts to increase the memory utilization during linear operations by simultaneously *prefetching the upcoming KV cache for the next attention layer into a faster, though potentially smaller, memory system (e.g., 3D memory).*

Figure 6.4 shows an overview for Time-Multiplexed KV-Prefetching. In LLMs, there are generally 4 linear operations, QKV-Projection, O-Projection, and 2 layers in the Feed-

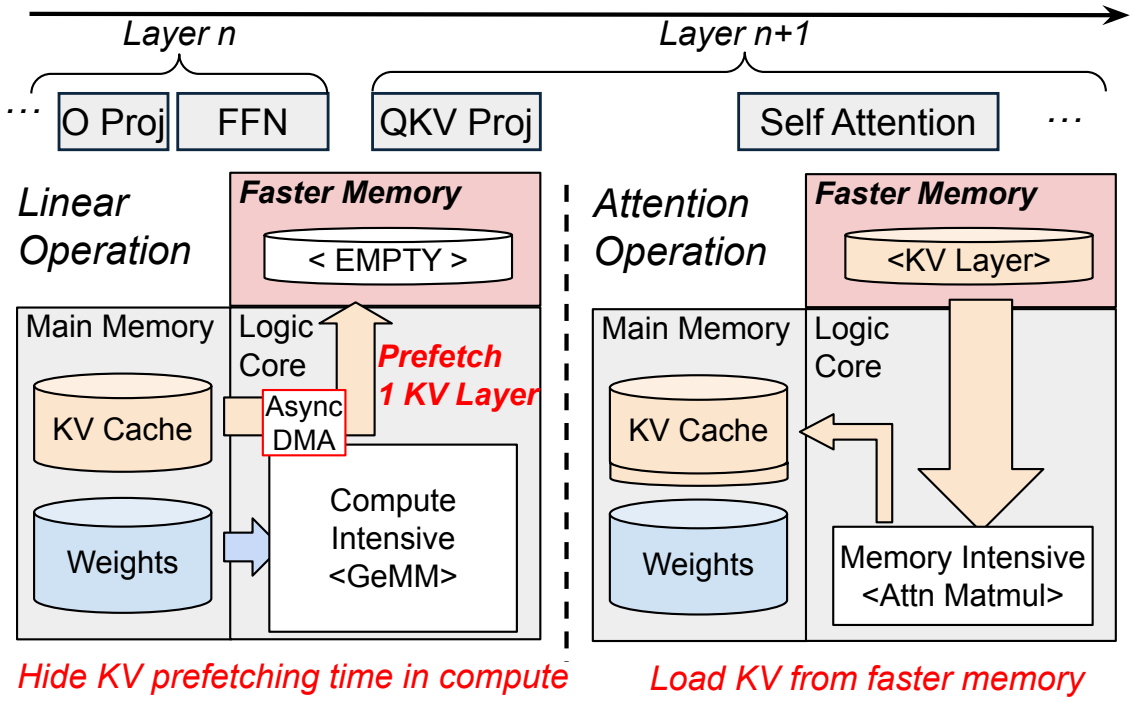



Figure 6.4: Time-Multiplexed KV-Prefetching overview. While performing compute bounded linear operations, we opportunistically move the KV cache of the next attention operation to a faster memory. The attention operation operates entirely out of the faster memory reduce the overall latency with only a small update for new KV cache values back to main memory.

Forward Network (FFN) in each transformer layer. Before starting the O-projection, an asynchronous DMA engine is programmed to start start prefetching KV cache data from the main memory to the 3D memory system. These memory requests are arbitrated at a lower priority with memory requests from the cores. If there is sufficient amount of work to be done during the 4 linear operations, then the data movement from main memory to the 3D memory system can be completely hidden. This technique allows us use the capacity of the main memory system while leveraging the bandwidth of the 3D memory system during the memory bounded attention operations.

## *6.3 Hybrid Bonding 3D DRAM Accelerator*

This section introduces ChronoStack, an LLM inference accelerator integrating 3D hybrid-bonded die-stacked memory. While this technology offers substantial bandwidth and power advantages, its adoption necessitates addressing challenges like heat dissipation, Through-Silicon Via (TSV) overhead, and memory controller integration. We detail the architecture and key design considerations, including area and thermal analyses, that inform our evaluation target.

### *6.3.1 Architectural Overview*

The ChronoStack architecture, depicted in Figure 6.5, modifies a baseline GPU-like structure by incorporating a 3D DRAM memory controller and a DRAM-to-DRAM DMA (D2D-DMA) unit. This D2D-DMA facilitates asynchronous data movement between the existing 2.5D HBM (High Bandwidth Memory) and the new 3D stacked DRAM. The 3D memory operates in a separate address space, with L2 cache slices managing portions of both memory types.

Compute is handled by multiple SIMD cores, each equipped with local shared memory, vector processing units, and tensor processing units. These cores access the L2 cache via a hierarchical crossbar Network-on-Chip (NoC). The 3D memory die itself comprises an array of independent memory banks. Each bank, providing 32MB of capacity, features 1024 data pins operating at 500 MHz, delivering a peak bandwidth of 64 GB/s per bank. These characteristics are based on prior hybrid bonding memory work [217].

Integrating the 3D memory via face-to-face (F2F) hybrid bonding with 3µm pitch Direct Bond Interconnects (DBI) introduces specific overheads. Notably, TSVs with a 25µm pitch are required for power and off-chip signals to bypass the logic die's Front-End-of-Line (FEOL), which now abuts the interposer. While communication between the logic die and the 3D DRAM die uses DBI, area penalties for PHY and ESD protection are considered, though these are less prohibitive than for off-chip interfaces.

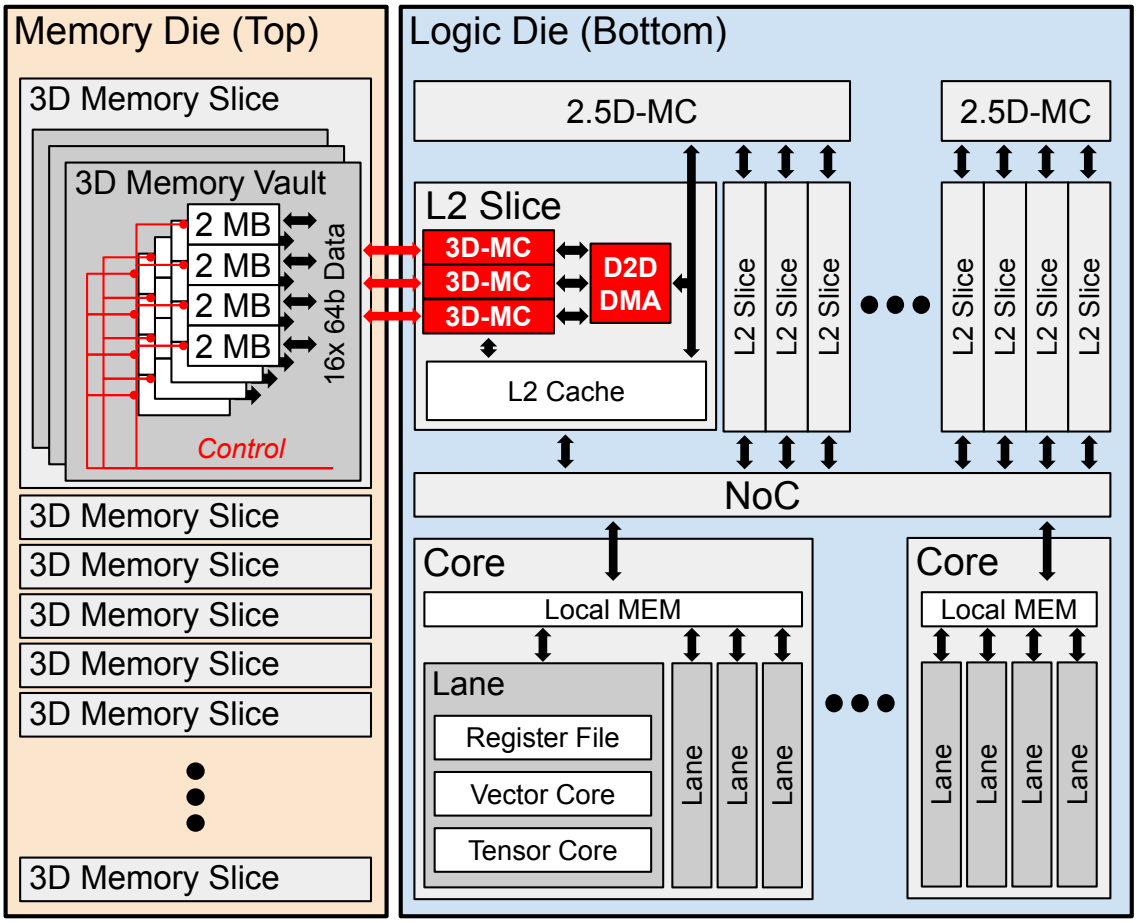



Figure 6.5: Architectural overview of ChronoStack. The logic die sits below the memory die. The logic die is composed of SIMD cores that use both vector and tensor processing units for the computation. These cores talk with the L2 cache over an on-chip network. The L2 cache is organized as slices with each slice containing a portion of the L2 cache, a DRAM-to-DRAM DMA controller, as well as 3D memory controller. The memory die is organized into slices with multiple memory vaults per slice corresponding to the 3D memory controllers per L2 slice.

| **Logic Die** | **Specification** |
|---|---|
| Area | 29mm x 28mm (TSMC 4N) |
| Core | 95 Active Cores (110 Cores @ 87% yield)<br>4 Lanes / Core<br>64x FP16 Vector Core / Lane<br>512x FP16 PE Tensor Core / Lane |
| L2 Cache | 48 MB<br>96x 0.5MB Slices |
| Other | 6 HBM3 Dies<br>18 Off-chip Communication Channels |
| **Memory Die** | **Specification** |
| Area | 29mm x 28mm (17nm DRAM Technology) |
| Capacity | 9 GB, 288 Channels, 32 MB/ch |
| Bandwidth | 18 TB/sec, 64 GB/sec/vault |
| Freq. | 500 MHz |

Table 6.1: Modified H100 [133] design specification with 3D hybrid bonded memory for ChronoStack.

### *6.3.2 Design Specification*

Table 6.1 summarizes the ChronoStack design specification, which is benchmarked against an NVIDIA H100 GPU [133] due to its state-of-the-art status for LLM inference.

To derive these specifications, an analysis based on NVIDIA A100 die photos was used to estimate silicon area allocation for Streaming Multiprocessors (SMs) and L2 cache, which then informed H100 SM area estimates. The H100 L2 cache was assumed to use 0.5MB slices similar to the A100. To accommodate the overheads of the 3D die-stacked memory (memory controllers and D2D-DMA engines), the number of SMs was reduced from the H100's physical count of 144 to 110 (resulting in 95 active SMs at an 87% yield, compared to H100's 132 active out of 144). Similarly, the L2 cache was reduced from 100 slices (50MB)

to 96 slices (48MB). Each L2 slice interfaces with three 3D DRAM banks, modeled after the memory in [217]. The area for the additional 3D memory controllers and D2D-DMA engines was estimated by scaling 12nm process implementations (using Synopsys tools for synthesis and layout of the DMA) to a target 4N/5nm-class process node, using the ratio of Minimum Metal Pitch (MMP) times Contacted Polysilicon Pitch (CPP). These additional components account for approximately 7.1% of the final logic die area.

## *6.4 Evaluation*

Our evaluation is conducted on two systems. The baseline system is an H100 GPU-based configuration, employing the state-of-the-art chunked mixed continuous batching (C-MCB) mechanism [74, 4]. The second system, ChronoStack, is the proposed 3D hybrid-bonded memory enhanced architecture, which leverages Time-Multiplexed KV-Prefetching to accelerate attention operations. Both systems share identical system-level configurations, such as the number of nodes and node interconnects (4th generation NVLink and NVLink Switch [135]). The number of nodes utilized varies depending on the model and maximum context length, as detailed in Table 6.3. Both systems adopt NVIDIA's TensorRT-LLM MultiShot [103] for inter-node communication, ensuring identical communication overheads. It is important to note that the chosen node count does not represent the minimum requirement for running these models. Instead, increasing the number of nodes enhances SLO performance, as mixed continuous batching benefits from scaling. This is a widely adopted practice in real-world LLM inference studies [143]. The latest NVLink Switch [135] supports up to 576 GPUs in direct connection, while TPU-based clusters [38] scale up to 4,096 nodes, making configurations in the range of 64 to 96 nodes well within practical deployment limits. This simulator employs chunked mixed continuous batching as the baseline batching methodology, allowing us to evaluate the impact of different chunk sizes on SLOs.

### *6.4.1 Main Insights*

We first evaluate the two systems on different models, tasks, and maximum context lengths. The 8K code-based and conversation-based traces are generated directly from real product traces [143, 126]. We linearly scale both the input and output to generate the 32K and 128K

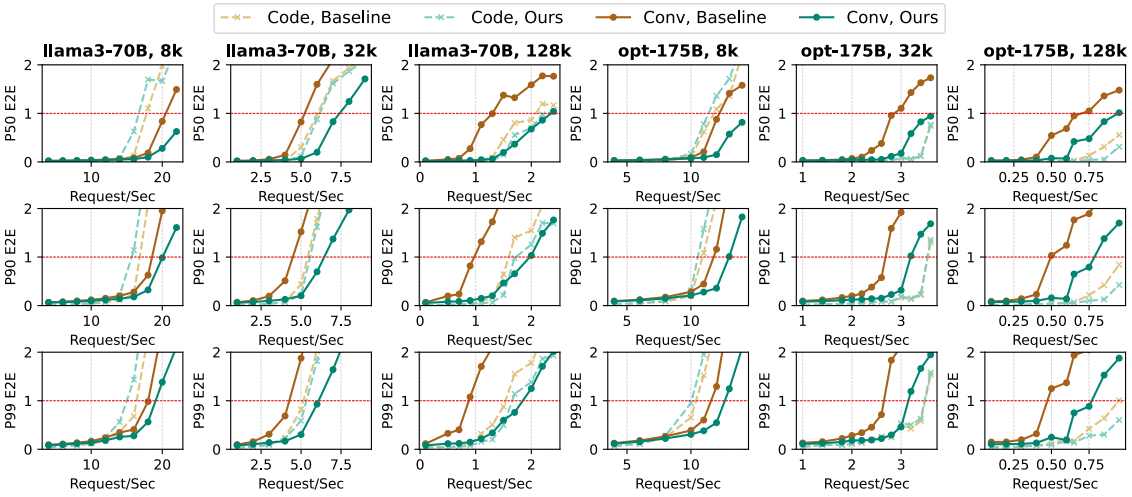



Figure 6.6: E2E comparison of proposed and baseline system across input request rates. Different columns of the graph represent different models and maximum context lengths. Dashed red lines indicate E2E SLO.

traces, preserving the input-output ratio. To stress-test the systems, Figure 6.6 presents the P50, P90, and P99 end-to-end (E2E) latencies at varying input request rates. These latencies are normalized to the SLOs set in Table 6.2.

Following the trace generation methodology in Splitwise [143], we use exponential distributions to model the arrival times of requests based on expected request rates. Each column in Figure 6.6 represents a combination of model and maximum context length, with the input load (requests per second) increasing from left to right. Upon analyzing the figure, we observed the following insights:

**In most cases, ChronoStack effectively reduces the E2E latency.** In the majority of figures, the proposed system (ChronoStack, blue lines) consistently shows similar or significantly lower end-to-end (E2E) latency compared to the baseline system (brown lines) at any given input load. For instance, in the conversation task with Llama3-70B at an 8K context, when the input load is 20 requests per second, our system reduces the P50, P90, and P99 E2E latencies by 66.8%, 49.4%, and 44.8%, respectively. This also means that

| Context | TTFT | TBT | E2E |
|---|---|---|---|
| **8K** | 400 ms | 100 ms | 25.4 s (250 tokens generated) |
| **16K** | 800 ms | 100 ms | 50.8 s (500 tokens generate) |
| **32K** | 1600 ms | 100 ms | 101.6 s (1000 tokens generate) |
| **64K** | 3200 ms | 100 ms | 203.2 s (2000 tokens generate) |
| **128K** | 6400 ms | 100 ms | 406.4 s (4000 tokens generate) |

Table 6.2: SLOs for evaluation. E2E is set to be TTFT plus the time to generate a number of tokens while meeting TBT.

| | 8K | 16K-32K | 64K-128K |
|---|---|---|---|
| **Llama3-70B** | 16 | 32 | 64 |
| **OPT-175B** | 24 | 48 | 96 |

Table 6.3: Number of nodes used in different models and maximum context lengths.

our system can handle higher input loads while still meeting SLOs. Figure 6.7 illustrates the throughput improvement over the baseline, defined as the maximum input load that the system can handle while still meeting the E2E SLO. Our system achieves up to a 2× improvement in throughput. Additionally, both models exhibit performance gains, highlighting the versatility of the proposed system for different LLMs, regardless of their scale or attention mechanisms.

**ChronoStack is more advantageous in conversation-based tasks than in code-based tasks.** The latency reduction in code-based tasks (dashed lines) is less pronounced compared to conversation tasks (solid lines). In some instances, such as Llama3-70B with an 8K context (first column), the proposed system exhibits higher latency than the baseline. This is due to the fact that code tasks typically have a much larger input-output ratio, as shown in Figure 4.5. Consequently, code tasks involve more prefill tokens and fewer decode tasks, making the system compute-bound for a longer duration. Since ChronoStack

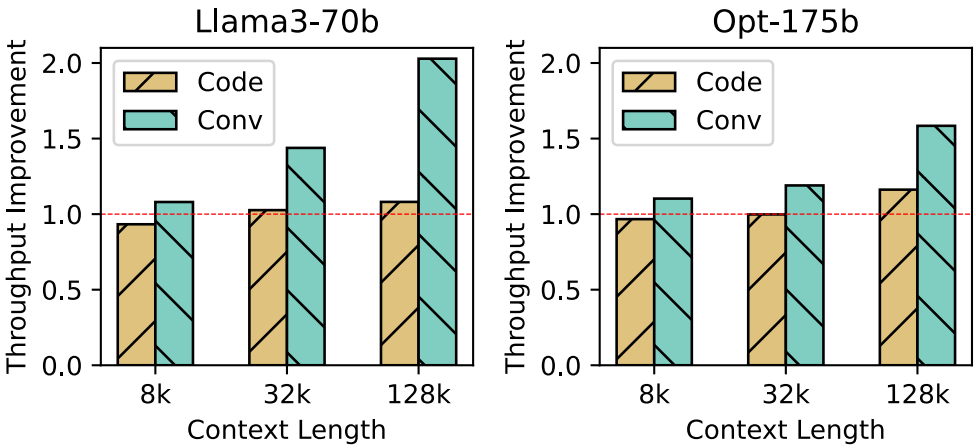



Figure 6.7: End-to-end throughput improvement. Throughput is measured as the maximum input request rate while the system still meets the E2E SLO.

experiences thermal throttling, it has lower peak performance, which results in the baseline system outperforming ChronoStack for these compute-intensive workloads. However, Figure 6.7 demonstrates that our system still improves throughput for code-based tasks by up to 1.2×, though this improvement is smaller than that observed for conversation tasks.

**ChronoStack becomes more beneficial as the context length increases.** As the context length increases (from columns 1 to 3 or 4 to 6 in Figure 6.6), the proposed system reduces latency more significantly across a range of input loads. This trend is also reflected in the throughput comparison in Figure 6.7. For Llama3-70B conversation tasks, throughput improves by 1.1×, 1.5×, and 2.0× for 8K, 32K, and 128K contexts, respectively. A more detailed analysis of the relationship between context length and input-output ratio is provided in the next section.

### *6.4.2 Impact of context length and ratio*

Both context length and input-output ratio are important characteristics of a workload. To study how these factors impact performance, we generate synthetic traces with maximum

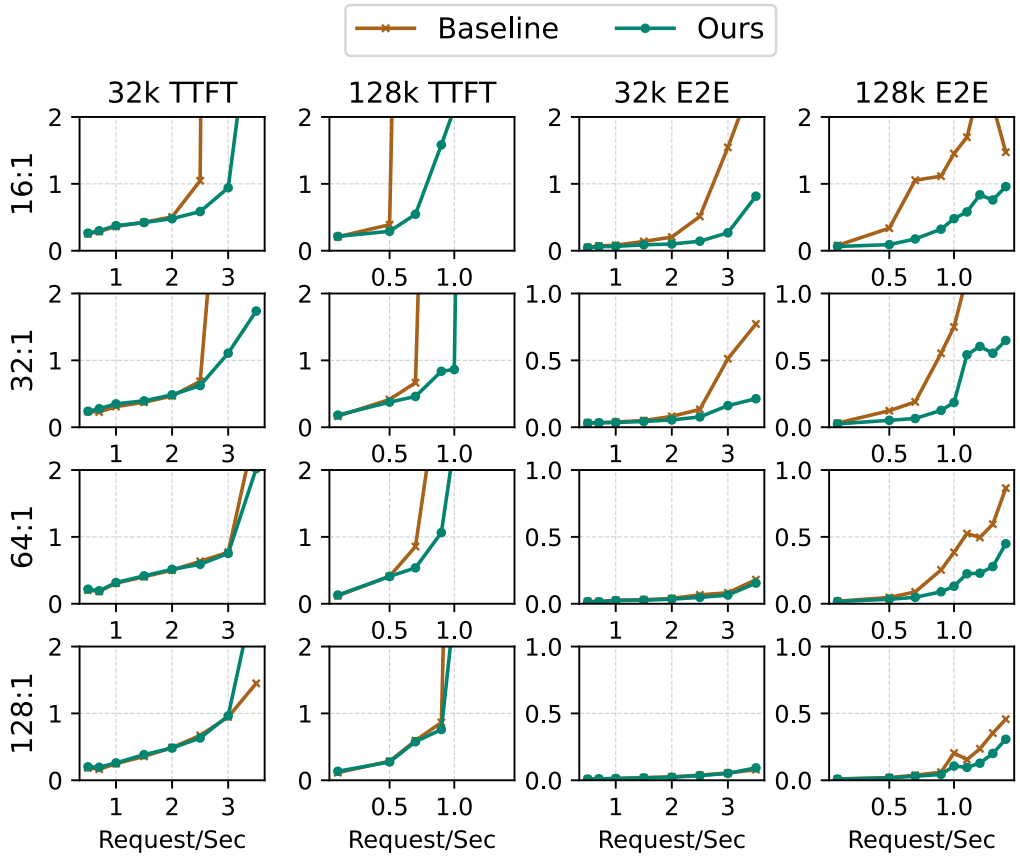



Figure 6.8: Llama3-70B P90 TTFT and E2E latency for 32K and 128K context lengths and different input-output ratios.

context lengths ranging from 16K to 128K and input-output ratios from 16:1 to 256:1.

Figure 6.8 plots the P90 TTFT and E2E latency for different synthetic traces. In each column, the input-output ratio decreases from bottom to top, with performance improvements in TTFT and E2E becoming more significant (evidenced by larger gaps between the lines). For each row, our system shows more substantial E2E latency reduction for the 128K context compared to the 32K context.

Figure 6.9 shows the E2E throughput improvement of ChronoStack at various context lengths and input-output ratios. The value X indicates that the system can sustain an input request rate X times higher than the baseline while still meeting the E2E SLO. A smaller

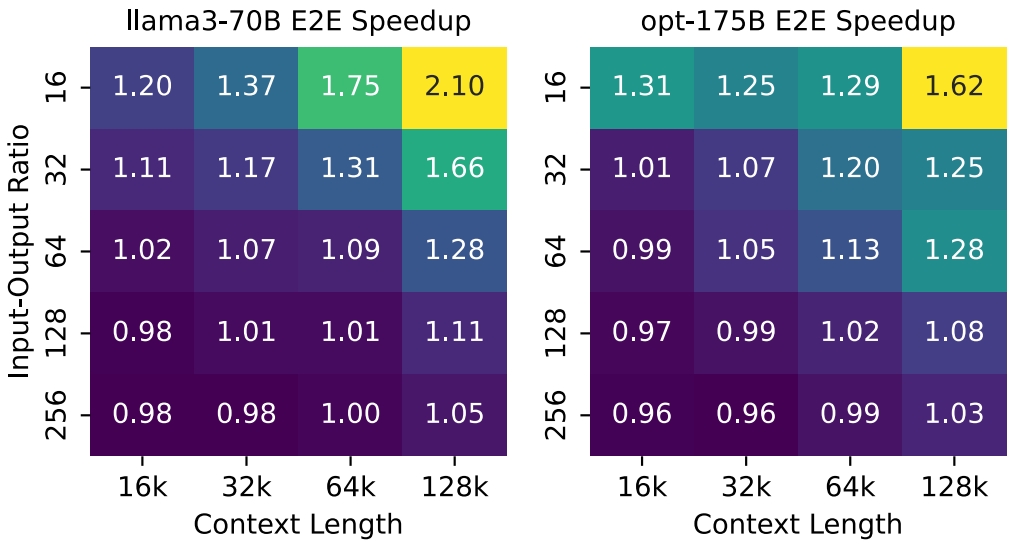



Figure 6.9: Throughput improvement of the proposed system over the baseline given different context lengths and input-output ratios.

input-output ratio results in more decode tasks, increasing the memory-bound attention operations. Similarly, longer context lengths require reading more KV cache during the attention operation, making the entire workload more memory-intensive. Therefore, the greatest improvements are observed at longer context lengths and smaller input-output ratios. For example, with 128K contexts and an input-output ratio of 16, ChronoStack improves the E2E throughput by 2.10× and 1.62× for Llama3-70B and OPT-175B, respectively.

### *6.4.3 Evaluation on Models with Optimized KV-Cache*

Recently, models like the DeepSeek family [46, 47, 45] have adopted Multi-head Latent Attention (MLA) to reduce KV-cache size and optimize memory bandwidth usage. While MLA helps lower the overall memory footprint, memory bandwidth can still be a bottleneck in certain scenarios.

To evaluate whether ChronoStack continues to provide benefits for models with optimized KV-cache, we compare DeepSeek V3 [47] inference latency across two systems, both using

mixed continuous batching with a prefill block size of 2048. Each system consists of 64 nodes and employs 64-way expert parallelism, following the original DeepSeek setup. Figure 6.10 presents latency comparisons across two dimensions. The left plot shows latency vs. average context length per decode task where ChronoStack consistently outperforms GPU, though the gap remains nearly constant across different context lengths. This suggests that while MLA reduces KV-cache pressure, Time-Multiplexed KV-Prefetching still improves overall efficiency. The right plot illustrates latency vs. the number of concurrent decoding tasks in mixed continuous batching, showing that ChronoStack becomes increasingly beneficial as concurrency grows, leading to larger latency reductions (17%) compared to GPUs. This is mainly because there are some Matmul kernels with low computational intensity in MLA decode tasks, which indicates that it still requires high memory bandwidth for efficient large-scale serving. Overall, while MLA reduces KV-cache overhead, ChronoStack ensures efficient data movement for memory-bound kernels, making it particularly valuable in large-scale nference scenarios.

Other techniques, such as sliding window attention [16], trade accuracy for efficiency by discarding earlier tokens, limiting their adoption in frontier models like DeepSeek R1 [45], GPT-4o [137], and Gemini 2.0 [44]. ChronoStack provides a more scalable KV-cache management approach, maintaining both efficiency and accuracy for long-context workloads.

### *6.4.4 Chunk Size and 3D Memory Usage*

One important parameter to set in both the baseline and our systems is the prefill chunk size. Figure 6.11 demonstrates how the chunk size impacts all latency metrics for Llama3-70B at an 8K context. Smaller prefill chunk sizes reduce TTFT but increase TBT, affecting the overall E2E latency. Selecting the optimal prefill chunk size is critical for meeting SLOs. In the case shown, we choose a chunk size of 512 to minimize ETE latency. It's also important to note that smaller chunk sizes reduce the time required to prefetch data from HBM to 3D. Typically, larger chunk sizes are preferred for longer context lengths since more KV values need to be prefetched. For workloads with context lengths greater than 16K, we set the chunk size to 2048.

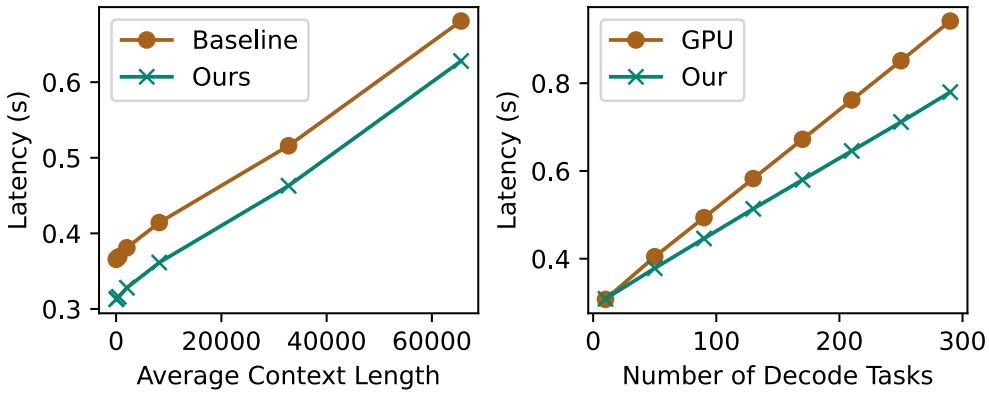



Figure 6.10: Latency comparison of DeepSeek V3. Left: Latency vs. average context length per decode task. Right: Latency vs. number of concurrent decoding tasks. ChronoStack consistently outperforms GPU, with increasing benefits as number of decode task grows.

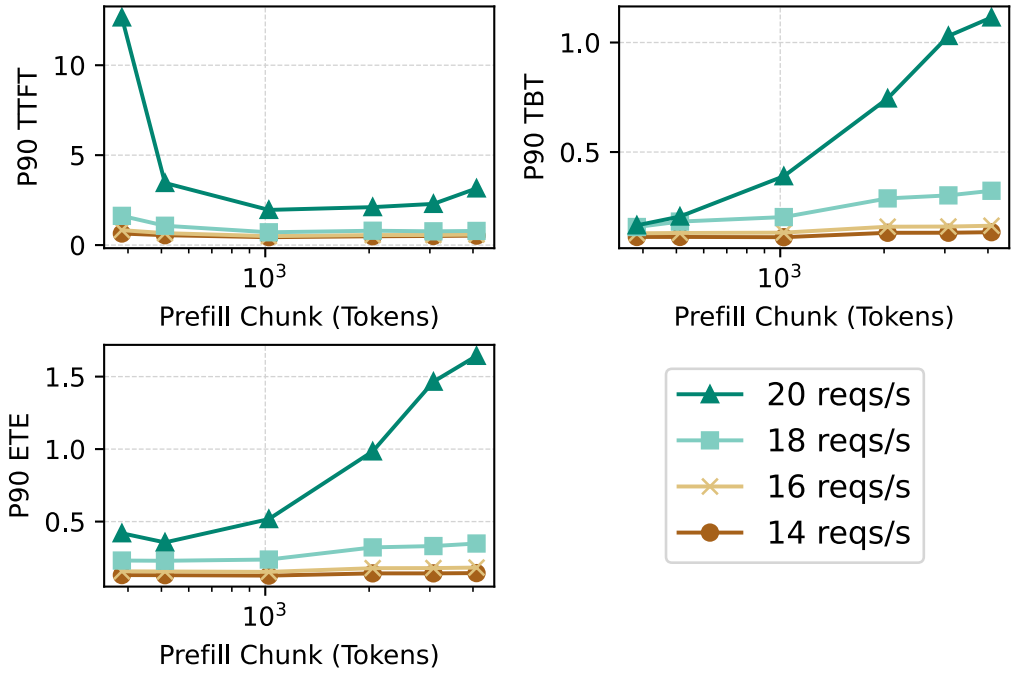



Figure 6.11: Latencies across different chunk sizes for Llama3-70B at 8K context. Small prefill chunk sizes reduce TTFT and increase TBT, while affecting E2E.

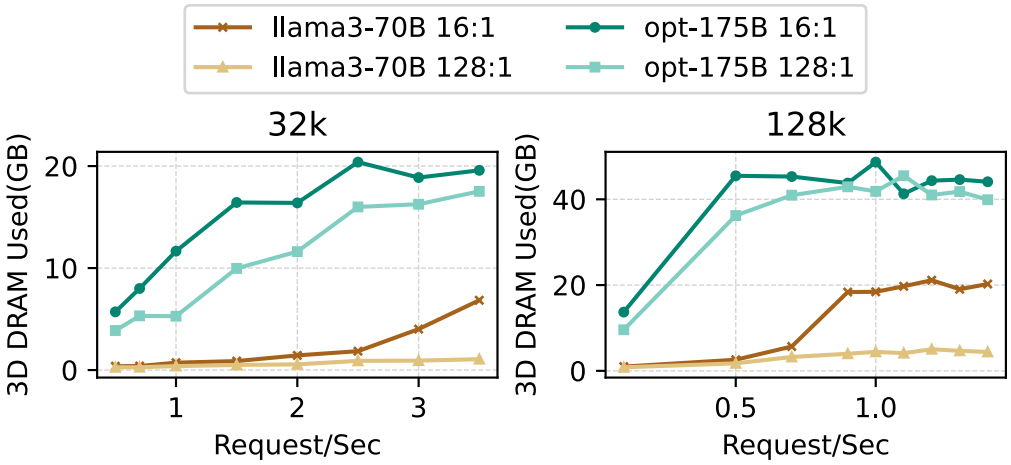



Figure 6.12: 3D memory usage in the system. 3D memory usage in the system. Higher input loads use more 3D memory due to more concurrent tasks. Multi-head models require more memory than grouped-query model.

Figure 6.12 illustrates the total 3D DRAM used by the system, representing the KV cache data prefetched from HBM. Larger input workloads involve more concurrent tasks and, consequently, require more 3D memory. The multi-head model, OPT-175B, demands more memory than the grouped-query model, Llama3-70B, as it provides a distinct KV cache for each attention head. Regardless of the model, the memory usage remains well below the total 3D memory capacity in the system.

It's also important to note that the Time-Multiplexed KV-Prefetching mechanism is not limited to hybrid bonding 3D memory. The concept can be applied to any memory architecture that is faster than the main memory and has sufficient capacity to accommodate one layer of KV.

## *6.5 Conclusion*

In this chapter, we introduce ChronoStack, a hybrid bonded 3D-DRAM integrated architecture for LLM inference optimized for improved context length scalability in low-ratio workloads and end-to-end latency service level objects. To overcome limitations in the memory

capacity of the 3D-DRAM, we propose Time-Multiplexed KV-Prefetching as a technique to utilize the improved 3D-DRAM bandwidth despite capacity limitations when scaling up the context length by time-multiplexing the 3D-DRAM through prefetching. We show that ChronoStack with Time-Multiplexed KV-Prefetching is able to improve the end-to-end latency of Llama3-70B by 2.1× and OPT-175B by 1.62× in low-ratio, large context length workloads with minimal degradation to high-ratio, small context length workloads.

# Chapter 7

# CONCLUSION

The continued advancement of Artificial Intelligence, particularly the development of Large Language Models, has led to a significant demand for specialized and efficient hardware. General-purpose architectures, while foundational, face challenges in meeting the performance, efficiency, and scalability needs of these complex models. This dissertation has addressed these challenges by investigating, designing, and evaluating ASIC accelerator architectures and supporting methodologies, with a primary focus on AI inference, especially for LLMs. This chapter summarizes the main contributions of this research and explores potential avenues for future work.

## *7.1 Summary of Research and Contributions*

This doctoral research aimed *to design and evaluate high-performance and efficient ASIC accelerator architectures and supporting methodologies for diverse and evolving AI applications, with a primary focus on Large Language Models.* The work presented has made several contributions towards this objective:

The research began with foundational explorations into accelerating earlier AI paradigms (Chapter 3). Contributions in this area included involvement in the development of **iFPNA**, a flexible processor architecture adaptable to various neural network models, and the design of **DRLP** with the **F|B|C dataflow** for Deep Q-Learning. These studies provided insights into architectural adaptability, dataflow optimization, and hardware-software co-design.

Recognizing the complexities of contemporary LLM inference, **ReaLLM** was developed as a holistic, multi-level hardware system simulation framework (Chapter 4) [146]. ReaLLM facilitates the evaluation of LLM inference systems by bridging detailed accelerator-level analysis with system-wide performance considerations, utilizing a precomputed kernel library and trace-driven simulation.

To address the economic and scalability aspects of large-scale LLM deployment, **Chiplet Cloud** was proposed as a TCO-optimized, chiplet-based ASIC architecture (Chapter 5) [145]. This architecture features an on-chip memory system (CC-MEM) for model parameters, a methodology for co-optimizing chiplet die size with software mapping, and support for model sparsity to enhance TCO.

Finally, to manage the memory bandwidth demands of long-context LLMs, **ChronoStack**, a 3D-memory integrated architecture, was investigated as part of a collaborative effort (Chapter 6). This work introduced the **Time-Multiplexed KV-Prefetching** technique, aiming to leverage the high bandwidth of hybrid-bonded 3D DRAM to accelerate attention mechanisms.

These contributions, taken together, address the research objective by providing both specific architectural solutions and supporting methodologies for AI inference. The focus on LLMs in the latter contributions directly targets the challenges posed by these large-scale models. The work aims to contribute to the development of more capable and efficient AI systems, and the methodologies developed offer tools for researchers and practitioners in AI hardware.

### *7.2 Future Work*

While this dissertation presents several advancements, the field of AI hardware continues to evolve, suggesting numerous avenues for future investigation:

- **Advancing Simulation and Modeling Capabilities:** Future work on simulation tools like ReaLLM could involve integrating automated Design Space Exploration (DSE) algorithms, incorporating more detailed power and thermal modeling, and extending support for emerging LLM architectures and novel hardware technologies. Specifically, for architectures like Mixture-of-Experts (MoE) models, ReaLLM can be enhanced to model the dynamic activation of experts, simulate complex communication patterns arising from expert parallelism, evaluate various load balancing strategies among experts, and analyze the performance implications of different token routing mechanisms. Expanding simulation to cover distributed LLM training and

other emerging model types like multimodal models would also be beneficial.

- **Evolving Disaggregated and Cost-Optimized Architectures:** For architectures such as Chiplet Cloud, further research could focus on heterogeneous chiplet integration, advanced dynamic resource management for large-scale disaggregated systems, and continued co-optimization of interconnects.

- **Innovations in Advanced Memory Systems for AI:** Regarding 3D-integrated memory systems like ChronoStack, future investigations could explore multi-layered 3D stacking of DRAM or heterogeneous memory types. Developing more intelligent prefetching and caching algorithms for techniques, and exploring near-memory processing within 3D stacks, are also promising directions.

- **Addressing Overarching Themes in AI Hardware:** Broader research areas include ongoing algorithm-hardware co-design for model compression techniques like sparsity and quantization. A continued focus on sustainable AI hardware, addressing energy efficiency across the stack, is important. Furthermore, contributions to standardization and open ecosystems for AI hardware can foster wider collaboration and innovation.

The research presented in this dissertation aims to provide a foundation for these and other future explorations in the dynamic field of AI inference hardware.

# BIBLIOGRAPHY

[1] Dennis Abts, Garrin Kimmell, Andrew Ling, John Kim, Matt Boyd, Andrew Bitar, Sahil Parmar, Ibrahim Ahmed, Roberto DiCecco, David Han, John Thompson, Michael Bye, Jennifer Hwang, Jeremy Fowers, Peter Lillian, Ashwin Murthy, Elyas Mehtabuddin, Chetan Tekur, Thomas Sohmers, Kris Kang, Stephen Maresh, and Jonathan Ross. A Software-defined Tensor Streaming Multiprocessor for Large-scale Machine Learning. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA)*, 2022.

[2] A. Agarwal, S. Amarasinghe, R. Barua, M. Frank, W. Lee, V. Sarkar, D. Srikrishna, and M. Taylor. The RAW compiler project. In *Proceedings of the Second SUIF Compiler Workshop*, pages 21–23, 1997.

[3] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in llm inference with sarathi-serve. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, 2024.

[4] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, and Ramachandran Ramjee. SARATHI: Efficient LLM Inference by Piggybacking Decodes with Chunked Prefills, 2023.

[5] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 105–117, 2015.

[6] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. *arXiv:2305.13245 [cs]*, 2023.

[7] Tutu Ajayi, Khalid Al-Hawaj, Aporva Amarnath, Steve Dai, Scott Davidson, Paul Gao, Gai Liu, Atieh Lotfi, Julian Puscar, Anuj Rao, Austin Rovinski, Loai Salem, Ningxiao Sun, Christopher Torng, Luis Vega, Bandhav Veluri, Xiaoyang Wang, Shaolin Xie, Chun Zhao, Ritchie Zhao, Christopher Batten, Ronald G. Dreslinski, Ian Galton, Rajesh K. Gupta, Patrick P. Mercier, Mani Srivastava, Michael Bedford Taylor, and Zhiru Zhang. Celerity: An Open Source RISC-V Tiered Accelerator Fabric. In *HOTCHIPS*, Aug 2017.

[8] Alric Althoff, Joseph McMahan, Luis Vega, Scott Davidson, Timothy Sherwood, Michael Taylor, and Ryan Kastner. Hiding Intermittant Information Leakage with Architectural Support for Blinking. In *International Symposium on Computer Architecture (ISCA)*, 2018.

[9] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, and Yuxiong He. DeepSpeed-Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2022.

[10] Manish Arora, Jack Sampson, Nathan Goulding-Hotta, Jonathan Babb, Ganesh Venkatesh, Michael Bedford Taylor, and Steven Swanson. Reducing the Energy Cost of Irregular Code Bases in Soft Processor Systems. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2011.

[11] Saahil Athrij. Vectorizing the Hamerblade Compiler. Master's thesis, University of Washington, 2024.

[12] Z. Azad, G. Yang, R. Agrawal, D. Petrisko, M. Taylor, and A. Joshi. Race: Risc-v soc for en/decryption acceleration on the edge for homomorphic computation. In *ISLPED*, 2022.

[13] Zahra Azad, Guowei Yang, Rashmi Agrawal, Daniel Petrisko, Michael Taylor, and Ajay Joshi. RISE: RISC-V SoC for En/Decryption Acceleration on the Edge for Homomorphic Encryption. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2023.

[14] Jonathan Babb, Matthew Frank, Victor Lee, Elliot Waingold, Rajeev Barua, Michael Taylor, Jang Kim, Srikrishna Devabhaktuni, and Anant Agarwal. The Raw Benchmark Suite: Computation Structures for General Purpose Computing. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 1997.

[15] Abhimanyu Bambhaniya, Ritik Raj, Geonhwa Jeong, Souvik Kundu, Sudarshan Srinivasan, Midhilesh Elavazhagan, Madhu Kumar, and Tushar Krishna. Demystifying platform requirements for diverse llm inference use cases. 2024.

[16] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer, 2020.

[17] B. Beresini, S. Ricketts, and M.B. Taylor. Unifying manycore and fpga processing with the RUSH architecture. In *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on*, pages 22 –28, 2011.

[18] Vikram Bhatt, Nathan Goulding-Hotta, Qiaoshi Zheng, Jack Sampson, Steve Swanson, and Michael B. Taylor. Sichrome: Mobile web browsing in Hardware to save Energy . In *Dark Silicon Workshop, ISCA*, 2012.

[19] BigScience. BLOOM: A 176B-Parameter Open-Access Multilingual Language Model. *arXiv:2211.05100 [cs]*, 2023.

[20] Ajay Brahmakshatriya, Emily Furst, Victor Ying, Claire Hsu, Max Ruttenberg, Yunming Zhang, Tommy Jung, Dustin Richmond, Michael Taylor, Julian Shun, Mark Oskin, Daniel Sanchez, and Saman Amarasinghe. Taming the zoo: A unified graph compiler framework for novel architectures. In *ISCA*, 2021.

[21] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. *arXiv:2005.14165 [cs]*, 2020.

[22] Sadullah Canakci, Leila Delshadtehrani, Furkan Eris, Michael Bedford Taylor, Manuel Egele, and Ajay Joshi. Directfuzz: Automated test generation for rtl designs using directed graybox fuzzing. In *DAC*, 2021.

[23] Sadullah Canakci, Chathura Rajapaksha, Leila Delshadtehrani, Anoop Nataraja, Michael Bedford Taylor, Manuel Egele, and Ajay Joshi. Processorfuzz: Processor fuzzing with control and status registers guidance. In *HOST*, 2023.

[24] Chixiao Chen, Hongwei Ding, Huwan Peng, Haozhe Zhu, Rui Ma, Peiyong Zhang, Xiaolang Yan, Yu Wang, Mingyu Wang, Hao Min, and Richard C.-J Shi. OCEAN: An on-Chip Incremental-Learning Enhanced Processor with Gated Recurrent Neural Network Accelerators. In *43rd IEEE European Solid State Circuits Conference (ESSCIRC)*, pages 259–262, September 2017.

[25] Chixiao Chen, Xindi Liu, Huwan Peng, Hongwei Ding, and C.-J. Richard Shi. iFPNA: A Flexible and Efficient Deep Learning Processor in 28-nm CMOS Using a Domain-Specific Instruction Set and Reconfigurable Fabric. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):346–357, June 2019.

[26] Chixiao Chen, Huwan Peng, Xindi Liu, Hongwei Ding, and C.-J. Richard Shi. Exploring the Programmability for Deep Learning Processors: From Architecture to Tensorization. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, San Francisco, CA, June 2018.

[27] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, page 269–284, New York, NY, USA, 2014. Association for Computing Machinery.

[28] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, January 2017.

[29] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. *arXiv preprint arXiv:1807.07928*, May 2019.

[30] Lin Cheng, Peitian Pan, Zhongyuan Zhao, Krithik Ranjan, Jack Weber, Bandhav Veluri, Seyed Borna Ehsani, Max Ruttenberg, Dai Cheol Jung, Preslav Ivanov, Dustin Richmond, Michael B. Taylor, Zhiru Zhang, and Christopher Batten. A tensor processing framework for cpu-manycore heterogeneous systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1620–1635, 2022.

[31] Lin Cheng, Max Ruttenberg, Dai Cheol Jung, Dustin Richmond, Michael Taylor, Mark Oskin, and Christopher Batten. Beyond Static Parallel Loops: Supporting Dynamic Task Parallelism on Manycore Architectures with Software-Managed Scratchpad Memories. In *ASPLOS*, 2023.

[32] Hyungmin Cho, Pyeongseok Oh, Jiyoung Park, Wookeun Jung, and Jaejin Lee. FA3C: FPGA-Accelerated Deep Reinforcement Learning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 499–513, Providence, RI, April 2019.

[33] Seungkyu Choi, Jaehyeong Sim, Myeonggu Kang, Yeongjae Choi, Hyeonuk Kim, and Lee-Sup Kim. An Energy-Efficient Deep Convolutional Neural Network Training Accelerator for in Situ Personalization on Smart Devices. *IEEE Journal of Solid-State Circuits*, 55(10):2691–2702, 2020.

[34] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal,

Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. PaLM: Scaling Language Modeling with Pathways. *arXiv:2204.02311 [cs]*, 2022.

[35] Yuan-Mao Chueh. A Complete Open Source Network Stack For BlackParrot. Master's thesis, University of Washington, 2022.

[36] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling, 2014.

[37] Google Cloud. Cloud TPU Pricing. `https://cloud.google.com/tpu/pricing#v4-pricing`, 2023.

[38] Google Cloud. Tensor Processing Units (TPUs). `https://cloud.google.com/tpu?hl=en`, 2024.

[39] CNET. Gigawatt: The solar energy term you should know about. `https://www.cnet.com/home/energy-and-utilities/gigawatt-the-solar-energy-term-you-should-know-about/`, 2021.

[40] Jason Cong and Bingjun Xiao. Minimizing computation in convolutional neural networks. In *International conference on artificial neural networks*, pages 281–290. Springer, 2014.

[41] William J. Dally, Yatish Turakhia, and Song Han. Domain-specific hardware accelerators. *Commun. ACM*, 63(7):48–57, June 2020.

[42] Scott Davidson, Shaolin Xie, Chris Torng, Khalid Al-Hawaj, Austin Rovinski, Tutu Ajayi, Luis Vega, Chun Zhao, Ritchie Zhao, Steve Dai, Aporva Amarnath, Bandhav Veluri, Paul Gao, Anuj Rao, Gai Liu, Rajesh K. Gupta, Zhiru Zhang, Ronald Dreslinski, Christopher Batten, and Michael Bedford Taylor. The Celerity Open-Source 511-core RISC-V Tiered Accelerator Fabric. *Micro, IEEE*, Mar/Apr. 2018.

[43] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. Large Scale Distributed Deep Networks. In *NeurIPS*, volume 25. Curran Associates, Inc., 2012.

[44] Google DeepMind. Gemini 2.0. `https://deepmind.google/technologies/gemini/`, 2025.

[45] DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025.

[46] DeepSeek-AI, Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Hao Yang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jin Chen, Jingyang Yuan, Junjie Qiu, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruizhe Pan, Runxin Xu, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shirong

Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Size Zheng, T. Wang, Tian Pei, Tian Yuan, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Liu, Xin Xie, Xingkai Yu, Xinnan Song, Xinyi Zhou, Xinyu Yang, Xuan Lu, Xuecheng Su, Y. Wu, Y. K. Li, Y. X. Wei, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Zheng, Yichao Zhang, Yiliang Xiong, Yilong Zhao, Ying He, Ying Tang, Yishi Piao, Yixin Dong, Yixuan Tan, Yiyuan Liu, Yongji Wang, Yongqiang Guo, Yuchen Zhu, Yuduan Wang, Yuheng Zou, Yukun Zha, Yunxian Ma, Yuting Yan, Yuxiang You, Yuxuan Liu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhewen Hao, Zhihong Shao, Zhiniu Wen, Zhipeng Xu, Zhongyu Zhang, Zhuoshu Li, Zihan Wang, Zihui Gu, Zilin Li, and Ziwei Xie. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model, 2024.

[47] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda

Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. Deepseek-v3 technical report, 2024.

[48] Maria Deutscher. Taalas raises $50M to develop chips optimized for specific AI models. `https://siliconangle.com/2024/03/06/taalas-raises-50m-develop-chips-optimized-specific-ai-models/`, 2024.

[49] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019.

[50] Hadi Esmaeilzadeh and Michael Bedford Taylor. Open Source Hardware: Stone Soups and Not Stone Satues, Please. In *SIGARCH Computer Architecture Today*, Dec 2017.

[51] William Fedus, Barret Zoph, and Noam Shazeer. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *arXiv:2101.03961 [cs]*, January 2021. arXiv: 2101.03961.

[52] Elias Frantar and Dan Alistarh. SparseGPT: Massive Language Models Can Be Accurately Pruned in One-Shot. *arXiv:2301.00774 [cs]*, 2023.

[53] Emily Furst. *Code Generation and Optimization of Graph Programs on a Manycore Architecture*. PhD thesis, University of Washington, 2021.

[54] Guilian Gao, Thomas Workman, Cyprian Uzoh, K. M. Bang, Laura Mirkarimi, Jeremy Theil, Dominik Suwito, Rajesh Katkar, Gill Fountain, Gabe Guevara, and Bongsub Lee. Die to wafer stacking with low temperature hybrid bonding. In *2020 IEEE 70th Electronic Components and Technology Conference (ECTC)*, pages 589–594, 2020.

[55] Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis. TANGRAM: Optimized Coarse-Grained Dataflow for Scalable NN Accelerators. In *ASPLOS*, pages 807–820, Providence, RI, April 2019. ACM.

[56] S. Garcia, Donghwan Jeon, C. Louie, and M.B. Taylor. The Kremlin Oracle for Sequential Code Parallelization. *Micro, IEEE*, 32(4):42–53, July-Aug. 2012.

[57] Saturnino Garcia, Donghwan Jeon, Chris Louie, Sravanthi Kota Venkata, and Michael Bedford Taylor. Bridging the Parallelization Gap: Automating Parallelism Discovery and Planning. In *USENIX Workshop on Hot Topics in Parallelism (HOTPAR)*, 2010.

[58] Saturnino Garcia, Donghwan Jeon, Chris Louie, and Michael Bedford Taylor. Kremlin: Rethinking and Rebooting gprof for the Multicore Age. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[59] GitHub. GitHub Copilot Your AI Pair Programmer. `https://github.com/features/copilot`, 2023.

[60] Nathan Goulding, Jack Sampson, Ganesh Venkatesh, Saturnino Garcia, Joe Auricchio, Jonathan Babb, Michael Taylor, and Steven Swanson. GreenDroid: A Mobile Application Processor for a Future of Dark Silicon. In *HOTCHIPS*, 2010.

[61] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor. The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future. *Micro, IEEE*, pages 86–95, March 2011.

[62] Nathan Goulding-Hotta. *Specialization as a Candle in the Dark Silicon Regime*. PhD thesis, University of California, San Diego, 2020.

[63] Nathan Goulding-Hotta, Jack Sampson, Qiaoshi Zheng, Vikram Bhatt, Steven Swanson, and Michael Taylor. GreenDroid: An Architecture for the Dark Silicon Age. In *Asia and South Pacific Design Automation Conference (ASPDAC)*, 2012.

[64] Cong Guo, Feng Cheng, Zhixu Du, James Kiessling, Jonathan Ku, Shiyu Li, Ziru Li, Mingyuan Ma, Tergel Molom-Ochir, Benjamin Morris, et al. A survey: Collaborative hardware and software design in the era of large language models. *IEEE Circuits and Systems Magazine*, 25(1):35–57, 2025.

[65] Anshuman Gupta, Jack Sampson, and Michael Bedford Taylor. DR-SNUCA: An energy-scalable dynamically partitioned cache. In *International Conference on Computer Design (ICCD)*, 2013.

[66] Anshuman Gupta, Jack Sampson, and Michael Bedford Taylor. Time Cube: A Manycore Embedded Processor with Interference-Agnostic Progress Tracking. In *International Conference On Embedded Computer Systems: Architectures, Modeling And Simulation (SAMOS)*, 2013.

[67] Anshuman Gupta, Jack Sampson, and Michael Bedford Taylor. Qualitytime: A simple online technique for quantifying multicore execution efficiency. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014.

[68] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. Eie: efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, page 243–254. IEEE Press, 2016.

[69] Byron Hawkins, Brian Demsky, and Michael Bedford Taylor. BlackBox: Lightweight Security Monitoring for COTS Binaries. In *Code Generation and Optimization*, 2016.

[70] Byron Hawkins, Brian Demsky, and Michael Bedford Taylor. A Runtime Approach to Security and Privacy. In *European Security and Privacy*, 2016.

[71] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *arXiv preprint arXiv:1512.03385*, December 2015.

[72] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining Improvements in Deep Reinforcement Learning. *arXiv preprint arXiv:1710.02298*, October 2017.

[73] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[74] Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, and Yuxiong He. DeepSpeed-FastGen: High-throughput Text Generation for LLMs via MII and DeepSpeed-Inference, 2024.

[75] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. Distributed Prioritized Experience Replay. *arXiv preprint arXiv:1803.00933*, March 2018. arXiv: 1803.00933.

[76] Hu, Zhu, Taylor, and Cheng. FPGA Global Routing Architecture Optimization Using a Multicommodity Flow Approach . In *ICCD*, 2007.

[77] Qijing Huang, Aravind Kalaiah, Minwoo Kang, James Demmel, Grace Dinh, John Wawrzynek, Thomas Norell, and Yakun Sophia Shao. CoSA: Scheduling by Constrained Optimization for Spatial Accelerators. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 554–566, June 2021. ISSN: 2575-713X.

[78] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. *arXiv:1811.06965 [cs]*, July 2019. arXiv: 1811.06965.

[79] Donghwan Jeon, Saturnino Garcia, Chris Louie, Sravanthi Kota Venkata, and Michael Bedford Taylor. Kremlin: Like gprof, but for Parallelization. In *Principles and Practice of Parallel Programming (PPoPP)*, 2011.

[80] Donghwan Jeon, Saturnino Garcia, Chris Louie, and Michael Bedford Taylor. Kismet: Parallel Speedup Estimates for Serial Programs. In *Conference on Object-Oriented Programming, Systems, Language and Applications (OOPSLA)*, 2011.

[81] Donghwan Jeon, Saturnino Garcia, Chris Louie, and Michael Bedford Taylor. Parkour: Parallel Speedup Estimates from Serial Code. In *USENIX Workshop on Hot Topics in Parallelism (HOTPAR)*, 2011.

[82] Donghwan Jeon, Saturnino Garcia, and Michael Bedford Taylor. Skadu: Efficient Vector Shadow Memories for Poly-scopic Program Analysis. In *Conference on Code Generation and Optimization (CGO)*, 2013.

[83] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, Tiegang Chen, Guangxiao Hu, Shaohuai Shi, and Xiaowen Chu. Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes. *arXiv:1807.11205 [cs]*, 2018.

[84] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, et al. TPU v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2023.

[85] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. Ten Lessons From Three Generations Shaped Google's TPUv4i : Industrial Product. In *Proceedings of the 48th International Symposium on Computer Architecture (ISCA)*, 2021.

[86] Norman P. Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. A Domain-specific Supercomputer for Training Deep Neural Networks. *Communications of the ACM*, 2020.

[87] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon.

In-datacenter performance analysis of a tensor processing unit. *SIGARCH Comput. Archit. News*, 45(2):1–12, June 2017.

[88] Dai Cheol Jung. Caches for Complex Open Source System-on-Chip Designs. Master's thesis, University of Washington, 2019.

[89] Dai Cheol Jung, Scott Davidson, Chun Zhao, Dustin Richmond, and Michael Bedford Taylor. Ruche Networks: Wire-Maximal, No-Fuss NoCs. In *NOCS*, 2020.

[90] Dai Cheol Jung, Scott Davidson, Chun Zhao, Dustin Richmond, and Michael Bedford Taylor. Ruche Networks: Wire-Maximal, No-Fuss NoCs. In *2020 14th IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, page 8, September 2020.

[91] Dai Cheol Jung, Max Ruttenberg, Paul Gao, Scott Davidson, Daniel Petrisko, Kangli Li, Aditya K Kamath, Lin Cheng, Shaolin Xie, Peitian Pan, Zhongyuan Zhao, Zichao Yue, Bandhav Veluri, Sripathi Muralitharan, Adrian Sampson, Andrew Lumsdaine, Zhiru Zhang, Christopher Batten, Mark Oskin, Dustin Richmond, and Michael Bedford Taylor. Scalable, Programmable and Dense: The HammerBlade Open-Source RISC-V Manycore. In *ISCA*, 2024.

[92] Ajaykumar Kannan, Natalie Enright Jerger, and Gabriel H. Loh. Enabling Interposer-based Disintegration of Multi-core Processors. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, 2015.

[93] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling Laws for Neural Language Models. *arXiv:2001.08361 [cs, stat]*, January 2020. arXiv: 2001.08361.

[94] Steven Kapturowski, Georg Ostrovski, John Quan, Remi Munos, and Will Dabney. Recurrent Experience Replay in Distributed Reinforcement Learning. page 19, 2019.

[95] Moein Khazraee. *Reducing the development cost of customized cloud infrastructure.* PhD thesis, University of California, San Diego, 2020.

[96] Moein Khazraee, Luis Vega, Ikuo Magaki, and Michael Taylor. Specializing a Planet's Computation: ASIC Clouds. *IEEE Micro*, May 2017.

[97] Moein Khazraee, Lu Zhang, Luis Vega, and Michael Taylor. Moonwalk: NRE Optimization in ASIC Clouds or, accelerators will use old silicon. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[98] Moein Khazraee, Lu Zhang, Luis Vega, and Michael Bedford Taylor. Moonwalk: NRE Optimization in ASIC Clouds. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[99] Changhyeon Kim, Sanghoon Kang, Sungpill Choi, Dongjoo Shin, Youngwoo Kim, and Hoi-Jun Yoo. An Energy-Efficient Deep Reinforcement Learning Accelerator With Transposable PE Array and Experience Compression. *IEEE Solid-State Circuits Letters*, 2(11):228–231, November 2019. Conference Name: IEEE Solid-State Circuits Letters.

[100] Hanjoon Kim, Younggeun Choi, Junyoung Park, Byeongwook Bae, Hyunmin Jeong, Sang Min Lee, Jeseung Yeon, Minho Kim, Changjae Park, Boncheol Gu, Changman Lee, Jaeick Bae, SungGyeong Bae, Yojung Cha, Wooyoung Choe, Jonguk Choi, Juho Ha, Hyuck Han, Namoh Hwang, Seokha Hwang, Kiseok Jang, Haechan Je, Hojin Jeon, Jaewoo Jeon, Hyunjun Jeong, Yeonsu Jung, Dongok Kang, Hyewon Kim, Minjae Kim, Muhwan Kim, Sewon Kim, Suhyung Kim, Won Kim, Yong Kim, Youngsik Kim, Younki Ku, Jeong Ki Lee, Juyun Lee, Kyungjae Lee, Seokho Lee, Minwoo Noh, Hyuntaek Oh, Gyunghee Park, Sanguk Park, Jimin Seo, Jungyoung Seong, June Paik, Nuno P. Lopes, and Sungjoo Yoo. Tcp: A tensor contraction processor for ai workloads industrial product*. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 890–902, 2024.

[101] Jason Kim, Michael B. Taylor, Jason Miller, and David Wentzlaff. Energy Characterization of a Tiled Architecture Processor with On-Chip Networks. In *International Symposium on Low Power Electronics and Design (ISLPED)*, August 2003.

[102] Simon Knowles. Graphcore Colossus Mk2 IPU. In *Hot Chips 33 Symposium*, 2021.

[103] Anton Korzh, Brian Pharris, Nick Comly, Ashraf Eassa, and Amr Elmeleegy. 3x Faster AllReduce with NVSwitch and TensorRT-LLM MultiShot, 2024.

[104] Atli Kosson, Vitaliy Chiley, Abhinav Venigalla, Joel Hestness, and Urs Köster. Pipelined Backpropagation at Scale: Training Large Models without Batches. *arXiv:2003.11666 [cs, stat]*, April 2021. arXiv: 2003.11666.

[105] Sravanthi Kota Venkata, IkkJin Ahn, Donghwan Jeon, Anshuman Gupta, and Michael Taylor. SD-VBS: The San Diego Vision Benchmark Suite. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[106] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[107] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *NeurIPS*, volume 25. Curran Associates, Inc., 2012.

[108] Joyjit Kundu, Wenzhe Guo, Ali BanaGozar, Udari De Alwis, Sourav Sengupta, Puneet Gupta, and Arindam Mallik. Performance Modeling and Workload Analysis of Distributed Large Language Model Training and Inference . In *IISWC*, 2024.

[109] Hsiang-Tsung Kung. *Why systolic architecture?* Design Research Center, Carnegie-Mellon University, 1982.

[110] Karol Kurach, Anton Raichuk, Piotr Stańczyk, Michał Zając, Olivier Bachem, Lasse Espeholt, Carlos Riquelme, Damien Vincent, Marcin Michalski, Olivier Bousquet, and Sylvain Gelly. Google Research Football: A Novel Reinforcement Learning Environment. *arXiv:1907.11180 [cs, stat]*, April 2020. arXiv: 1907.11180.

[111] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. Understanding Reuse, Performance, and Hardware Cost of DNN Dataflows: A Data-Centric Approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, October 2019.

[112] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. MAERI: Enabling Flexible Dataflow Mapping Over DNN Accelerators via Reconfigurable Interconnects. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 461–475, Williamsburg, VA, March 2018.

[113] Lambda. The Best Prices for Cloud GPUs. `https://lambdalabs.com/service/gpu-cloud`, 2023.

[114] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[115] Kangli Li. An Open Source Non-Blocking Manycore L2 Cache. Master's thesis, University of Washington, 2024.

[116] Mu Li, David G Andersen, Alexander Smola, and Kai Yu. Communication Efficient Distributed Machine Learning with the Parameter Server. In *Proceedings of the 27th International Conference on Neural Information Processing Systems (NeurIPS)*, Montreal, Canada, 2014.

[117] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *arXiv:2006.15704 [cs]*, June 2020. arXiv: 2006.15704.

[118] Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. Sequence parallelism: Long sequence training from system perspective, 2022.

[119] Youjie Li, Iou-Jen Liu, Yifan Yuan, Deming Chen, Alexander Schwing, and Jian Huang. Accelerating Distributed Reinforcement Learning with In-Switch Computing. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*, pages 279–291, Phoenix, AZ, June 2019.

[120] Youjie Li, Jongse Park, Mohammad Alian, Yifan Yuan, Zheng Qu, Peitian Pan, Ren Wang, Alexander Schwing, Hadi Esmaeilzadeh, and Nam Sung Kim. A Network-Centric Hardware/Algorithm Co-Design to Accelerate Distributed Training of Deep Neural Networks. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 175–188, Fukuoka, Japan, October 2018.

[121] Ryan Liu and Chuang Feng. AI Compute Chip from Enflame. In *Hot Chips 33 Symposium*, 2021.

[122] LlamaTeam. The llama 3 herd of models, 2024.

[123] Ikuo Magaki, Moein Khazraee, Luis Vega Gutierrez, and Michael Bedford Taylor. ASIC Clouds: Specializing the Datacenter. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, 2016.

[124] Ikuo Magaki, Moein Khazraee, Luis Vega, and Michael Taylor. ASIC Clouds: Specializing the Datacenter. In *International Symposium on Computer Architecture (ISCA)*, 2016.

[125] James L McClelland and Randall C O'Reilly. Why There Are Complementary Learning Systems in the Hippocampus and Neocortex:insightsfrom the Successesand Failuresof Connectionist Models of Learning and Memory. *Psychological Review*, 102(3):419–457, 1995.

[126] Microsoft. Azure LLM Inference Trace 2023. `https://github.com/Azure/AzurePublicDataset/blob/master/AzureLLMInferenceDataset2023.md`, 2024.

[127] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. *arXiv preprint arXiv:1602.01783*, June 2016. arXiv: 1602.01783.

[128] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen

King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-Level Control Through Deep Reinforcement Learning. *Nature*, 518(7540):529–533, February 2015.

[129] Maryam Mohsin. 10 Google Search Statistics You Need to Know in 2023. `https://www.oberlo.com/blog/google-search-statistics#:~:text=We%20know%20that%20there%20are,Internet%20Live%20Stats%2C%202022)`, 2023.

[130] Sripathi Muralitharan. TinyParrot: An Integration-Optimized Linux-Capable Host Multicore. Master's thesis, University of Washington, 2021.

[131] Yuyao Niu, Zhengyang Lu, Meichen Dong, Zhou Jin, Weifeng Liu, and Guangming Tan. TileSpMV: A Tiled Algorithm for Sparse Matrix-Vector Multiplication on GPUs. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021.

[132] NVIDIA. Nvidia a100 tensor core gpu architecture, 2020. Accessed: Nov. 2024. [Online].

[133] NVIDIA. Nvidia h100 tensor core gpu architecture, 2023. Accessed: Nov. 2024. [Online].

[134] NVIDIA. DGX B200. `https://www.nvidia.com/en-us/data-center/dgx-b200/`, 2024.

[135] NVIDIA. NVLink and NVLink Switch. `https://www.nvidia.com/en-us/data-center/nvlink/`, 2024.

[136] OpenAI. Introducing ChatGPT. `https://openai.com/blog/chatgpt`, 2022.

[137] OpenAI. Hello GPT-4o. `https://openai.com/index/hello-gpt-4o/`, 2024.

[138] OpenAI. Sora. `https://openai.com/sora/`, 2025.

[139] Joseph O'Neill, Barty Pleydell-Bouverie, David Dupret, and Jozsef Csicsvari. Play It Again: Reactivation of Waking Experience and Memory. *Trends in Neurosciences*, 33(5):220–229, May 2010.

[140] S. Pal, D. Park, S. Feng, P. Gao, J. Tan, A. Rovinski, S. Xie, C. Zhao, A. Amarnath, T. Wesley, J. Beaumont, K. Chen, C. Chakrabarti, M. Taylor, T. Mudge, D. Blaauw, H. Kim, and R. Dreslinski. A 7.3 M Output Non-Zeros/J Sparse Matrix-Matrix Multiplication Accelerator using Memory Reconfiguration in 40 nm. In *Symposium on VLSI Circuits*, pages C150–C151, 2019.

[141] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W. Keckler, and Joel Emer. Timeloop: A Systematic Approach to DNN Accelerator Evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 304–315, March 2019.

[142] D. Park, S. Pal, S. Feng, P. Gao, J. Tan, A. Rovinski, S. Xie, C. Zhao, A. Amarnath, T. Wesley, J. Beaumont, K. Chen, C. Chakrabarti, M. B. Taylor, T. Mudge, D. Blaauw, H. Kim, and R. G. Dreslinski. A 7.3 M Output Non-Zeros/J, 11.7 M Output Non-Zeros/GB Reconfigurable Sparse Matrix–Matrix Multiplication Accelerator. *IEEE Journal of Solid-State Circuits*, pages 933–944, April 2020.

[143] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2024.

[144] David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. Carbon Emissions and Large Neural Network Training. *arXiv:2104.10350 [cs]*, April 2021. arXiv: 2104.10350.

[145] Huwan Peng, Scott Davidson, Richard Shi, Shuaiwen Leon Song, and Michael Taylor. Chiplet Cloud: Building AI Supercomputers for Serving Large Generative Language Models. *arXiv:2307.02666 [cs]*, 2024.

[146] Huwan Peng, Scott Davidson, Richard Shi, and Michael Taylor. Reallm: A trace-driven framework for rapid simulation of large-scale llm inference. In *2025 IEEE 36th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2025.

[147] D. Petrisko, F. Gilani, M. Wyse, D. C. Jung, S. Davidson, P. Gao, C. Zhao, Z. Azad, S. Canakci, B. Veluri, T. Guarino, A. Joshi, M. Oskin, and M. B. Taylor. BlackParrot: An Agile Open-Source RISC-V Multicore for Accelerator SoCs. *IEEE Micro*, pages 93–102, Jul/Aug. 2020.

[148] Daniel Petrisko, Chun Zhao, Scott Davidson, Paul Gao, Dustin Richmond, and Michael Bedford Taylor. NoC Symbiosis. In *NOCS*, 2020.

[149] Daniel Petrisko, Chun Zhao, Scott Davidson, Paul Gao, Dustin Richmond, and Michael Bedford Taylor. NoC Symbiosis. In *IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, 2020.

[150] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently Scaling Transformer Inference. *arXiv:2211.05102 [cs]*, 2022.

[151] John W. Poulton, John M. Wilson, Walker J. Turner, Brian Zimmer, Xi Chen, Sudhir S. Kudva, Sanquan Song, Stephen G. Tell, Nikola Nedovic, Wenxu Zhao, Sunil R. Sudhakaran, C. Thomas Gray, and William J. Dally. A 1.17-pJ/b, 25-Gb/s/pin Ground-Referenced Single-Ended Serial Link for Off- and On-Package Communication Using a Process- and Temperature-Adaptive Voltage Regulator. *IEEE Journal of Solid-State Circuits*, 2019.

[152] Raghu Prabhakar and Sumti Jairath. SambaNova SN10 RDU: Accelerating Software 2.0 with Dataflow. In *Hot Chips 33 Symposium*, 2021.

[153] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language Models are Unsupervised Multitask Learners. *OpenAI blog*, 2019.

[154] Jack W. Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, Eliza Rutherford, Tom Hennigan, Jacob Menick, Albin Cassirer, Richard Powell, George van den Driessche, Lisa Anne Hendricks, Maribeth Rauh, Po-Sen Huang, Amelia Glaese, Johannes Welbl, Sumanth Dathathri, Saffron Huang, Jonathan Uesato, John Mellor, Irina Higgins, Antonia Creswell, Nat McAleese, Amy Wu, Erich Elsen, Siddhant Jayakumar, Elena Buchatskaya, David Budden, Esme Sutherland, Karen Simonyan, Michela Paganini, Laurent Sifre, Lena Martens, Xiang Lorraine Li, Adhiguna Kuncoro, Aida Nematzadeh, Elena Gribovskaya, Domenic Donato, Angeliki Lazaridou, Arthur Mensch, Jean-Baptiste Lespiau, Maria Tsimpoukelli, Nikolai Grigorev, Doug Fritz, Thibault Sottiaux, Mantas Pajarskas, Toby Pohlen, Zhitao Gong, Daniel Toyama, Cyprien de Masson d'Autume, Yujia Li, Tayfun Terzi, Vladimir Mikulik, Igor Babuschkin, Aidan Clark, Diego de Las Casas, Aurelia Guy, Chris Jones, James Bradbury, Matthew Johnson, Blake Hechtman, Laura Weidinger, Iason Gabriel, William Isaac, Ed Lockhart, Simon Osindero, Laura Rimell, Chris Dyer, Oriol Vinyals, Kareem Ayoub, Jeff Stanway, Lorrayne Bennett, Demis Hassabis, Koray Kavukcuoglu, and Geoffrey Irving. Scaling Language Models: Methods, Analysis & Insights from Training Gopher. *arXiv:2112.11446 [cs]*, 2022.

[155] Robert "Max" Ramstad. Enabling Vector Load and Store instructions on HammerBlade Architecture. Master's thesis, University of Washington, 2024.

[156] Shashank Vijaya Ranga. ParrotPiton and ZynqParrot: FPGA Enablements for the BlackParrot RISC-V Processor. Master's thesis, University of Washington, 2021.

[157] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. Survey of machine learning accelerators. In *2020 IEEE high performance extreme computing conference (HPEC)*, pages 1–12. IEEE, 2020.

[158] A. Rovinski, C. Zhao, K. Al-Hawaj, P. Gao, S. Xie, C. Torng, S. Davidson, A. Amarnath, L. Vega, B. Veluri, A. Rao, T. Ajayi, J. Puscar, S. Dai, R. Zhao, D. Richmond,

Z. Zhang, I. Galton, C. Batten, M. B. Taylor, and R. G. Dreslinski. A 1.4 GHz 695 Giga Risc-V Inst/s 496-Core Manycore Processor With Mesh On-Chip Network and an All-Digital Synthesized PLL in 16nm CMOS. In *2019 Symposium on VLSI Circuits*, pages C30–C31, 2019.

[159] A. Rovinski, C. Zhao, K. Al-Hawaj, P. Gao, S. Xie, C. Torng, S. Davidson, A. Amarnath, L. Vega, B. Veluri, A. Rao, T. Ajayi, J. Puscar, S. Dai, R. Zhao, D. Richmond, Z. Zhang, I. Galton, C. Batten, M. B. Taylor, and R. G. Dreslinski. Evaluating Celerity: A 16-nm 695 Giga-RISC-V Instructions/s Manycore Processor With Synthesizable PLL. *IEEE Solid-State Circuits Letters*, 2(12):289–292, 2019.

[160] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. pearson, 2016.

[161] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution Strategies as a Scalable Alternative to Reinforcement Learning. *arXiv preprint arXiv:1703.03864*, September 2017.

[162] Jack Sampson, Manish Arora, Nathan Goulding-Hotta, Ganesh Venkatesh, Jonathan Babb, Vikram Bhatt, Michael Bedford Taylor, and Steven Swanson. An Evaluation of Selective Depipelining for FPGA-based Energy-Reducing Irregular Code Coprocessors. In *Conference on Field Programmable Logic and Applications (FPL)*, 2011.

[163] Jack Sampson, Ganesh Venkatesh, Nathan Goulding-Hotta, Saturnino Garcia, Steven Swanson, and Michael Bedford Taylor. Efficient Complex Operators for Irregular Codes. In *High Performance Computing Architecture (HPCA)*, 2011.

[164] Peter Sanders, Jochen Speck, and Jesper Larsson Träff. Two-tree algorithms for full bandwidth broadcast, reduction and scan. *Parallel Computing*, 2009.

[165] Abu Sebastian, Manuel Le Gallo, Riduan Khaddam-Aljameh, and Evangelos Eleftheriou. Memory devices and applications for in-memory computing. *Nature nanotechnology*, 15(7):529–544, 2020.

[166] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. Isaac: a convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, page 14–26. IEEE Press, 2016.

[167] Noam Shazeer. Fast Transformer Decoding: One Write-Head is All You Need. *arXiv:1911.02150 [cs]*, 2019.

[168] Noam Shazeer. GLU Variants Improve Transformer. *arXiv:2002.05202 [cs]*, 2020.

[169] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer, 2017.

[170] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv:1909.08053 [cs]*, 2020.

[171] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529(7587):484–489, January 2016.

[172] Misha Smelyanskiy. Zion: Facebook Next- Generation Large Memory Training Platform. In *Hot Chips 31 Symposium*, 2019.

[173] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, Elton Zhang, Rewon Child, Reza Yazdani Aminabadi, Julie Bernauer, Xia Song, Mohammad Shoeybi, Yuxiong He, Michael Houston, Saurabh Tiwary, and Bryan Catanzaro. Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model. *arXiv:2201.11990 [cs]*, 2022.

[174] Suno. Suno AI. `https://suno.com/home/`, 2025.

[175] Steven Swanson and Michael Taylor. GreenDroid: Exploring the next evolution for smartphone application processors. In *IEEE Communications Magazine*, March 2011.

[176] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. *Efficient processing of deep neural networks*. Springer, 2020.

[177] MB Taylor, J Kim, J Miller, F Ghodrat, B Greenwald, P Johnson, W Lee, A Ma, N Shnidman, V Strumpen, et al. The raw processor-a scalable 32-bit fabric for embedded and general purpose computing. In *Proceedings of Hot Chips XIII*, 2001.

[178] Michael Taylor. *Tiled Microprocessors*. PhD thesis, Massachusetts Institute of Technology, 2007.

[179] Michael Taylor. A Landscape of the New Dark Silicon Design Regime. *Micro, IEEE*, Sept-Oct. 2013.

[180] Michael Taylor. A Landscape of the New Dark Silicon Design Regime. In *Design Automation and Test in Europe*, April 2014.

[181] Michael Taylor. The Evolution of Bitcoin Hardware. *Computer, IEEE*, Sept-Oct. 2017.

[182] Michael B. Taylor. Is Dark Silicon Useful? Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse. In *Design Automation Conference (DAC)*, 2012.

[183] Michael B. Taylor. Bitcoin and the Age of Bespoke Silicon. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2013.

[184] Michael B. Taylor. BaseJump STL: SystemVerilog needs a Standard Template Library for Hardware Design. In *Design Automation Conference*, June 2018.

[185] Michael B. Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. In *IEEE Micro*, March 2002.

[186] Michael B. Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffmann, Paul Johnson, Walter Lee, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Saman Amarasinghe, and Anant Agarwal. A 16-issue Multiple-Program-Counter Microprocessor with Point-to-Point Scalar Operand Network. In *IEEE International Solid-State Circuits Conference (ISSCC)*, February 2003.

[187] Michael B. Taylor, Walter Lee, Saman Amarasinghe, and Anant Agarwal. Scalar Operand Networks: On-Chip Interconnect for ILP in Partitioned Architectures. In *International Symposium on High Performance Computer Architecture (HPCA)*, February 2003.

[188] Michael B. Taylor, Walter Lee, Saman Amarasinghe, and Anant Agarwal. Scalar Operand Networks. In *IEEE Transactions on Parallel and Distributed Systems*, February 2005.

[189] Michael B. Taylor, Walter Lee, Saman Amarasinghe, and Anant Agarwal. Scalar Operand Networks. In *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, February 2005.

[190] Michael B. Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *International Symposium on Computer Architecture (ISCA)*, June 2004.

[191] Michael Bedford Taylor. Geocomputers and the Commercial Borg. In *SIGARCH Computer Architecture Today*, Dec 2017.

[192] Michael Bedford Taylor. BaseJump STL: SystemVerilog Needs a Standard Template Library for Hardware Design. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, San Francisco, CA, June 2018.

[193] Michael Bedford Taylor. Your agile open source HW stinks (because it is not a system). In *ICCAD*, 2020.

[194] Michael Bedford Taylor, Luis Vega, Moein Khazraee, Ikuo Magaki, Scott Davidson, and Dustin Richmond. ASIC clouds: Specializing the datacenter for planet-scale applications. *CACM*, pages 103–109, 2020.

[195] Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, Soroosh Mariooryad, Yifan Ding, Xinyang Geng, Fred Alcober, Roy Frostig, Mark Omernick, Lexi Walker, Cosmin Paduraru, Christina Sorokin, Andrea Tacchetti, Colin Gaffney, Samira Daruki, Olcan Sercinoglu, Zach Gleicher, Juliette Love, Paul Voigtlaender, Rohan Jain, Gabriela Surita, Kareem Mohamed, Rory Blevins, Junwhan Ahn, Tao Zhu, Kornraphop Kawintiranon, Orhan Firat, Yiming Gu, Yujing Zhang, Matthew Rahtz, Manaal Faruqui, Natalie Clay, Justin Gilmer, JD Co-Reyes, Ivo Penchev, Rui Zhu, Nobuyuki Morioka, Kevin Hui, Krishna Haridasan, Victor Campos, Mahdis Mahdieh, Mandy Guo, Samer Hassan, Kevin Kilgour, Arpi Vezer, Heng-Tze Cheng, Raoul de Liedekerke, Siddharth Goyal, Paul Barham, DJ Strouse, Seb Noury, Jonas Adler, Mukund Sundararajan, Sharad Vikram, Dmitry Lepikhin, Michela Paganini, Xavier Garcia, Fan Yang, Dasha Valter, Maja Trebacz, Kiran Vodrahalli, Chulayuth Asawaroengchai, Roman Ring, Norbert Kalb, Livio Baldini Soares, Siddhartha Brahma, David Steiner, Tianhe Yu, Fabian Mentzer, Antoine He, Lucas Gonzalez, Bibo Xu, Raphael Lopez Kaufman, Laurent El Shafey, Junhyuk Oh, Tom Hennigan, George van den Driessche, Seth Odoom, Mario Lucic, Becca Roelofs, Sid Lall, Amit Marathe, Betty Chan, Santiago Ontanon, Luheng He, Denis Teplyashin, Jonathan Lai, Phil Crone, Bogdan Damoc, Lewis Ho, Sebastian Riedel, Karel Lenc, Chih-Kuan Yeh, Aakanksha Chowdhery, Yang Xu, Mehran Kazemi, Ehsan Amid, Anastasia Petrushkina, Kevin Swersky, Ali Khodaei, Gowoon Chen, Chris Larkin, Mario Pinto, Geng Yan, Adria Puigdomenech Badia, Piyush Patil, Steven Hansen, Dave Orr, Sebastien M. R. Arnold, Jordan Grimstad, Andrew Dai, Sholto Douglas, Rishika Sinha, Vikas Yadav, Xi Chen, Elena Gribovskaya, Jacob Austin, Jeffrey Zhao, Kaushal Patel, Paul Komarek, Sophia Austin, Sebastian Borgeaud, Linda Friso, Abhimanyu Goyal, Ben Caine, Kris Cao, Da-Woon Chung, Matthew Lamm, Gabe Barth-Maron, Thais Kagohara, Kate Olszewska, Mia Chen, Kaushik Shivakumar, Rishabh Agarwal, Harshal Godhia, Ravi Rajwar, Javier Snaider, Xerxes Dotiwalla, Yuan Liu, Aditya Barua, Victor Ungureanu, Yuan Zhang, Bat-Orgil Batsaikhan, Mateo Wirth, James Qin, Ivo Danihelka, Tulsee Doshi, Martin Chadwick, Jilin Chen, Sanil Jain, Quoc Le, Arjun Kar, Madhu Gurumurthy, Cheng Li, Ruoxin Sang, Fangyu Liu, Lampros Lamprou, Rich Munoz, Nathan Lintz, Harsh Mehta, Heidi Howard, Malcolm Reynolds, Lora Aroyo, Quan Wang, Lorenzo Blanco, Albin Cassirer, Jordan Griffith, Dipanjan

Das, Stephan Lee, Jakub Sygnowski, Zach Fisher, James Besley, Richard Powell, Zafarali Ahmed, Dominik Paulus, David Reitter, Zalan Borsos, Rishabh Joshi, Aedan Pope, Steven Hand, Vittorio Selo, Vihan Jain, Nikhil Sethi, Megha Goel, Takaki Makino, Rhys May, Zhen Yang, Johan Schalkwyk, Christina Butterfield, Anja Hauth, Alex Goldin, Will Hawkins, Evan Senter, Sergey Brin, Oliver Woodman, Marvin Ritter, Eric Noland, Minh Giang, Vijay Bolina, Lisa Lee, Tim Blyth, Ian Mackinnon, Machel Reid, Obaid Sarvana, David Silver, Alexander Chen, Lily Wang, Loren Maggiore, Oscar Chang, Nithya Attaluri, Gregory Thornton, Chung-Cheng Chiu, Oskar Bunyan, Nir Levine, Timothy Chung, Evgenii Eltyshev, Xiance Si, Timothy Lillicrap, Demetra Brady, Vaibhav Aggarwal, Boxi Wu, Yuanzhong Xu, Ross McIlroy, Kartikeya Badola, Paramjit Sandhu, Erica Moreira, Wojciech Stokowiec, Ross Hemsley, Dong Li, Alex Tudor, Pranav Shyam, Elahe Rahimtoroghi, Salem Haykal, Pablo Sprechmann, Xiang Zhou, Diana Mincu, Yujia Li, Ravi Addanki, Kalpesh Krishna, Xiao Wu, Alexandre Frechette, Matan Eyal, Allan Dafoe, Dave Lacey, Jay Whang, Thi Avrahami, Ye Zhang, Emanuel Taropa, Hanzhao Lin, Daniel Toyama, Eliza Rutherford, Motoki Sano, HyunJeong Choe, Alex Tomala, Chalence Safranek-Shrader, Nora Kassner, Mantas Pajarskas, Matt Harvey, Sean Sechrist, Meire Fortunato, Christina Lyu, Gamaleldin Elsayed, Chenkai Kuang, James Lottes, Eric Chu, Chao Jia, Chih-Wei Chen, Peter Humphreys, Kate Baumli, Connie Tao, Rajkumar Samuel, Cicero Nogueira dos Santos, Anders Andreassen, Nemanja Rakićević, Dominik Grewe, Aviral Kumar, Stephanie Winkler, Jonathan Caton, Andrew Brock, Sid Dalmia, Hannah Sheahan, Iain Barr, Yingjie Miao, Paul Natsev, Jacob Devlin, Feryal Behbahani, Flavien Prost, Yanhua Sun, Artiom Myaskovsky, Thanumalayan Sankaranarayana Pillai, Dan Hurt, Angeliki Lazaridou, Xi Xiong, Ce Zheng, Fabio Pardo, Xiaowei Li, Dan Horgan, Joe Stanton, Moran Ambar, Fei Xia, Alejandro Lince, Mingqiu Wang, Basil Mustafa, Albert Webson, Hyo Lee, Rohan Anil, Martin Wicke, Timothy Dozat, Abhishek Sinha, Enrique Piqueras, Elahe Dabir, Shyam Upadhyay, Anudhyan Boral, Lisa Anne Hendricks, Corey Fry, Josip Djolonga, Yi Su, Jake Walker, Jane Labanowski, Ronny Huang, Vedant Misra, Jeremy Chen, RJ Skerry-Ryan, Avi Singh, Shruti Rijhwani, Dian Yu, Alex Castro-Ros, Beer Changpinyo, Romina Datta, Sumit Bagri, Arnar Mar Hrafnkelsson, Marcello Maggioni, Daniel Zheng, Yury Sulsky, Shaobo Hou, Tom Le Paine, Antoine Yang, Jason Riesa, Dominika Rogozinska, Dror Marcus, Dalia El Badawy, Qiao Zhang, Luyu Wang, Helen Miller, Jeremy Greer, Lars Lowe Sjos, Azade Nova, Heiga Zen, Rahma Chaabouni, Mihaela Rosca, Jiepu Jiang, Charlie Chen, Ruibo Liu, Tara Sainath, Maxim Krikun, Alex Polozov, Jean-Baptiste Lespiau, Josh Newlan, Zeyncep Cankara, Soo Kwak, Yunhan Xu, Phil Chen, Andy Coenen, Clemens Meyer, Katerina Tsihlas, Ada Ma, Juraj Gottweis, Jinwei Xing, Chenjie Gu, Jin Miao, Christian Frank, Zeynep Cankara, Sanjay Ganapathy, Ishita Dasgupta, Steph Hughes-Fitt, Heng Chen, David Reid, Keran Rong, Hongmin Fan, Joost van Amersfoort, Vincent Zhuang, Aaron Cohen, Shixiang Shane Gu, Anhad Mohananey, Anastasija Ilic, Taylor Tobin, John Wieting, Anna Bortsova, Phoebe Thacker, Emma Wang, Emily Caveness, Justin Chiu, Eren Sezener, Alex Kaskasoli, Steven Baker, Katie Millican, Mohamed Elhawaty, Kostas Aisopos, Carl Lebsack, Nathan Byrd, Hanjun Dai, Wenhao Jia, Matthew Wiethoff, Elnaz Davoodi, Albert

Weston, Lakshman Yagati, Arun Ahuja, Isabel Gao, Golan Pundak, Susan Zhang, Michael Azzam, Khe Chai Sim, Sergi Caelles, James Keeling, Abhanshu Sharma, Andy Swing, YaGuang Li, Chenxi Liu, Carrie Grimes Bostock, Yamini Bansal, Zachary Nado, Ankesh Anand, Josh Lipschultz, Abhijit Karmarkar, Lev Proleev, Abe Ittycheriah, Soheil Hassas Yeganeh, George Polovets, Aleksandra Faust, Jiao Sun, Alban Rrustemi, Pen Li, Rakesh Shivanna, Jeremiah Liu, Chris Welty, Federico Lebron, Anirudh Baddepudi, Sebastian Krause, Emilio Parisotto, Radu Soricut, Zheng Xu, Dawn Bloxwich, Melvin Johnson, Behnam Neyshabur, Justin Mao-Jones, Renshen Wang, Vinay Ramasesh, Zaheer Abbas, Arthur Guez, Constant Segal, Duc Dung Nguyen, James Svensson, Le Hou, Sarah York, Kieran Milan, Sophie Bridgers, Wiktor Gworek, Marco Tagliasacchi, James Lee-Thorp, Michael Chang, Alexey Guseynov, Ale Jakse Hartman, Michael Kwong, Ruizhe Zhao, Sheleem Kashem, Elizabeth Cole, Antoine Miech, Richard Tanburn, Mary Phuong, Filip Pavetic, Sebastien Cevey, Ramona Comanescu, Richard Ives, Sherry Yang, Cosmo Du, Bo Li, Zizhao Zhang, Mariko Iinuma, Clara Huiyi Hu, Aurko Roy, Shaan Bijwadia, Zhenkai Zhu, Danilo Martins, Rachel Saputro, Anita Gergely, Steven Zheng, Dawei Jia, Ioannis Antonoglou, Adam Sadovsky, Shane Gu, Yingying Bi, Alek Andreev, Sina Samangooei, Mina Khan, Tomas Kocisky, Angelos Filos, Chintu Kumar, Colton Bishop, Adams Yu, Sarah Hodkinson, Sid Mittal, Premal Shah, Alexandre Moufarek, Yong Cheng, Adam Bloniarz, Jaehoon Lee, Pedram Pejman, Paul Michel, Stephen Spencer, Vladimir Feinberg, Xuehan Xiong, Nikolay Savinov, Charlotte Smith, Siamak Shakeri, Dustin Tran, Mary Chesus, Bernd Bohnet, George Tucker, Tamara von Glehn, Carrie Muir, Yiran Mao, Hideto Kazawa, Ambrose Slone, Kedar Soparkar, Disha Shrivastava, James Cobon-Kerr, Michael Sharman, Jay Pavagadhi, Carlos Araya, Karolis Misiunas, Nimesh Ghelani, Michael Laskin, David Barker, Qiujia Li, Anton Briukhov, Neil Houlsby, Mia Glaese, Balaji Lakshminarayanan, Nathan Schucher, Yunhao Tang, Eli Collins, Hyeontaek Lim, Fangxiaoyu Feng, Adria Recasens, Guangda Lai, Alberto Magni, Nicola De Cao, Aditya Siddhant, Zoe Ashwood, Jordi Orbay, Mostafa Dehghani, Jenny Brennan, Yifan He, Kelvin Xu, Yang Gao, Carl Saroufim, James Molloy, Xinyi Wu, Seb Arnold, Solomon Chang, Julian Schrittwieser, Elena Buchatskaya, Soroush Radpour, Martin Polacek, Skye Giordano, Ankur Bapna, Simon Tokumine, Vincent Hellendoorn, Thibault Sottiaux, Sarah Cogan, Aliaksei Severyn, Mohammad Saleh, Shantanu Thakoor, Laurent Shefey, Siyuan Qiao, Meenu Gaba, Shuo yiin Chang, Craig Swanson, Biao Zhang, Benjamin Lee, Paul Kishan Rubenstein, Gan Song, Tom Kwiatkowski, Anna Koop, Ajay Kannan, David Kao, Parker Schuh, Axel Stjerngren, Golnaz Ghiasi, Gena Gibson, Luke Vilnis, Ye Yuan, Felipe Tiengo Ferreira, Aishwarya Kamath, Ted Klimenko, Ken Franko, Kefan Xiao, Indro Bhattacharya, Miteyan Patel, Rui Wang, Alex Morris, Robin Strudel, Vivek Sharma, Peter Choy, Sayed Hadi Hashemi, Jessica Landon, Mara Finkelstein, Priya Jhakra, Justin Frye, Megan Barnes, Matthew Mauger, Dennis Daun, Khuslen Baatarsukh, Matthew Tung, Wael Farhan, Henryk Michalewski, Fabio Viola, Felix de Chaumont Quitry, Charline Le Lan, Tom Hudson, Qingze Wang, Felix Fischer, Ivy Zheng, Elspeth White, Anca Dragan, Jean baptiste Alayrac, Eric Ni, Alexander Pritzel, Adam Iwanicki, Michael Isard, Anna Bulanova, Lukas Zilka, Ethan Dyer, Devendra Sachan, Srivatsan Srinivasan, Hannah

Muckenhirn, Honglong Cai, Amol Mandhane, Mukarram Tariq, Jack W. Rae, Gary Wang, Kareem Ayoub, Nicholas FitzGerald, Yao Zhao, Woohyun Han, Chris Alberti, Dan Garrette, Kashyap Krishnakumar, Mai Gimenez, Anselm Levskaya, Daniel Sohn, Josip Matak, Inaki Iturrate, Michael B. Chang, Jackie Xiang, Yuan Cao, Nishant Ranka, Geoff Brown, Adrian Hutter, Vahab Mirrokni, Nanxin Chen, Kaisheng Yao, Zoltan Egyed, Francois Galilee, Tyler Liechty, Praveen Kallakuri, Evan Palmer, Sanjay Ghemawat, Jasmine Liu, David Tao, Chloe Thornton, Tim Green, Mimi Jasarevic, Sharon Lin, Victor Cotruta, Yi-Xuan Tan, Noah Fiedel, Hongkun Yu, Ed Chi, Alexander Neitz, Jens Heitkaemper, Anu Sinha, Denny Zhou, Yi Sun, Charbel Kaed, Brice Hulse, Swaroop Mishra, Maria Georgaki, Sneha Kudugunta, Clement Farabet, Izhak Shafran, Daniel Vlasic, Anton Tsitsulin, Rajagopal Ananthanarayanan, Alen Carin, Guolong Su, Pei Sun, Shashank V, Gabriel Carvajal, Josef Broder, Iulia Comsa, Alena Repina, William Wong, Warren Weilun Chen, Peter Hawkins, Egor Filonov, Lucia Loher, Christoph Hirnschall, Weiyi Wang, Jingchen Ye, Andrea Burns, Hardie Cate, Diana Gage Wright, Federico Piccinini, Lei Zhang, Chu-Cheng Lin, Ionel Gog, Yana Kulizhskaya, Ashwin Sreevatsa, Shuang Song, Luis C. Cobo, Anand Iyer, Chetan Tekur, Guillermo Garrido, Zhuyun Xiao, Rupert Kemp, Huaixiu Steven Zheng, Hui Li, Ananth Agarwal, Christel Ngani, Kati Goshvadi, Rebeca Santamaria-Fernandez, Wojciech Fica, Xinyun Chen, Chris Gorgolewski, Sean Sun, Roopal Garg, Xinyu Ye, S. M. Ali Eslami, Nan Hua, Jon Simon, Pratik Joshi, Yelin Kim, Ian Tenney, Sahitya Potluri, Lam Nguyen Thiet, Quan Yuan, Florian Luisier, Alexandra Chronopoulou, Salvatore Scellato, Praveen Srinivasan, Minmin Chen, Vinod Koverkathu, Valentin Dalibard, Yaming Xu, Brennan Saeta, Keith Anderson, Thibault Sellam, Nick Fernando, Fantine Huot, Junehyuk Jung, Mani Varadarajan, Michael Quinn, Amit Raul, Maigo Le, Ruslan Habalov, Jon Clark, Komal Jalan, Kalesha Bullard, Achintya Singhal, Thang Luong, Boyu Wang, Sujeevan Rajayogam, Julian Eisenschlos, Johnson Jia, Daniel Finchelstein, Alex Yakubovich, Daniel Balle, Michael Fink, Sameer Agarwal, Jing Li, Dj Dvijotham, Shalini Pal, Kai Kang, Jaclyn Konzelmann, Jennifer Beattie, Olivier Dousse, Diane Wu, Remi Crocker, Chen Elkind, Siddhartha Reddy Jonnalagadda, Jong Lee, Dan Holtmann-Rice, Krystal Kallarackal, Rosanne Liu, Denis Vnukov, Neera Vats, Luca Invernizzi, Mohsen Jafari, Huanjie Zhou, Lilly Taylor, Jennifer Prendki, Marcus Wu, Tom Eccles, Tianqi Liu, Kavya Kopparapu, Francoise Beaufays, Christof Angermueller, Andreea Marzoca, Shourya Sarcar, Hilal Dib, Jeff Stanway, Frank Perbet, Nejc Trdin, Rachel Sterneck, Andrey Khorlin, Dinghua Li, Xihui Wu, Sonam Goenka, David Madras, Sasha Goldshtein, Willi Gierke, Tong Zhou, Yaxin Liu, Yannie Liang, Anais White, Yunjie Li, Shreya Singh, Sanaz Bahargam, Mark Epstein, Sujoy Basu, Li Lao, Adnan Ozturel, Carl Crous, Alex Zhai, Han Lu, Zora Tung, Neeraj Gaur, Alanna Walton, Lucas Dixon, Ming Zhang, Amir Globerson, Grant Uy, Andrew Bolt, Olivia Wiles, Milad Nasr, Ilia Shumailov, Marco Selvi, Francesco Piccinno, Ricardo Aguilar, Sara McCarthy, Misha Khalman, Mrinal Shukla, Vlado Galic, John Carpenter, Kevin Villela, Haibin Zhang, Harry Richardson, James Martens, Matko Bosnjak, Shreyas Rammohan Belle, Jeff Seibert, Mahmoud Alnahlawi, Brian McWilliams, Sankalp Singh, Annie Louis, Wen Ding, Dan Popovici, Lenin Simicich, Laura Knight, Pulkit Mehta, Nishesh Gupta, Chongyang Shi, Saaber Fatehi,

Jovana Mitrovic, Alex Grills, Joseph Pagadora, Dessie Petrova, Danielle Eisenbud, Zhishuai Zhang, Damion Yates, Bhavishya Mittal, Nilesh Tripuraneni, Yannis Assael, Thomas Brovelli, Prateek Jain, Mihajlo Velimirovic, Canfer Akbulut, Jiaqi Mu, Wolfgang Macherey, Ravin Kumar, Jun Xu, Haroon Qureshi, Gheorghe Comanici, Jeremy Wiesner, Zhitao Gong, Anton Ruddock, Matthias Bauer, Nick Felt, Anirudh GP, Anurag Arnab, Dustin Zelle, Jonas Rothfuss, Bill Rosgen, Ashish Shenoy, Bryan Seybold, Xinjian Li, Jayaram Mudigonda, Goker Erdogan, Jiawei Xia, Jiri Simsa, Andrea Michi, Yi Yao, Christopher Yew, Steven Kan, Isaac Caswell, Carey Radebaugh, Andre Elisseeff, Pedro Valenzuela, Kay McKinney, Kim Paterson, Albert Cui, Eri Latorre-Chimoto, Solomon Kim, William Zeng, Ken Durden, Priya Ponnapalli, Tiberiu Sosea, Christopher A. Choquette-Choo, James Manyika, Brona Robenek, Harsha Vashisht, Sebastien Pereira, Hoi Lam, Marko Velic, Denese Owusu-Afriyie, Katherine Lee, Tolga Bolukbasi, Alicia Parrish, Shawn Lu, Jane Park, Balaji Venkatraman, Alice Talbert, Lambert Rosique, Yuchung Cheng, Andrei Sozanschi, Adam Paszke, Praveen Kumar, Jessica Austin, Lu Li, Khalid Salama, Wooyeol Kim, Nandita Dukkipati, Anthony Baryshnikov, Christos Kaplanis, XiangHai Sheng, Yuri Chervonyi, Caglar Unlu, Diego de Las Casas, Harry Askham, Kathryn Tunyasuvunakool, Felix Gimeno, Siim Poder, Chester Kwak, Matt Miecnikowski, Vahab Mirrokni, Alek Dimitriev, Aaron Parisi, Dangyi Liu, Tomy Tsai, Toby Shevlane, Christina Kouridi, Drew Garmon, Adrian Goedeckemeyer, Adam R. Brown, Anitha Vijayakumar, Ali Elqursh, Sadegh Jazayeri, Jin Huang, Sara Mc Carthy, Jay Hoover, Lucy Kim, Sandeep Kumar, Wei Chen, Courtney Biles, Garrett Bingham, Evan Rosen, Lisa Wang, Qijun Tan, David Engel, Francesco Pongetti, Dario de Cesare, Dongseong Hwang, Lily Yu, Jennifer Pullman, Srini Narayanan, Kyle Levin, Siddharth Gopal, Megan Li, Asaf Aharoni, Trieu Trinh, Jessica Lo, Norman Casagrande, Roopali Vij, Loic Matthey, Bramandia Ramadhana, Austin Matthews, CJ Carey, Matthew Johnson, Kremena Goranova, Rohin Shah, Shereen Ashraf, Kingshuk Dasgupta, Rasmus Larsen, Yicheng Wang, Manish Reddy Vuyyuru, Chong Jiang, Joana Ijazi, Kazuki Osawa, Celine Smith, Ramya Sree Boppana, Taylan Bilal, Yuma Koizumi, Ying Xu, Yasemin Altun, Nir Shabat, Ben Bariach, Alex Korchemniy, Kiam Choo, Olaf Ronneberger, Chimezie Iwuanyanwu, Shubin Zhao, David Soergel, Cho-Jui Hsieh, Irene Cai, Shariq Iqbal, Martin Sundermeyer, Zhe Chen, Elie Bursztein, Chaitanya Malaviya, Fadi Biadsy, Prakash Shroff, Inderjit Dhillon, Tejasi Latkar, Chris Dyer, Hannah Forbes, Massimo Nicosia, Vitaly Nikolaev, Somer Greene, Marin Georgiev, Pidong Wang, Nina Martin, Hanie Sedghi, John Zhang, Praseem Banzal, Doug Fritz, Vikram Rao, Xuezhi Wang, Jiageng Zhang, Viorica Patraucean, Dayou Du, Igor Mordatch, Ivan Jurin, Lewis Liu, Ayush Dubey, Abhi Mohan, Janek Nowakowski, Vlad-Doru Ion, Nan Wei, Reiko Tojo, Maria Abi Raad, Drew A. Hudson, Vaishakh Keshava, Shubham Agrawal, Kevin Ramirez, Zhichun Wu, Hoang Nguyen, Ji Liu, Madhavi Sewak, Bryce Petrini, DongHyun Choi, Ivan Philips, Ziyue Wang, Ioana Bica, Ankush Garg, Jarek Wilkiewicz, Priyanka Agrawal, Xiaowei Li, Danhao Guo, Emily Xue, Naseer Shaik, Andrew Leach, Sadh MNM Khan, Julia Wiesinger, Sammy Jerome, Abhishek Chakladar, Alek Wenjiao Wang, Tina Ornduff, Folake Abu, Alireza Ghaffarkhah, Marcus Wainwright, Mario Cortes, Frederick Liu, Joshua Maynez, Andreas Terzis, Pouya Samangouei, Riham Mansour, Tomasz Kępa,

François-Xavier Aubet, Anton Algymr, Dan Banica, Agoston Weisz, Andras Orban, Alexandre Senges, Ewa Andrejczuk, Mark Geller, Niccolo Dal Santo, Valentin Anklin, Majd Al Merey, Martin Baeuml, Trevor Strohman, Junwen Bai, Slav Petrov, Yonghui Wu, Demis Hassabis, Koray Kavukcuoglu, Jeffrey Dean, and Oriol Vinyals. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context, 2024.

[196] Shelby Thomas, Chetan Gohkale, Enrico Tanuwidjaja, Tony Chong, David Lau, Saturnino Garcia, and Michael Bedford Taylor. CortexSuite: A Synthetic Brain Benchmark Suite. In *International Symposium on Workload Characterization (IISWC)*, Oct. 2014.

[197] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open Foundation and Fine-Tuned Chat Models. *arXiv:2307.09288 [cs]*, 2023.

[198] Walker J. Turner, John W. Poulton, John M. Wilson, Xi Chen, Stephen G. Tell, Matthew Fojtik, Thomas H. Greer, Brian Zimmer, Sanquan Song, Nikola Nedovic, Sudhir S. Kudva, Sunil R. Sudhakaran, Rizwan Bashirullah, Wenxu Zhao, William J. Dally, and C. Thomas Gray. Ground-referenced Signaling for Intra-chip and Short-reach Chip-to-chip Interconnects. In *Custom Integrated Circuits Conference (CICC)*, 2018.

[199] Hado van Hasselt, Matteo Hessel, and John Aslanides. When to Use Parametric Models in Reinforcement Learning? *arXiv preprint arXiv:1906.05243*, June 2019.

[200] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. *arXiv:1706.03762 [cs]*, 2017.

[201] Luis Vega and Michael Bedford Taylor. RV-IOV: Tethering RISC-V Processors via Scalable I/O Virtualization . In *CARRV*, 2017.

[202] Bandhav Veluri, Collin Pernu, Ali Saffari, Joshua Smith, Michael Taylor, and Shyam Gollakota. Neuricam: Low-power video acquisition using dual-mode iot cameras. In *MobiCom*, 2023.

[203] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: reducing the energy of mature computations. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[204] Ganesh Venkatesh, John Sampson, Nathan Goulding, Sravanthi Kota Venkata, Michael Bedford Taylor, and Steven Swanson. QsCores: Configurable Co-processors to Trade Dark Silicon for Energy Efficiency in a Scalable Manner. In *International Symposium on Microarchitecture (MICRO)*, 2011.

[205] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring it all to Software: Raw Machines. In *IEEE Computer*, September 1997.

[206] Ying Wang, Mengdi Wang, Bing Li, Huawei Li, and Xiaowei Li. A Many-Core Accelerator Design for On-Chip Deep Reinforcement Learning. In *Proceedings of the 39th International Conference on Computer-Aided Design*, pages 1–7, Virtual Event USA, November 2020. ACM.

[207] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 2009.

[208] Wm A Wulf and Sally A McKee. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

[209] John Wuu, Rahul Agarwal, Michael Ciraula, Carl Dietz, Brett Johnson, Dave Johnson, Russell Schreiber, Raja Swaminathan, Will Walker, and Samuel Naffziger. 3D V-Cache: the Implementation of a Hybrid-Bonded 64MB Stacked Cache for a 7nm x86-64 CPU. In *IEEE International Solid-State Circuits Conference (ISSCC)*, 2022.

[210] Chenhao Xie, Xie Li, Yang Hu, Huwan Peng, Michael Taylor, and Shuaiwen Leon Song. Q-VR: System-level design for future mobile collaborative virtual reality. In *ASPLOS*, 2021.

[211] Shaolin Xie, Scott Davidson, Ikuo Magaki, Moein Khazraee, Luis Vega, Lu Zhang, and Michael B. Taylor. Extreme datacenter specialization for planet-scale computing: Asic clouds. In *ACM Sigops Operating System Review*, 2018.

[212] Amy Yang, Jingyi Yang, Aya Ibrahim, Xinfeng Xie, Bangsheng Tang, Grigory Sizov, Jeremy Reizenstein, Jongsoo Park, and Jianyu Huang. Context parallelism for scalable million-token inference, 2024.

[213] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Ré, Christopher R. Aberger, and Christopher De Sa. PipeMare: Asynchronous Pipeline Parallel DNN Training. *arXiv:1910.05124 [cs, stat]*, February 2020. arXiv: 1910.05124.

[214] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, Christos Kozyrakis, and Mark Horowitz. Interstellar: Using Halide's Scheduling Language to Analyze DNN Accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 369–383, Lausanne, Switzerland, March 2020.

[215] Chris Ying, Sameer Kumar, Dehao Chen, Tao Wang, and Youlong Cheng. Image Classification at Supercomputer Scale. *arXiv:1811.06992 [cs]*, 2018.

[216] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022.

[217] Zhiheng Yue, Huizheng Wang, Jiahao Fang, Jinyi Deng, Guangyang Lu, Fengbin Tu, Ruiqi Guo, Yuxuan Li, Yubin Qin, Yang Wang, Chao Li, Huiming Han, Shaojun Wei, Yang Hu, and Shouyi Yin. Exploiting similarity opportunities of emerging vision ai models on hybrid bonding architecture. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 396–409, 2024.

[218] Karen Zee, Viktor Kuncak, Michael Taylor, and Martin C. Rinard. Runtime checking for program verification. In *RV*, 2007.

[219] Hengrui Zhang, August Ning, Rohan Baskar Prabhakar, and David Wentzlaff. LLM-Compass: Enabling Efficient Hardware Design for Large Language Model Inference. In *Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024.

[220] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. OPT: Open Pre-trained Transformer Language Models. *arXiv:2205.01068 [cs]*, 2022.

[221] Xingyao Zhang, Haojun Xia, Donglin Zhuang, Hao Sun, Xin Fu, Michael Taylor, and Shuaiwen Leon Song. $\eta$-LSTM: Co-designing highly-efficient large lstm training via exploiting memory-saving and architectural design opportunities. In *ISCA*, 2021.

[222] Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Huazuo Gao, Jiashi Li, Liyue Zhang, Panpan Huang, Shangyan Zhou, Shirong Ma, Wenfeng Liang, Ying He, Yuqing Wang, Yuxuan Liu, and Y. X. Wei. Insights into deepseek-v3: Scaling challenges and reflections on hardware for ai architectures, 2025.

[223] Ritchie Zhao, Chun Zhao, Shaolin Xie, Bandhav Veluri, Luis Vega, Christopher Torng, Ningxiao Sun, Austin Rovinski, Anuj Rao, Gai Liu, Paul Gao, Scott Davidson, Steve Dai, Aporva Amarnath, KhalidAl-Hawaj, Tutu Ajayi Christopher Batten, Ronald G. Dreslinski, Rajesh K.Gupta, Michael B.Taylor, and Zhiru Zhang. Celerity: An Open Source RISC-V Tiered Accelerator Fabric. In *7th RISC-V Workshop*, 2017.

[224] Qiaoshi Zheng, Nathan Goulding-Hotta, Scott Ricketts, Steven Swanson, Michael Bedford Taylor, and Jack Sampson. Exploring energy scalability in coprocessor-dominated architectures for dark silicon. *Transactions on Embedded Computing Systems (TECS)*, Mar 2014.

[225] Yi Zhu, Yuanfang Hu, Michael Taylor, and Chung-Kuan Cheng. Energy and switch area optimizations for FPGA global routing architectures. In *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, January 2009.

[226] Yi Zhu, Michael Taylor, Scott B. Baden, and Chung-Kuan Cheng. Advancing supercomputer performance through interconnection topology synthesis. In *International Conference on Computer-Aided Design (ICCAD)*, pages 555–558, 2008.