

# **Instruction-Level Performance Analysis and Optimization**

## **Strategies for Constrained AI Accelerators: A Case Study of CMP**

### **170HX**

Xing Kangwei

xkw1998@hotmail.com

#### **Abstract**

Limited GPUs are widely present in commercial product lines, where a significant gap often exists between their advertised computational power and actual available performance, rendering traditional peak FLOPs-based performance evaluation methods ineffective. Building upon early community observations that “disabling FP32 FMA instructions can yield performance gains on the CMP 170HX,” this paper extends the hypothesis of the restriction mechanism from a single instruction to selective performance suppression coupled across three dimensions: instruction types, numerical precision, and execution units.

To systematically characterize the computational behavior of such limited GPUs, this paper proposes an instruction-level performance analysis framework. This includes an instruction throughput and SM normalization model, an effective precision execution factor, and an instruction-level energy density metric to quantify the actual execution efficiency under different instruction and precision combinations. Based on this framework, a systematic instruction-level experimental analysis is conducted on the CMP 170HX (a restricted accelerator based on the GA100 architecture), with comparisons made against unrestricted GPUs of the same generation.

Furthermore, this paper summarizes and proposes four categories of operator-level performance circumvention strategies: instruction splitting, intrinsic instruction substitution, algebraic transformation, and execution unit migration, aiming to bypass restricted instruction paths within the legal programming model. Guided by these strategies, a custom operator extension based on PyTorch is implemented and validated across three typical AI inference tasks—Transformer language model inference, Stable Diffusion image generation, and Diffusion Transformer (DiT, e.g., Z-Image-Turbo). Experimental results demonstrate that on the CMP 170HX, the proposed methods can achieve up to approximately 2 $\times$  end-to-end inference performance improvement

without significantly compromising numerical accuracy.

This study provides a reproducible instruction-level analysis framework for performance evaluation and engineering utilization of restricted GPUs, offering practical references for the reuse and energy efficiency optimization of low-cost computational resources.

**Keywords:** Restricted GPU, Instruction-level Performance Analysis, GPU Microbenchmarking, Energy-aware Modeling, AI Inference Acceleration, Diffusion Models, CMP 170HX

## Table of Contents

Abstract .....	1
Table of Contents .....	3
Figures.....	7
Tables .....	7
1 Introduction.....	8
1.1 Research Background: Commercially Constrained Accelerators .....	8
1.2 Case Motivation: The CMP Series as a Representative of Restricted Accelerators.....	9
1.3 Review of Preliminary Research.....	10
1.4 Research Questions and Contributions of This Paper .....	11
2 Related Work .....	13
2.1 GPU Performance Limitation Techniques .....	13
2.1.1 Hardware Limitation (Hardware Binning/Fusing) .....	13
2.1.2 Software/Firmware/Driver Limitation (Software/Firmware/Driver Gating) .....	14
2.2 Instruction-Level Performance Optimization .....	14
2.2.1 Optimization of Typical Computational Instructions: Fused Multiply-Add and Mixed-Precision Computing .....	15
2.2.2 Optimization of Typical Memory Access Instructions: Kernel Fusion.....	16
2.2.3 Optimization of Special Function Instructions .....	16
2.3 Constrained Hardware and Low-Cost Computing Practices .....	17
3 Abstract Model and Quantitative Metrics for Constrained GPU Performance Analysis .....	22
3.1 Motivation.....	22
3.2 Model Assumptions and Applicability Boundaries .....	23
3.3 Instruction Throughput and SM Normalization Model .....	24
3.4 Precision Ratio and the Effective Precision Model .....	25
3.5 Instruction-Level Energy Density .....	27
3.6 Derivation from Models to Experimental Design .....	28
3.6.1 Independent Control Strategy Based on Saturation Throughput.....	29
3.6.2 Cross-Validation Strategy for Theoretical and Measured Precision .....	29
3.6.3 Dynamic Monitoring Strategy for Power-Performance Coupling.....	30
3.6.4 Overall Mapping of Experimental Design .....	30
4 Research Hypotheses and Scope.....	31
4.1 Instruction-Level Performance Limitation Hypothesis .....	31
4.2 Hypothesis on Precision-Execution Unit Correlation .....	31
4.3 Hypothesis on Power-Performance Side-Channel Analysis .....	32

4.4 Research Scope.....	32
5 Experimental Design and Methodology .....	34
5.1 Selection of Typical Inference Operators and Instruction Mapping.....	34
5.2 Categorization of CUDA Instruction Types .....	37
5.3 Rationale for Experimental Subject Selection .....	38
5.4 Instruction-Level Testing Methodology .....	39
5.5 Automation Testing Tools and Workflow .....	40
6 Experimental Results.....	42
6.1 Overall Characteristics.....	42
6.2 FP32 Instruction Throughput Characteristics .....	43
6.3 FP16 Instruction Throughput Characteristics .....	47
6.4 BF16 Instruction Throughput Characteristics .....	50
7 Analysis and Discussion .....	53
7.1 Analysis of the Multi-Level Restriction Mechanism in Restricted AI Accelerators .....	53
7.2 Empirical Refinement and Applicability Assessment of the Theoretical Model .....	54
7.3 Interpretation of the Motivations Behind Possible Limiting Mechanisms .....	55
7.4 Inference on the State of Tensor Core .....	56
7.5 Discussion on the Generalization of the Proposed Model .....	56
8 Operator-level Circumvention Strategies and Engineering Practice .....	58
8.1 Overview of Circumvention Strategies.....	58
8.1.1 Instruction Decomposition Strategy.....	58
8.1.2 Intrinsic Instruction Substitution Strategy.....	58
8.1.3 Algebraic Transformation Strategy .....	58
8.1.4 Execution Unit Migration Strategy .....	59
8.2 Implementation of Custom Operators .....	59
8.2.1 FMA Deconstruction of the Linear Transformation Operator (Linear/GEMM) ..	59
8.2.2 Intrinsic Instruction Optimization for the SiLU Activation Function.....	60
8.2.3 Algebraic Reconstruction and Reciprocal Square Root Approximation for RMSNorm .....	60
8.3 Operator-Level Theoretical Performance Evaluation.....	61
8.4 Performance in Operator Engineering Applications.....	64
8.4.1 Stable Diffusion Image Generation Inference .....	65
8.4.2 Large Language Model Inference .....	65
8.4.3 DiT Image Diffusion Model Inference.....	66
9 Deficiencies and Limitations of the Research .....	67
9.1 Partial Instructions Subject to Underlying Hardware Constraints, Difficult to Fully	

Overcome via Software Means .....	67
9.2 High Engineering Cost of Manual Optimization for Complex Operators (e.g., Attention Mechanism).....	68
9.3 Kernel Launch Overhead Introduced by Fine-grained Operator Replacement .....	68
9.4 Practical Challenges in Integrating Patches into Large-Scale Deep Learning Frameworks .....	69
9.5 Lack of In-Depth Instruction Behavior Analysis at the PTX/SASS Level.....	70
10 Conclusion .....	72
10.1 Core Finding: Constraints are Instruction-Level and Hierarchical .....	72
10.2 Practical Guidelines for CUDA Programming on the 170HX .....	72
10.3 Power Side-Channel Analysis: A New Paradigm for Inferring Hardware Behavior in Constrained Environments.....	73
10.4 Environmental Value and Economic Significance: Promoting the Sustainable Reuse of Low-Cost Computing Power .....	74
Acknowledgments.....	75
References .....	77
Appendix .....	79
Appendix 1 Experimental Environment Configuration and System Parameters.....	79
1.1 Local Environment Evaluation Platform .....	79
1.2 Cloud Baseline Environment.....	79
1.3 Compilation Environment Configuration and Optimization Parameters.....	80
Appendix 2 Analysis of Temperature Drift Characteristics of Static Power Consumption in GA100 Architecture .....	80
2.1 Experimental Observation Data.....	80
2.2 Explanation of Physical Mechanism.....	81
2.3 Impact on Energy Efficiency Modeling .....	81
Appendix 3 Multi-Platform Operator-Level Performance Comparative Analysis .....	82
Appendix 4 Numerical Precision Preservation Verification of Custom Operators.....	83
4.1 Objective Quality Assessment for Image Generation Tasks .....	83
4.2 Perplexity Analysis for Language Model Inference .....	84
Appendix 5 Analysis of Limitations in Custom Operator Design and Attribution of Performance Differences .....	85
5.1 Insufficient Operator-Level Optimization Depth and Resource Allocation Trade-offs .....	85
5.2 Cache Locality Failure Due to Micro-Architectural Differences .....	86
Appendix 6 Design Principles and Implementation Details of Instruction-Level Micro-	

Benchmarking Programs .....	87
6.1 High Parallelism Architecture Design.....	87
6.2 Steady-State Long-Duration Operation Mechanism.....	89
6.3 Compiler Optimization Prevention Mechanisms.....	90
Appendix 7 Analysis of Discrepancies Between Officially Advertised Computing Power Metrics and Actually Achievable Performance.....	91
7.1 Discrepancies in Numerical Precision Representation .....	92
7.2 Performance Differences between Dense and Sparse Matrices .....	92
7.3 Empirical Observations of FP16 Performance on GA100 Architecture: The Gap from 4x to 2x in Practice .....	92
7.4 Side-channel Analysis of BF16 Performance Limitation in RTX 3080 (GA102)...	93

## Figures

**Figure 5-1:** Proportion of Each Operator in the Transformer Inference Stage

**Figure 5-2:** Proportion of Each Operator in the Stable Diffusion Inference Stage

**Figure 6-1:** Instruction Execution Factor - Precision Graph for 170HX

**Figure 6-2:** FP32 Arithmetic Floating-Point Performance Comparison: 170HX-A800-RTX3080

**Figure 6-3:** FP32 Mathematical Floating-Point Instruction Performance Comparison: 170HX-A800-RTX3080

**Figure 6-4:** FP32 Instruction Energy Density Comparison: 170HX-A800-RTX3080

**Figure 6-5:** FP16 Arithmetic Floating-Point Performance Comparison: 170HX-A800-RTX3080

**Figure 6-6:** FP16 Arithmetic Instruction Energy Density Comparison: 170HX-A800-RTX3080

**Figure 6-7:** FP16 Computational Energy Density Comparison: 170HX-A800-RTX3080

**Figure 6-8:** BF16 Arithmetic Floating-Point Performance Comparison: 170HX-A800-RTX3080

**Figure 6-9:** BF16 Computational Energy Density Comparison: 170HX-A800-RTX3080

**Figure 8-1:** Custom Operator vs. Native Operator Throughput Comparison with Applied 170HX Optimization Strategy

**Figure 8-2:** Custom Operator vs. Native Operator Power Consumption Comparison with Applied 170HX Optimization Strategy

**Figure 8-3:** Speedup Ratio of Custom Operator with 170HX Optimization Strategy vs. Native Operator Across AI Inference Application Stages

**Figure EX-1:** Custom Operator (170HX) vs. Native Operator (A800/RTX3080) Throughput Comparison

## Tables

**Table 5-1:** Classification of CUDA Programming Instruction Types

**Table EX-1:** GA100 Temperature-Power Table at Idle State

**Table EX-2:** Result Error Table for Custom Operator in Stable Diffusion Inference

**Table EX-3:** Result Error Table for Custom Operator in LLM Inference

# 1 Introduction

## 1.1 Research Background: Commercially Constrained Accelerators

Commercial accelerators are specialized hardware devices designed to expedite dedicated computational tasks such as scientific computing and artificial intelligence (particularly neural networks). The current mainstream AI accelerators can be categorized into the following three classes:

a. Neural Processing Units (NPUs):

Represented by the Tensor Processing Unit (TPU) developed and deployed by Google in its data centers, specifically for supporting services like Gemini; Huawei's Ascend series adopts a system-on-chip (SoC) design, integrating the AI Core tailored for neural network computation and the AI CPU for non-matrix computations [1].

b. General-Purpose Graphics Processing Units (GPGPUs):

Exemplified by NVIDIA GPUs, which feature thousands of computational cores and can act as coprocessors to CPUs for executing large-scale parallel computing, thus finding extensive application in AI computing [2][3]. Since the Volta architecture, NVIDIA has introduced Tensor Cores alongside traditional CUDA Cores, specifically designed for matrix multiply-accumulate operations [1].

c. Other Research-based / Specialized Accelerators:

For instance, a high-efficiency reconfigurable accelerator for convolutional neural networks previously proposed by MIT [1].

The large-scale adoption of AI accelerators is inseparable from corporate production and manufacturing. Presently, the primary global supplier of general-purpose GPUs is NVIDIA, whose product lines include the GeForce series for gaming users, the RTX Pro (formerly Quadro) series for workstations, and the NVIDIA (formerly Tesla) series for data centers, AI, and High-Performance Computing (HPC) [8]. Additionally, NVIDIA has previously released the CMP/P series GPUs specifically for cryptocurrency (e.g., Ethereum) mining [4].

However, community research and verification analyses have shown that the actual AI computational power (e.g., floating-point computational performance) of such GPUs is

significantly lower than theoretical values, with specific manifestations as follows:

- a. Traditional stress-testing tools (e.g., GPU-Burn) exhibit abnormally low power consumption on such GPUs;
- b. Standard dense matrix multiplication tests also demonstrate low performance and low power consumption;
- c. In large language model inference tests based on Llama-bench, these mining GPUs similarly show low performance and low power consumption;
- d. Using ETHash tools (e.g., NBMiner) allows measurement of real-time power consumption close to the Thermal Design Power (TDP) [4].

Thus, such GPUs can be regarded as "constrained AI accelerators," meaning their actual performance and power consumption boundaries are difficult to accurately characterize using traditional testing methods. Furthermore, considering the specific phenomena observed thus far, other similar AI accelerators that cannot be accurately described by traditional testing methods may emerge in the future.

## **1.2 Case Motivation: The CMP Series as a Representative of Restricted Accelerators**

Mining-specific GPUs are graphics processors designed for cryptocurrency (e.g., Ethereum) mining algorithms. These GPUs typically have their display functionalities removed to maximize mining efficiency [4]. NVIDIA has introduced several such products, primarily including:

- a. CMP Series: Models such as the CMP 30HX (TU116), 40HX (TU106), 50HX (TU102), 90HX (GA102), and 170HX (GA100) [4];
- b. P10x Series: Such as the P106 (GP106), P104 (GP104), and P102 (GP102) [5].

Although NVIDIA does not market these products as general-purpose GPUs, their underlying architecture is based on GPGPU, theoretically retaining a certain degree of general-purpose computing capability [6].

With Ethereum's transition from Proof of Work (PoW) to Proof of Stake (PoS), these GPUs specifically designed for PoW mining have gradually lost their primary application scenarios, and

their market availability is also relatively limited [4]. Among them, the CMP series is a typical representative of mining-specific GPUs and restricted AI accelerators, especially the CMP 170HX model. This model utilizes the GA100-105F-A1 GPU core, comes standard with 8GB of HBM2e memory, and features 70 Streaming Multiprocessors (SMs) and 4480 CUDA cores. Since it employs the same GA100 core as the high-end data center compute card NVIDIA A100, and HBM2e memory offers significant advantages in AI inference tasks, it is suitable as the research subject for this study. It should be noted that the community has verified that some models (e.g., P106/P104/P102/30HX/40HX) can still be used for graphics computing (e.g., gaming), but they differ fundamentally from AI inference tasks, thus holding only referential significance [4].

### 1.3 Review of Preliminary Research

Through systematic review and experimental validation of fragmented research findings within the community, previous studies have made the following progress:

a. **Performance Improvement of FP32 by Disabling FMA Instructions:** On the NVIDIA CMP 170HX platform, by modifying CUDA source code and compiling with fused multiply-add (FMA) instructions disabled, the FP32 computational performance was successfully increased from approximately 0.39 TFLOPS to about 6.2 TFLOPS, recovering roughly 50% of its theoretical performance. This method shows significant effectiveness in large language model inference tasks, particularly enhancing decoding speed noticeably for quantized models (e.g., Q6, Q4\_K\_M).

b. **Confirmation of CMP 170HX's Energy Efficiency and Cost Advantages in Specific AI Inference Scenarios:** In tasks constrained by memory bandwidth with model sizes not exceeding 8GB (such as the decoding stage of LLMs), the energy efficiency of the CMP 170HX approaches that of the same-architecture A100. Combined with its secondary market price advantage, this card demonstrates potential economic and environmental value in low-cost scenarios like edge computing and lightweight AI inference.

However, significant limitations remain in the preliminary research:

a. **Validation Limited to a Single Instruction Set, Lacking Systematic Analysis:** The research only validated the impact of disabling FMA instructions on FP32 performance, failing to systematically investigate other potentially restricted instruction sets or hardware functions (e.g., Tensor Cores, FP64 units). The specific mechanisms by which NVIDIA implements performance limitations in the CMP series (e.g., whether through drivers, microcode, or hardware fuses) were

also not explored in depth, limiting the potential for further performance recovery.

- b. **No Universal Solution Proposed, Reliance on Specific Compilation Environment:** The performance recovery method heavily depends on manual modification of CUDA source code and compilation parameters, and its effectiveness is inconsistent across different frameworks (e.g., FP16 performance was not improved in PyTorch). The lack of a cross-platform, cross-framework universal optimization solution restricts the practical application scope and usability of the method.
- c. **Insufficient Discussion on Theoretical Performance Recovery Paths:** Although potential recovery paths such as "driver cracking," "open-source drivers," and "custom CUDA programming" were proposed, none were substantively validated or assessed for feasibility, especially lacking effective solutions for FP64 performance recovery. The absence of in-depth analysis into possible hardware-level restriction mechanisms within the GPU makes it difficult to judge the possibility of achieving full performance unlock.

#### 1.4 Research Questions and Contributions of This Paper

Based on the aforementioned research landscape, this paper poses the following key questions:

- a. Does the anomalous computational performance stem from instruction-level selective restrictions within CUDA programming? If so, which specific instructions are involved?
- b. Are the pathways of instruction restriction consistent across different computational precisions (e.g., FP16, FP32, FP64)? Can representative combinations of precision and computational instructions be selected for analysis?
- c. Can legitimate CUDA programming techniques (such as the method of disabling FMA mentioned in prior research), through the refactoring of key operators involved in AI inference, achieve partial recovery of AI inference performance?

The main contributions of this paper are as follows:

- a. **At the methodological and conceptual level:** A unified analytical framework of "Instruction × Precision × Execution Unit × Energy Consumption" for commercially restricted accelerators is proposed. It is demonstrated that performance restrictions can manifest as selective suppression at the instruction level and across different tiers. The effectiveness of power side-channels as a means to infer the operational status of execution units is validated.

**b. At the engineering and practical application level:** The aforementioned analytical model is validated using the CMP 170HX as a case study. Guided by this model, operator refactoring is performed, and the effectiveness and feasibility of this approach are verified on real-world inference tasks (including models such as SDXL, Transformer, and DIT).

## 2 Related Work

### 2.1 GPU Performance Limitation Techniques

NVIDIA's GPU product line primarily covers two major categories: consumer-grade and professional-grade, and additionally includes subsequent special SKUs designed for purposes such as cryptocurrency mining [4][9]. As a commercial enterprise, NVIDIA typically adopts a high-reuse strategy for chip design and production to maximize economic benefits. The company primarily employs a combination of hardware limitations and software/firmware restrictions to differentiate performance levels of GPU chips manufactured from the same wafer, thereby achieving product line segmentation and comprehensive market coverage [4]. The specific methods mainly fall into the following two categories:

#### 2.1.1 Hardware Limitation (*Hardware Binning/Fusing*)

Hardware limitation is the most fundamental tiering method. During the production process, some chips may have defects in functional units, such as certain Streaming Multiprocessors (SMs), caches, or memory controllers failing to operate properly. To address this, NVIDIA permanently disables these defective units at the hardware level through fuses. Taking the NVIDIA GA102 architecture as an example, its full design includes 84 SMs. If some SM units fail testing, they can be permanently disabled via hardware fusing, resulting in derivative models with fewer SMs (e.g., versions with 82 or 80 SMs) [10]. This type of limitation is physical and irreversible, directly determining the upper limit of available computational resources for the chip.

This process is typically completed during the Final Test (FT test) or Binning stage. The packaged chips are placed on high-end test platforms and undergo comprehensive testing under conditions close to actual application, including voltage, frequency, and temperature. The test content covers the functionality and performance of all SMs, cache integrity and speed, memory controller stability and bandwidth, as well as power consumption and frequency limits. Based on the test results, chips are assigned to different grades. For chips with defects, specific voltage pulses (electrical fuses) or laser technology (laser fuses) are applied to permanently cut off the circuit paths of defective units or rewrite the chip's device ID to a lower-tier model. Once fusing is completed, the corresponding units are removed from the system address space at the hardware level, making them inaccessible to drivers and software [11].

Due to the irreversibility of hardware limitations, no feasible unlocking solutions currently exist, and it is almost impossible for such solutions to emerge in the future.

#### *2.1.2 Software/Firmware/Driver Limitation (Software/Firmware/Driver Gating)*

Building on hardware limitations, NVIDIA further utilizes drivers, firmware (e.g., vBIOS), and microcode to achieve more flexible functional partitioning and performance control. This mechanism is often referred to as “feature gating.” For example, even if the hardware supports certain advanced features (such as professional rendering engines, full performance modes of AI tensor cores, or the full bandwidth of third-generation NVLink high-speed interconnects), NVIDIA can selectively enable or restrict these features across different product lines (e.g., consumer-grade GeForce and data center-grade Tesla) through drivers or firmware [12][13].

Additionally, for market and compliance reasons, such as adhering to export control policies for specific regions (e.g., restrictions targeting the Chinese market), NVIDIA introduces performance-limited “special-edition” GPUs (e.g., A800, H20). These models typically restrict key performance metrics (such as interconnect bandwidth and computational capability) by modifying drivers or firmware, without requiring a redesign of the chip hardware [4].

This type of limitation does not involve physical shielding, so theoretically, some professional features can be restored through software means. Most related research currently focuses on this direction.

In summary, NVIDIA employs a layered strategy for SKU differentiation: hardware limitations (e.g., fusing, SM disabling) primarily handle “physical tiering” based on manufacturing yield, determining the chip’s original performance and basic specifications. Software/firmware/driver limitations (feature gating) then implement more refined “functional tiering” and “market tiering” on this foundation, distinguishing product positioning, controlling the enablement and disablement of advanced features, and meeting specific compliance requirements. These two approaches complement each other, enabling NVIDIA to maximize chip utilization and build a comprehensive product portfolio spanning from consumer-grade to data center-grade applications.

## **2.2 Instruction-Level Performance Optimization**

In CUDA programming, instructions can be functionally categorized into the following three types:

- a. **Computational Instructions:** Used to perform arithmetic and logical operations. Related literature often refers to "memory access instructions and computational instructions." Furthermore, the GPU Streaming Multiprocessor (SM) contains an Arithmetic Logic Unit (ALU) that supports operations at various precisions such as Single Precision (SP) and Double Precision (DP).
- b. **Memory Access Instructions:** Used for reading and writing data between different memory hierarchies such as registers, shared memory, and global memory. Existing research generally emphasizes the importance of optimizing memory access.
- c. **Special Function Instructions:** Used to execute complex mathematical functions such as trigonometric and exponential functions. Reference materials indicate that the SM includes a Special Function Unit (SFU) dedicated to the execution of this type of instruction [14].

It should be noted that the specific details of the instruction set (e.g., PTX instructions or device native instructions) vary across NVIDIA GPU architectures. However, the aforementioned functional classification is universally applicable.

#### *2.2.1 Optimization of Typical Computational Instructions: Fused Multiply-Add and Mixed-Precision Computing*

The optimization of Fused Multiply-Add (FMA) and Multiply-Add (MAD) instructions lies in merging a multiplication and an addition into a single instruction, thereby reducing the instruction count, enhancing instruction throughput, and better utilizing the GPU hardware pipeline [15]. The specific advantages are as follows:

- a. **Reduced Instruction Count and Increased Throughput:** By combining independent multiplication (e.g., `FMUL`) and addition (e.g., `FADD`) instructions into a single FMA or MAD instruction, the total number of instructions the GPU needs to issue and execute is reduced, contributing to the optimization goal of "maximizing instruction throughput."
- b. **Leveraging Dedicated Hardware Pipelines:** The MAD instruction (especially for floating-point operations) is typically executed within the GPU's floating-point pipeline. Modern GPU architectures (such as NVIDIA Fermi and subsequent architectures) are equipped with multiple dedicated compute units (e.g., SP, DP, SFU) [14]. Through instruction fusion, the compiler can more effectively schedule instructions to the corresponding hardware units for execution, even achieving instruction-level parallelism between integer and floating-point pipelines [16].

c. **Reduced Latency and Improved Precision:** The execution latency of a single FMA instruction is typically lower than that of a separate sequence of multiply and add instructions. For example, the `fma.rn.f32` instruction is mapped at the hardware level to an `FFMA` instruction, which can have a latency as low as 2 cycles. Furthermore, FMA performs rounding only once on the intermediate result, which can sometimes offer higher computational precision compared to performing separate rounding for multiplication and addition [16].

Mixed-precision computing aims to significantly increase computational throughput, reduce memory footprint, and alleviate bandwidth pressure while maintaining numerical accuracy. A typical approach involves performing computations in a lower precision format (e.g., FP16) while accumulating results in a higher precision format (e.g., FP32) to maintain numerical stability. Modern NVIDIA GPUs (such as the Volta and Ampere architectures) integrate Tensor Cores specifically designed to accelerate mixed-precision matrix operations [17]. CUDA provides APIs like `wmma` (Warp Matrix Multiply Accumulate) to directly invoke Tensor Cores, for example, using the `wmma::load_matrix_sync` and `wmma::mma_sync` functions to perform mixed-precision matrix multiplication and accumulation. In the Ampere architecture (e.g., A100), the TF32 data type was further introduced. It adopts the exponent range of FP32 and the mantissa precision of FP16, enabling applications like AI training to leverage Tensor Core acceleration by default without code modification [13].

### *2.2.2 Optimization of Typical Memory Access Instructions: Kernel Fusion*

The primary goals of kernel fusion are to reduce kernel launch overhead, decrease dependence on global memory bandwidth, and enhance performance by improving computational intensity. This technique merges multiple data-dependent operations, which would originally be executed by independent kernels, into a single kernel. The fundamental concept involves manually crafting or utilizing the compiler to generate custom CUDA kernels, thereby fusing multiple sequential operations—such as matrix multiplication (GEMM), element-wise division, reduction sum, and scaling—into a single kernel function. For instance, a fused kernel can combine operators like matrix multiplication (GEMM), element-wise division (Divide), reduction sum (Sum), and scaling. Consequently, intermediate results can be directly passed via registers or shared memory, avoiding the overhead of writing each operator's output back to global memory and subsequently reading it again [18].

### *2.2.3 Optimization of Special Function Instructions*

Special Function Units (SFUs) are crucial hardware components in GPUs for the efficient execution of complex mathematical operations. Optimizing the use of SFUs and preventing them from becoming a performance bottleneck is a vital step in maximizing the instruction throughput of CUDA programs [15]. Common optimization strategies include:

- a. **Algorithmic Reduction of Usage:** Redesign algorithms to minimize reliance on complex special functions (e.g., trigonometric and exponential functions) or replace them with simpler polynomial approximations.
- b. **Utilizing Fast Approximate Functions:** The CUDA math library (e.g., `libdevice`) provides fast approximate versions of some mathematical functions (e.g., `_sinf`, `_expf`). These functions typically offer lower precision but execute faster, effectively alleviating SFU pressure or avoiding its use altogether.
- c. **Enhancing Instruction-Level Parallelism:** By structuring warp-level code to interleave SFU instructions with ALU instructions and memory access instructions, the latency of SFU instructions can be hidden, thereby increasing overall throughput [16].
- d. **Leveraging New Hardware Features:** In the latest GPU architectures (e.g., Hopper), the new instruction set architecture (ISA) and dedicated compute units (e.g., tensor cores) offer more efficient execution paths for specific computational patterns (e.g., matrix multiplication). These features can offload SFU workload or provide higher-performance alternatives [20].

### 2.3 Constrained Hardware and Low-Cost Computing Practices

This section reviews research aligned with the "Green Computing" and "Clean Computing" initiatives. Existing literature indicates that the entire lifecycle of hardware, particularly AI computing hardware like GPUs—from raw material extraction to final disposal—is accompanied by significant resource consumption and environmental burden. Quantitative analyses show that AI hardware is rich in heavy metals and toxic elements such as copper, iron, tin, silicon, and nickel. Its manufacturing process involves intensive mining of these materials, while disposal generates toxic electronic waste. For instance, training large AI models (e.g., GPT-4) typically consumes thousands of GPUs, implying the extraction and subsequent processing costs of tons of toxic elements [21].

In response to the above issues, academia has proposed mitigation pathways centered on

improving resource efficiency. Studies indicate that by increasing hardware utilization and extending equipment service life, the demand for new hardware and corresponding material consumption can be significantly reduced. For example, increasing hardware utilization from 20% to 60% can reduce GPU demand by approximately 67%; similarly, extending hardware lifespan from 1 to 3 years can yield resource savings of a comparable magnitude [21]. This fully illustrates the crucial role of hardware-software co-optimization for achieving sustainable AI development.

However, under high-intensity workloads, the expected lifespan of datacenter-grade GPUs (e.g., NVIDIA A100) is typically only 1 to 3 years, rarely exceeding 3 years. High utilization, while maximizing Return on Investment (ROI), also accelerates hardware iteration and obsolescence, leading to a significant influx of second-hand or decommissioned hardware into the market [21]. This reality provides a solid foundation for secondary market circulation and reuse of GPUs (e.g., downgrading decommissioned hardware for application in scenarios with lower computing demands).

In the realm of community edge computing, the core challenge lies in delivering services with low latency, low cost, and high privacy protection in resource-constrained environments. The traditional cloud deployment model often faces risks of high latency, high bandwidth costs, and data privacy breaches. While edge devices can effectively mitigate these issues due to their proximity to data sources, they are limited by their own computing and storage resources, making it difficult to independently support complex applications like large models [22]. To meet demands while controlling costs, existing research primarily explores the following two types of hardware solutions:

- a. **Energy-Efficient Dedicated Computing Devices:** This category of hardware, distinguished by its high energy efficiency, is particularly suitable for regions lacking access to datacenter-grade GPUs [2]. Their design prioritizes cost-effectiveness, service latency, and security, making them applicable to community edge nodes that do not require top-tier computing power but need to run small-scale Large Language Model (LLM) inference or high-memory-bandwidth applications [4].
- b. **Collaborative Computing Frameworks and Heterogeneous Hardware Integration:** Given the computational ceiling of a single edge device, it is necessary to rely on software frameworks to integrate various heterogeneous, low-cost hardware resources within a community. For example, the Collaborative Edge Computing (CEC) paradigm advocates aggregating geographically distributed edge devices (e.g., Wi-Fi access points, roadside units, 5G base stations) with cloud server resources to collaboratively complete computing tasks.

This implies that communities, through intelligent task scheduling algorithms (such as the EdgeShard framework), can fully leverage existing devices of different performance levels (including the aforementioned low-cost GPUs or domestic AI accelerator chips) to achieve load sharing and cost sharing [21].

In summary, the demands of community edge computing focus on near-site, low-cost, secure service delivery. Its low-cost hardware solutions do not rely on a single high-performance device but rather tend to employ energy-efficient dedicated computing cards and, more critically, depend on software frameworks capable of integrating various scattered, heterogeneous computing resources within the community. This path aligns with the technological trends of Computing Power Network (CPN) and Decentralized Physical Infrastructure Networks (DePIN), which involve utilizing distributed resources to collaboratively complete complex computing tasks [4][21].

Furthermore, engineering practices that use software techniques to compensate for hardware resource limitations or architectural differences are evident in multiple domains. The following research demonstrates coping strategies across various software layers, from algorithm optimization, runtime scheduling, driver-layer adaptation, to development tool enhancement:

- a. **General Software Optimization for Specific Hardware Characteristics:** In research on optimizing small matrix multiplication based on AMD GPU Matrix Cores, although the  $4 \times 4 \times 4$  Matrix Core is unique to AMD, the proposed tiling strategy and memory access optimization methods are general and portable to other hardware platforms. This demonstrates compensating for performance limitations caused by unavailable dedicated units in specific hardware through designing portable software optimization methods [23].
- b. **Software Stack Adaptation for New Hardware Architectures:** Addressing differences in architecture and software stack between Huawei Ascend Neural Network Accelerators and GPUs, researchers proposed strategies such as appropriately setting data tiling and employing low-precision operator implementations to achieve high performance on the Ascend platform. This indicates that software-layer adaptation and optimization can effectively compensate for limitations in the original software ecosystem or programming model when migrating from mature platforms (e.g., GPUs) to new hardware platforms [1].
- c. **Resource Constraint Breakthrough Based on Thread Scheduling:** In the field of GPU performance optimization, various software or hardware-software co-scheduling strategies aim

- to improve hardware resource utilization:
- a. **Thread Reorganization Mechanisms:** Methods like HWS (Hybrid Warp Size) reorganize threads to match the actual SIMD physical channel width, thereby enhancing the effective utilization of computing resources [9].
  - b. **Memory-Aware Scheduling and Cache Bypassing:** Mechanisms like MATB (Memory-aware TLP Throttling and Cache Bypassing) dynamically adjust Thread-Level Parallelism (TLP) and employ cache bypassing strategies during intense cache contention to alleviate access conflicts and improve on-chip computing and network resource utilization [9].
  - c. **Thread Block Compression and Reorganization:** Methods like TSTBC compress and reorganize thread blocks under specific conditions to reduce synchronization wait overhead and effectively hide long-latency operations [9].
- d. **Circumventing or Refactoring Constrained Driver Layers:** Addressing performance limitations at the driver layer of certain GPUs (e.g., CMP 170HX), such as PyTorch's inability to utilize its FP16 performance, research explores theoretical software solutions, such as adopting a no-FMA CUDA compilation scheme, attempting to bypass the generality limitations of proprietary drivers [4].
- e. **AI-Assisted Hardware-Aware Code Generation:** In research on automatically generating high-performance computing kernels, addressing the shortcomings of Large Language Models (LLMs) in generating hardware-efficient code, scholars attempt to provide contextual examples containing detailed hardware specifications and best practices (e.g., operator fusion, data tiling) to guide models in generating better code, thereby compensating for the models' deficiencies in low-level hardware optimization knowledge [18].

## 2.4 Limitations of Existing Research and the Entry Point of This Paper

Integrating existing research on GPU architecture analysis, performance modeling, and AI acceleration reveals that current work, in terms of research subjects, analytical methods, and observability assumptions, inherently presupposes that "the hardware is in a fully operational state." This premise does not always hold in constrained GPU scenarios.

First, in terms of research subjects, existing studies predominantly focus on commercial or datacenter-grade GPUs. Their performance analysis and modeling typically assume that hardware execution units are fully available, and that instruction scheduling strategies are transparent and stable. In contrast, constrained devices such as GPUs dedicated to cryptocurrency mining may feature selective instruction suppression, precision path limitations, or execution unit gating at the

microarchitectural level. However, these characteristics have not been explicitly modeled or defined in existing research.

Second, methodologically, a large body of research relies on application-level benchmarking or throughput tests for single instructions to evaluate GPU performance. While such methods offer good representativeness on full-featured GPUs, in constrained GPU scenarios, application-level performance is often jointly influenced by instruction selection, numerical precision, execution unit mapping, and framework scheduling policies. This makes it difficult to deduce the specific restriction mechanisms. Conversely, single-instruction tests are insufficient to characterize the coupling effects between instruction combinations and precision paths.

Finally, regarding tools and observability, existing performance analysis tools and profiling methods typically depend on performance counters or architecture documentation provided by vendors. Their observable scope may itself be affected by restriction policies, rendering certain execution unit states or energy efficiency characteristics unobservable directly. This consequently weakens the characterization of the actual hardware capabilities of constrained GPUs.

Based on the aforementioned limitations, this paper chooses to start from instruction-level behavior. It introduces an analytical perspective combining power side-channels with numerical precision–execution unit coupling. Within the legal CUDA programming model, a reproducible instruction-level testing methodology is constructed to indirectly infer the execution characteristics and restriction strategies of constrained GPUs.

### **3 Abstract Model and Quantitative Metrics for Constrained GPU Performance Analysis**

#### **3.1 Motivation**

During experimental testing on constrained Graphics Processing Units (GPUs) such as the CMP 170HX, we observed a significant number of performance anomalies that cannot be effectively explained by traditional peak computational metrics (e.g., FLOPs, TOPS). Specifically, under the same numerical precision (FP32), the actual achievable throughput of different arithmetic instructions (FMA vs. AND) exhibits a substantial discrepancy, far exceeding the magnitude theoretically expected (FMA: 0.39 TFlops, AND: 6.23 TFlops) [4]. Furthermore, under identical instruction paths, the performance (6.5 TFlops vs. 0.39 TFlops) and power consumption characteristics (78W vs. 70W) corresponding to different numerical precisions (FP16 vs. FP32) significantly deviate from the proportional relationships defined in existing architectural documentation [13]. At the application level, despite having similar theoretical computational power (FP32: 12 TFlops vs. 19 TFlops), different devices exhibit vast heterogeneity in their actual performance (973 t/s vs. 6782 t/s) and energy efficiency trends (90W vs. 203W) when executing the same inference task [4].

These empirical observations indicate that the traditional performance evaluation paradigm based on peak computational power implicitly relies on idealized assumptions, such as the complete availability of execution units, the equivalence of instruction paths, and the stability of precision conversion ratios. However, in the application scenarios of constrained GPUs, these premises are often difficult to satisfy. Particularly in contexts where instruction-level selective suppression or precision path limitations exist, a single metric like FLOPs or TOPS is no longer sufficient to accurately characterize the device's true and effective computational power.

To precisely depict the computational behavior of constrained GPUs, this paper deconstructs the broad concept of "computational power" into more granular dimensions. These encompass the actual instruction-level throughput capability, the effective execution ratio of different numerical precisions relative to a baseline precision, and the energy consumption characteristics accompanying the execution process. Building upon this foundation, this paper constructs a set of instruction-level, normalized performance and energy efficiency metric systems. This aims to systematically analyze the behavioral characteristics of constrained GPUs under multi-dimensional combinations of instructions, precisions, and execution units.

### 3.2 Model Assumptions and Applicability Boundaries

To ensure the generalizability and controllability of the hypothesized model, this paper clearly defines the model's applicability boundaries and underlying assumptions, which primarily encompass the following two dimensions:

- a. **Hardware Architecture Programability and Observability:** This model is primarily applicable to AI Accelerators possessing Turing-complete characteristics and fine-grained function control capabilities (e.g., NVIDIA/AMD GPUs). The programmability of such architectures is the foundational prerequisite for constructing instruction-level testing tools and obtaining microarchitectural-level empirical data. Conversely, completely heterogeneous Application-Specific Integrated Circuits (ASICs) or deeply black-boxed devices are explicitly excluded from the model's scope, as they cannot satisfy the input requirements due to a lack of instruction-level controllability and observability.
- b. **Accessibility of Runtime Energy Consumption Data:** The construction and analysis of this model rely on power side-channel data acquired during device operation. This paper recognizes the following three data acquisition methods as credible sources: (a) Utilizing the standard API interfaces provided by the device's onboard sensors and their drivers; (b) Leveraging sensors integrated into system-level components such as the motherboard or PCIe power delivery interfaces; (c) Performing physical measurements using high-precision external instruments (e.g., digital ammeters, oscilloscopes). If the target device cannot obtain valid power consumption data through the aforementioned or equivalent means due to software or hardware limitations, it does not meet the model's application preconditions.

It should be noted that the aforementioned model is primarily used to analyze performance behavior in compute-bound or instruction-bound scenarios. When the device's operating state is predominantly governed by significant non-computational factors (e.g., hitting thermal/power limits, current throttling, or where application performance is primarily dominated by memory bandwidth, thread scheduling, or kernel launch overhead), inferences based on instruction throughput and energy density within the model may no longer strictly hold. In the experimental design of this paper, such interfering factors are mitigated through measures like prolonged steady-state operation, minimization of memory access, and parameter control.

### 3.3 Instruction Throughput and SM Normalization Model

Based on the existing concepts: instruction throughput per unit time (Ginst/s), average power consumption (W), and the number of physical SM/CU units in the device, the following concepts are defined:

- a. Theoretical Single-SM Throughput Coefficient: Let  $N_{sm}$  be the number of execution units in the device, and  $E_{inst}$  be the instruction throughput per unit time for instruction `inst`. For a non-throttled device, the theoretical single-SM throughput coefficient of this architecture for instruction `inst` is defined as:

$$\alpha_{inst} = \frac{E_{inst}}{N_{sm}}.$$

This metric reflects the theoretical peak throughput capability of a single execution unit in this architecture for processing instruction `inst` under baseline conditions, such as consistent architecture generation and idealized operating frequency. This value is typically calibrated based on measured data from standard non-throttled devices (e.g., NVIDIA A100) under optimal experimental conditions.

- b. Actual Per-SM Throughput Coefficient: Let  $N_{sm}^*$  be the number of streaming multiprocessors (SM) of the device, and  $E_{inst}^*$  be the instruction throughput of instruction `inst` per unit time for the target device. Then the actual per-SM throughput coefficient of the device for instruction `inst` is defined as:

$$\alpha_{inst}^* = \frac{E_{inst}^*}{N_{sm}^*}$$

This metric reflects the instruction processing capability exhibited by a single SM of the target device in the current actual runtime environment. Compared to theoretical values, this coefficient is influenced not only by the hardware characteristics of the device itself but may also be constrained by commercial restrictions imposed by manufacturers (such as frequency locking, compute capability capping), as well as external physical limitations including power limits (power wall), thermal design power (TDP) constraints, and cooling conditions.

- c. Effective Instruction Execution Factor: Based on the above calculation, the effective

instruction execution factor for instruction  $\text{inst}$  on the target device is defined as:

$$\mu_{inst} = \frac{\alpha_{inst}^*}{\alpha_{inst}} \times 100\%$$

This factor is used to characterize the actual utilization of the instruction pipeline on the target device. For non-restricted devices or unrestricted instruction types, this value should approach 100%. If  $\mu_{inst}$  is significantly lower than 100%, it indicates the presence of a performance bottleneck during the execution of this instruction on the target device, which may be caused by hardware policies or environmental constraints.

### 3.4 Precision Ratio and the Effective Precision Model

For mainstream high-performance GPUs (such as those from NVIDIA and AMD) and modern CPUs/AI Accelerators supporting SIMD instruction sets, their floating-point computational capabilities generally share the following common characteristics:

1. **Instruction Set and Numerical Standards:** To accelerate scientific computing, these architectures generally adhere to the IEEE 754 floating-point standard and support the FMA (Fused Multiply-Add) instruction at the hardware level, which performs the operation  $a \times b + c$ .
2. **Operation Counting Convention:** Within the standard performance evaluation systems of high-performance computing (e.g., the LINPACK benchmark), a single FMA operation is uniformly counted as 2 FLOPs (1 multiplication + 1 addition).
3. **Microarchitecture Execution Model:** Modern GPU architectures (such as NVIDIA's Ampere/Hopper and AMD's CDNA) typically use the FMA instruction as the fundamental dispatch unit for scalar or vector pipelines within their Streaming Multiprocessors (SMs/CUs), with optimizations designed around its throughput capability.[13]

Based on these commonalities, the FP32 FMA computational capability can be considered as the baseline computational capability at the architectural level. Let the number of device cores be  $N_{core}$  and the peak frequency be  $F$  GHz; then its theoretical FP32 computational power (Tflops) is:

$$P_{FP32}^T = \frac{N_{core} \times F \times 2}{1000}$$

- a. Definition of Theoretical Precision Ratio: Based on NVIDIA's official architecture whitepapers or equivalent microarchitecture design documentation, the theoretical peak

compute performance of the same architecture across different numerical precisions (e.g., FP16, BF16, FP64) is obtained. Let  $P_{pr}^T$  denote the theoretical precision performance of the device, then the theoretical precision ratio relative to FP32 is defined as:

$$S_{pr}^T = \frac{P_{pr}^T}{P_{FP32}^T}$$

This ratio reflects the design proportions of computational units, issue widths, or execution frequencies for different precisions at the architectural level, independent of the specific device's power consumption, thermal constraints, or operating environment.

- b. Engineering Representation Convention for Precision Ratio: In official documentation and engineering practices of vendors such as NVIDIA, the theoretical precision ratio mentioned above is typically not presented as a continuous real number. Instead, it is approximated in the form of integer powers of 2, such as 1x, 2x, 4x, 1/2x, 1/32x, 1/64x. [13] To maintain consistency with this engineering convention, this paper approximates the precision ratio as follows when discrete representation is required:

$$S_{pr}^T = 2^{\text{round}(\log_2 \frac{P_{pr}^T}{P_{FP32}^T})}$$

This approximation is solely intended to describe the hierarchical relationship of precision levels at the architectural level and does not constitute a correction to the theoretical peak performance itself. Actual numerical calibration can be derived from official architecture whitepapers or from measured peak performance obtained under optimal experimental conditions on standard, unrestricted devices (e.g., NVIDIA A100/A800).

- c. Measured Precision Ratio: For a given device, let its measured computational performance at each precision level be  $P_{pr}^{T*}$ . The theoretical precision ratio relative to FP32 is defined as:

$$S_{pr}^{T*} = \frac{P_{pr}^{T*}}{P_{FP32}^T}$$

This value reflects the actual hierarchical relationship of precision levels as exhibited by the device and is used for comparison with the theoretical precision. As before, in practical applications, its

engineering representation is used:

$$S_{pr}^{T*} = 2^{\text{round}(\log_2 \frac{P_{pr}^{T*}}{P_{FP32}^T})}$$

- d. Effective Precision Execution Factor: The effective execution factor of the device for a given precision is defined as:

$$\mu_{pr} = \frac{S_{pr}^{T*}}{S_{pr}^T} \times 100\%$$

This factor characterizes the actual pipeline utilization when the target device performs computations at a specific precision. For non-constrained devices or unrestricted precision types, this value should approach 100%. If  $\mu_{pr}$  is significantly lower than 100%, it indicates that a hardware-related performance bottleneck exists when the target device executes computations at that precision. Since this metric already incorporates normalization, unlike the effective instruction execution factor, it is not visibly affected by external environmental factors.

### 3.5 Instruction-Level Energy Density

The enhancement of a system's (or hardware device's) real-time performance (e.g., by increasing operating frequency or parallelism) is typically accompanied by a rise in power consumption. Power consumption primarily consists of dynamic power and static power. Dynamic power depends directly on the hardware's operating voltage and frequency, while static power is mainly caused by leakage current [9]. It is noteworthy that the actual power consumption level is not limited solely by the hardware's peak performance, but is more critically determined by its performance utilization rate. For example, when the computational capability of hardware (such as an NPU) is fully utilized, the average power may be high; however, due to a significant reduction in execution time, the total energy consumption may actually decrease. Conversely, in idle or low-utilization scenarios, due to the presence of a relatively high baseline power consumption, the energy efficiency ratio deteriorates significantly [1][9].

Based on the above theoretical foundation, we propose:

- a. Instruction-Level Energy Density: Let  $W_s$  represent the device's power consumption in an idle

state, and  $W_{inst}$  be the device's power consumption when executing instruction **inst**. The Instruction-Level Energy Density (unit: mW/Ginst/s) for instruction **inst** is defined as:

$$\lambda_{inst} = \frac{W_{inst} - W_s}{E_{inst}} \times 1000$$

When the device experiences significant pipeline throughput limitations during task execution, this metric can be interpreted as a measure of the functional unit's activity level. A noticeable decline in this value occurs when the device encounters thermal throttling, power limits, or current limits.

- b. Computational Energy Density: Let  $P_{pr}^{T*}$  denote the device's measured computational performance at a specific precision, and  $E_{inst}^*$  represent the instruction throughput per unit time for instruction **inst**. The Computational Energy Density (unit: W/Tflops) for the current device is defined as:

$$\lambda_{pr} = \frac{W_{pr} - W_s}{P_{pr}^*}$$

This definition aligns with the conventional concept of "energy efficiency" and is similar to metrics like SPECaccel\_energy\_base, which are used in SPEC ACCEL benchmarks to compare the energy consumption of a test system against a reference system. A higher value indicates better energy efficiency [24].

The aforementioned metrics are primarily used for power side-channel analysis based on physical factors to assess the activity level of the device's computational units. Referring to the previously proposed Execution Factor, although it is theoretically possible to design an "Energy Density Factor," it is unnecessary to introduce an additional mathematical tool, as the power ratio can be directly utilized as a substitute during the analysis process.

### 3.6 Derivation from Models to Experimental Design

The normalized model for instruction throughput, the precision ratio model, and the energy density model constructed earlier aim to eliminate the interference of device physical scale (e.g., the number of SMs,  $N_{sm}$ ) and baseline power consumption differences on the evaluation by introducing the theoretical per-SM throughput coefficient  $\alpha_{inst}$ , the effective execution factor  $\mu_{inst}$ , and the energy density  $\lambda_{inst}$ . To accurately calibrate these metrics in subsequent experiments

and verify their sensitivity to hardware bottlenecks, this section will derive the necessary control strategies and validation schemes for the experiments based on the mathematical properties of the model variables.

### *3.6.1 Independent Control Strategy Based on Saturation Throughput*

The core metric defined by the model,  $\alpha_{inst} = \frac{E_{inst}}{N_{sm}}$ , aims to reflect the "peak" capability of a single SM. If the test program is constrained by memory bandwidth or instruction issue efficiency rather than by the computational units,  $E_{inst}$  will fail to reach its peak. This leads to an artificially low calculated effective instruction execution factor  $\mu_{inst}$ , resulting in an erroneous "bottleneck" misjudgment.

Therefore, independent control over the input instruction set is essential in the experimental design. The experiments in this chapter will select compute-intensive benchmark programs that can fully utilize the hardware pipeline characteristics, ensuring the measured instruction **inst** is in a "saturation issue" state. By independently controlling the saturation level of the test load, we assume that the decrease in  $\alpha_{inst}^*$  originates solely from hardware physical limitations or vendor policies, not from insufficient parallelism at the software level. This approach guarantees the physical meaningfulness and accuracy of the  $\mu_{inst}$  metric.

### *3.6.2 Cross-Validation Strategy for Theoretical and Measured Precision*

In Section 3.4, we proposed the engineering-metric precision ratio  $S_{pr}^{T*}$  (i.e., in the form of  $2^n$ ) and the measured precision ratio  $S_{pr}^{T*}$ . The model indicates that the Effective Precision Execution Factor  $\mu_{pr}$  should approach 100% or exhibit specific engineering discretization characteristics.

To validate the effectiveness of this model, the experimental design must adopt a multi-precision cross-comparison strategy. We will test the actual computational power  $P_{pr}^*$  for different precisions, such as FP32, FP16, and BF16, on the same device and perform a cross-mapping with the theoretical model  $S_{pr}^T$ . By comparing the deviation between theoretical and measured values, the experiment can not only validate whether  $\mu_{pr}$  effectively identifies precision units that have been "disabled" or not exposed by the vendor, but also analyze whether the actual acceleration ratio of the microarchitecture for different precisions on non-restricted devices aligns with the theoretical expectation of  $2^n$ .

### *3.6.3 Dynamic Monitoring Strategy for Power-Performance Coupling*

The definition of instruction-level energy density,  $\lambda_{inst} = \frac{W_{inst}-W_s}{E_{inst}} \times 1000$ , indicates that this metric is a coupling function of dynamic power consumption and computational throughput. The model posits that the value of  $\lambda_{inst}$  will exhibit abnormal fluctuations when the device approaches the power wall or the Thermal Design Power (TDP) limit.

Given this characteristic, the experimental design cannot merely measure a single steady-state point at the power wall; instead, it necessitates the introduction of a dynamic load scanning mechanism. The experiment will maintain computational load intensity and ensure that the variation in real-time power consumption ( $W_{inst}$ ) matches the change in instruction throughput ( $E_{inst}$ ). This design aims to determine whether the instruction power consumption follows a "linear increase" or becomes "limited." Consequently, the  $\lambda_{inst}$  curve is utilized to validate the model's derivation regarding the relationship between "execution unit activity" and "physical environmental constraints."

### *3.6.4 Overall Mapping of Experimental Design*

In summary, based on the derivation of the aforementioned model characteristics, this control strategy requires that the experiments encompass three levels: 1) Saturation load testing with independent control (corresponding to Strategy 1 above), to establish baseline measurements for the device's  $\alpha_{inst}$  and  $\mu_{inst}$ ; 2) Multi-precision cross-testing (corresponding to Strategy 2 above), utilizing  $\mu_{pr}$  to assess the true utilization efficiency of different architectures in terms of numerical precision; 3) Stress testing (corresponding to Strategy 3 above), utilizing  $\lambda_{inst}$  to identify single-instruction energy efficiency bottlenecks within the device during actual operation.

## 4 Research Hypotheses and Scope

### 4.1 Instruction-Level Performance Limitation Hypothesis

Based on prior empirical findings, disabling Fused Multiply-Add (FMA) instructions when compiling CUDA applications can significantly enhance the device's FP32 computational performance, increasing its performance throughput from 0.39 TFlops to 6.25 TFlops, achieving approximately a 15x performance gain. This phenomenon has been validated not only after migrating the OpenCL application Flux3D to a CUDA environment but also in the prefill and decode stages of llama.cpp, where significant performance improvements were similarly observed. [4] This evidence strongly suggests that FMA instructions play a critical role in the performance-limiting mechanisms of current devices.

Considering that FMA instructions are a core component of the IEEE 754 floating-point arithmetic standard and are highly prevalent in various CUDA source codes, [13] the actual AI application workloads involve instruction sets far more complex and diverse than simple matrix multiplication-addition computations. Building on this, this paper proposes the hypothesis that the performance bottleneck of the constrained AI accelerator is not solely specific to the FMA instruction but is closely related to the "instruction type-combination pattern." In other words, the performance limitation mechanism may depend on specific instruction sequences or computational combinations, meaning that similar performance ceilings may exist for other instruction types or their specific combinations within particular computational contexts, in addition to FMA.

### 4.2 Hypothesis on Precision-Execution Unit Correlation

Previous experimental data revealed significant differences in the response of various numerical precisions to the disabling of FMA instructions: while disabling FMA instructions substantially improved FP32 performance, it had almost no effect on FP16 and BF16 performance. This phenomenon suggests that the hardware throttling mechanism may not be single-dimensional.

Further analysis of the underlying software stack indicates that, whether based on general frameworks like PyTorch or lightweight inference engines like llama.cpp, the core essence of their matrix operations is ultimately calling NVIDIA's official cuBLAS or cuDNN libraries. According to explicit statements in NVIDIA's official documentation, when performing low-precision operations such as FP16, BF16, and INT8, the system preferentially activates Tensor Cores instead

of traditional CUDA Cores (scalar/vector cores). [25] This switching of hardware execution units implies that, on the same device, numerical computations of different precaracies may be mapped to entirely different microarchitectural execution units.

Therefore, this paper proposes a hypothesis: there is a strong coupling relationship between the throttling strategy of restricted AI Accelerators and numerical precision as well as physical execution units. That is, different execution units (such as CUDA Cores and Tensor Cores) implement differentiated or asymmetric throttling strategies for computational workloads of different precaracies.

### **4.3 Hypothesis on Power-Performance Side-Channel Analysis**

Existing research indicates that constrained AI Accelerators often exhibit distinctly different power consumption characteristic curves under varying computational loads [4]. In fact, microarchitecture energy models have long been a key research direction in the field of computer architecture. Especially in the post-Dennard scaling era, as transistor size scaling no longer automatically reduces power density, energy efficiency (energy-performance ratio) has become a design goal equally important as, if not more critical than, peak performance [1]. Against this backdrop, the actual power consumption level of a processor is not only constrained by the hardware's physical peak performance but is also directly determined by the performance utilization and activity level of its internal computational units [1].

Based on the above analysis, this paper proposes the hypothesis that the performance limitation strategy of a constrained AI Accelerator can be observed via side-channel analysis. Specifically, when the instruction pipeline is artificially restricted, the device will exhibit anomalously low power consumption and low energy efficiency. Therefore, changes in power consumption and energy efficiency can be considered significant physical indicators reflecting the actual activity level and limitation state of the computational units.

### **4.4 Research Scope**

The core of this research lies in detecting and evaluating the practical usability of the NVIDIA CMP 170HX, as a representative of constrained AI accelerators, in artificial intelligence tasks. To ensure the focus of the study and the interpretability of experimental results, this paper confines its research scope to the analysis of instruction throughput and corresponding power consumption characteristics for medium and low precision (FP32, FP16, BF16) computational instructions.

Considering the specific architectural characteristics of the research subject and the need for experimental control, this study excludes several variables:

- a. **Precision Range:** Limited by hardware architecture support and the prevalence of current mainstream AI models, this paper does not involve computational analysis for very low precision formats (e.g., FP8, FP4). Simultaneously, considering that double-precision floating-point (FP64) is primarily used for scientific computing rather than mainstream AI inference or training, it is also excluded from the research scope.
- b. **System Variables:** To minimize confounding variables and precisely identify the performance limitations of the computational unit itself, this paper does not explicitly discuss factors such as integer computation, memory movement, thread synchronization, and PCIe data transfer bottlenecks.
- c. **Software Stack:** The research will bypass the overhead of upper-layer framework scheduling, focusing instead on low-level hardware computational behavior.

## 5 Experimental Design and Methodology

### 5.1 Selection of Typical Inference Operators and Instruction Mapping

To accurately capture the computational characteristics of edge-side AI inference, this study selects the widely used Transformer architecture (e.g., Large Language Models) and the Stable Diffusion model as the analysis subjects. The operator selection process adheres to the following rigorous experimental steps:

- a. Environment Setup and Configuration: Compile the CUDA source code for stable-diffusion.cpp and llama.cpp. To ensure data purity for analysis, disable the CUDA Graphics capture function in the compilation configuration, thereby obtaining underlying raw operator call information during profiling.
- b. Performance Profiling: On an experimental platform equipped with an NVIDIA RTX 3080 GPU, utilize the NVIDIA Nsight Systems tool to conduct fine-grained profiling of the complete model inference process.
- c. Data Collection and Filtering: For each stage of the inference pipeline, extract CUDA GPU Kernel Summary data.

Given the high heterogeneity and complexity of kernel types encountered during inference, this study filters and selects the top 20 most frequently called core operators as analysis samples based on their invocation frequency.

Based on similarities in computational characteristics, the filtered core operators are categorized into the following six types:

- a. **GEMM** Category: Encompasses various matrix multiplication and addition operations, including Linear, BMM, and other operators.
- b. **Conv** Category: Includes standard convolution and transposed convolution computations.
- c. **Softmax** Category: Approximately corresponds to the Attention mechanism segment, covering various Softmax variant computations.
- d. **Activate** Category: Includes non-linear activation functions such as SiLU and GELU.
- e. **Norm** Category: Encompasses various normalization operations such as GroupNorm and RMSNorm.
- f. Other Category: Primarily consists of auxiliary computations including memory operations, broadcasting mechanisms, and positional encoding.

Based on the above categorization, the proportion of total inference time consumed by each operator category is calculated (normalized to 100%). Experimental data shows that the proportion of inference time for common kernels in the Transformer inference process is illustrated in Figure 5-1; the proportion for the Stable Diffusion inference process is illustrated in Figure 5-2.

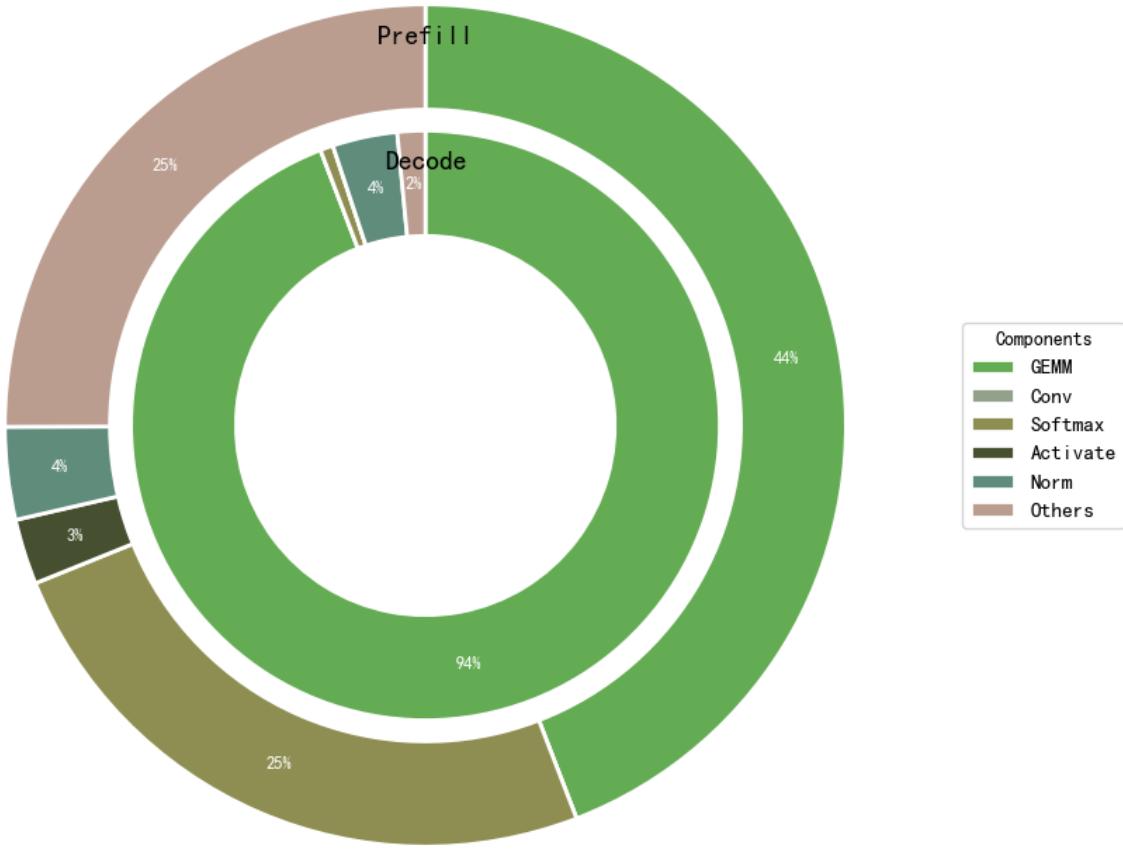


Figure 5-1: Proportion of Each Operator in the Transformer Inference Stage

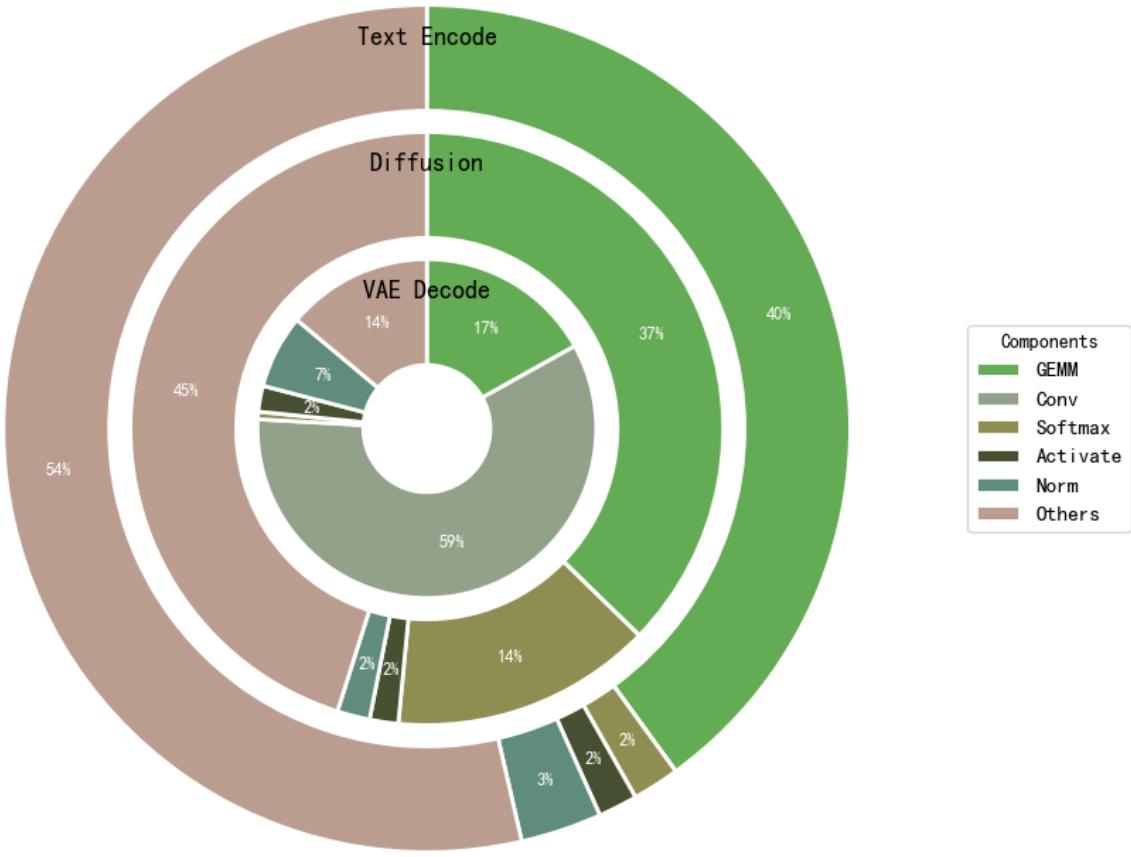


Figure 5-2: Proportion of Each Operator in the Stable Diffusion Inference Stage

For the aforementioned core operators, based on the NVIDIA CUTLASS library documentation and GPU architectural characteristics, their computational logic is mapped to the underlying CUDA instruction set, with a detailed analysis as follows:

- GEMM and Conv Classes:** The mathematical essence of matrix multiplication and convolution operations lies in multiply-accumulate operations within sliding windows. Their computational core primarily relies on arithmetic logic units, with corresponding key CUDA instructions including FMA (fused multiply-add), MUL (multiplication), and ADD/SUB (addition/subtraction) [26].
- Softmax Class:** The Softmax operation involves exponentiation, summation, and normalization. Such transcendental functions are typically executed by the GPU's Special Function Units (SFUs), mainly comprising instructions such as EXP (exponentiation), DIV (division), MAX (maximum), and ADD.

- c. Activation Class: For instance, the GELU activation function involves approximations of the error function and may invoke the ERF instruction; SiLU (Sigmoid-weighted Linear Unit) involves SIGMOID operations (which can be implemented through combinations of EXP and DIV). These computations are also primarily mapped to SFUs, with key instructions including TANH, SIN, MUL, ADD, DIV, and ERF [9].
- d. Normalization Class: Normalization operations require computing mean, variance, or root mean square, involving complex parallel reduction (e.g., parallel summation) and square root operations. The main instructions include ADD, MUL, DIV, SQRT (square root), and RSQRT (reciprocal square root) [25].

It should be noted that the actual generated instruction set may be influenced by differences in GPU microarchitectures (e.g., Turing, Ampere) and backend compiler optimization strategies. The above analysis is based on a theoretical mapping of typical computational patterns [13][25].

## 5.2 Categorization of CUDA Instruction Types

Based on the instruction frequency statistics from Chapter 4, this study categorizes CUDA instructions into four types according to computational complexity and hardware support characteristics. This classification system comprehensively considers the probability of instruction occurrence in AI inference workloads, the specificity of hardware execution units, and the requirements for numerical computational precision, providing a theoretical framework for subsequent micro-benchmarking. The classification is shown in Table 5-1.

Type	Instruction Examples
<b>Basic Arithmetic</b>	add: Addition mul: Multiplication div: Division
<b>Compound Arithmetic</b>	mad: Fused Multiply-Add
<b>Elementary Functions</b>	exp: Natural Exponential log: Natural Logarithm sqrt: Square Root rcp: Reciprocal Approximation pow: Power Function
<b>Advanced Functions</b>	sin/cos: Trigonometric Functions tanh: Hyperbolic Tangent Calculation erf: Error Function

Table 5-1: Classification of CUDA Programming Instruction Types

### 5.3 Rationale for Experimental Subject Selection

This experiment aims to investigate the performance differences of GPUs with different market positioning and hardware specifications under AI inference tasks. Therefore, we established an experimental scheme with NVIDIA CMP 170HX as the core, supplemented by NVIDIA A800 and GeForce RTX 3080 10GB as comparative groups. The specific selection rationales are as follows:

#### a. Core Experimental Subject: NVIDIA CMP 170HX

The CMP 170HX is equipped with NVIDIA's flagship GA100 core and 8GB of high-bandwidth HBM2e memory. As the highest-end model within the CMP series, it was originally targeted for professional cryptocurrency mining, but its architectural essence is a restricted version of a GPGPU. Due to its highly representative scale of compute units and its classification under specific market supply restrictions, it is selected as the primary restricted AI accelerator test subject for this study [4].

#### b. Unrestricted Control Group: NVIDIA A800

To compare performance between restricted and unrestricted environments, and considering the feasibility of acquiring top-tier data center hardware, we selected the NVIDIA A800 as the baseline for an unrestricted device. Although the A800 has a limited NVLink interconnect speed compared to the A100 (reduced from 600 GB/s to 400 GB/s), its core computational specifications—including the number of CUDA Cores, Tensor Cores, memory capacity, and frequency—are identical to the A100, and it is also based on the GA100 core architecture. Given that this experiment primarily focuses on single-card computational performance rather than inter-card communication, the A800 can be considered a perfect substitute for the A100 in single-card testing scenarios, offering high comparative value [27].

#### c. Consumer-grade Reference Control: NVIDIA GeForce RTX 3080 10GB

To extend the applicability of this study's findings to a broader computational ecosystem, we introduced the NVIDIA GeForce RTX 3080 10GB as a representative consumer-grade GPU. The RTX 3080 utilizes the GA102 core. While it shares the Ampere microarchitecture with the GA100, ensuring instruction set commonality, there are architectural differences in process technology, cache hierarchy, and double-precision floating-point capabilities. By including the RTX 3080, we can construct a cross-SKU comparison dimension encompassing "Restricted-grade (CMP)", "Data Center-grade (A800)", and "Consumer-grade (RTX)" devices, enabling an in-depth analysis of

precision ratios, power consumption strategies, and performance across different market segments [10].

#### 5.4 Instruction-Level Testing Methodology

This test adopts a precision-driven testing strategy, dividing the test dimensions into three primary categories based on the varying precision requirements of contemporary AI inference: FP32 (single-precision floating-point), FP16 (half-precision floating-point), and BF16 (brain floating-point). Regarding specific test coverage, considering that modern AI inference typically employs a mixed-precision computing paradigm of "low-precision for linear operations and high-precision for nonlinear operations," we formulated differentiated coverage plans. For FP32 precision testing, a full-coverage mode is employed, encompassing all instruction categories including basic arithmetic, composite arithmetic, elementary functions, and advanced functions, aiming to comprehensively evaluate general-purpose computing capabilities in a high-precision environment. For FP16 and BF16 precision testing, a targeted coverage strategy is adopted, focusing on testing arithmetic instructions and composite instructions to reflect the real performance of core computational paths in AI inference.

To deeply analyze the impact of different code generation paths and hardware microarchitecture characteristics on performance, we introduced multi-dimensional variable controls in the testing. First, for instructions supporting intrinsics (Intrinsic Instructions, also known as intrinsic functions in CUDA programming), they are divided into two test channels: standard library calls (also known as standard instructions, native instructions, or standard functions in CUDA programming) and intrinsic compilation, to evaluate the instruction-level overhead of different compilation optimization strategies. Second, leveraging the characteristics of the Ampere architecture, we separately test vectorized and non-vectorized instructions for FP16 and BF16 precision, thereby quantifying SIMD (Single Instruction, Multiple Data) execution efficiency. Furthermore, we further distinguish between the two execution paths of CUDA Cores (traditional streaming processors) and Tensor Cores, aiming to reveal the acceleration effect and energy efficiency ratio of specialized acceleration units in actual computational workloads.

In terms of defining and measuring performance metrics, this study will precisely record the single-instruction throughput rate of each instruction, with the unit Ginst/s (billion instructions per second), to evaluate the hardware's instruction issue and execution capabilities. Simultaneously, we record the average power consumption of instructions during sustained full-load operation for subsequent energy efficiency analysis. For arithmetic instructions and fused instructions, we will

also combine the hardware operating frequency to calculate and record the measured sustained floating-point computational performance, verifying the actual achievement rate of theoretical peak performance.

Regarding the specific design of micro-benchmark programs, all test programs are directly written in CUDA C++ language and adhere to strict micro-benchmark design principles to eliminate interference factors. First, program design minimizes memory bottlenecks through data prefetching and register reuse strategies, ensuring that the primary testing bottleneck resides in computational throughput rather than memory bandwidth. Second, dynamically generated random numbers are introduced as operands to prevent the compiler from performing excessive optimizations such as constant propagation or dead code elimination. Third, techniques like loop unrolling and pipeline parallelism are employed to ensure the hardware's computational pipeline is under sustained high-pressure saturation. Finally, all tests utilize a warm-up mechanism, running for a period before formal data collection to allow the GPU to reach thermal equilibrium and frequency stability, thereby obtaining accurate power consumption and performance data. For specific implementation details of the test kernels, please refer to the relevant sections in the appendix of this paper.

## 5.5 Automation Testing Tools and Workflow

To ensure the reproducibility of experimental data, improve data acquisition efficiency, and minimize errors introduced by manual intervention, this research developed and implemented a dedicated automated testing and monitoring framework. This tool integrates task scheduling, execution monitoring, and data logging functionalities, enabling it to adapt to complex testing environments.

In terms of intelligent file processing and task scheduling, the tool possesses flexible identification and management capabilities. It supports filtering files by extension (defaulting to .py and .sh), while also intelligently recognizing Linux executables without extensions (including ELF binaries or scripts with Shebangs). It further supports the exclusion of files matching specific patterns using regular expressions. Additionally, the tool features built-in automatic permission management, attempting to add execute permissions upon detecting non-executable files to ensure the continuity of the testing workflow.

For the GPU power monitoring module, the tool employs a high-precision telemetry strategy. To mitigate noise from GPU frequency fluctuations and unstable loads during the initialization phase,

the system incorporates a delayed sampling mechanism. This means data collection begins only after a preset wait time (defaulting to 3 seconds) following program launch. The monitoring process utilizes periodic sampling to record the average, minimum, and maximum power consumption of the GPU during the execution period. Concurrently, to ensure data validity, the system includes a runtime validation mechanism that issues warnings for samples with excessively short execution times, automatically discarding invalid power data caused by insufficient load.

Regarding comprehensive execution control and log recording, the framework provides a robust testing workflow. The tool supports executing individual test scripts multiple times, facilitating subsequent statistical analysis and error assessment. It also allows passing unified command-line arguments to all scripts, simplifying configuration management across different testing scenarios. For log output, the system supports generating both human-readable text logs and structured JSON logs. These logs provide detailed coverage of script execution status, standard output streams, comprehensive power consumption statistics, execution timestamps, and runtime environment information. Furthermore, the tool features robust exception handling, supporting graceful exit via interrupt signals (e.g., Ctrl+C) and automatically saving completed experimental data upon forced termination, effectively preventing data loss.

## 6 Experimental Results

### 6.1 Overall Characteristics

Using the Instruction Execution Factor defined previously, we plotted the heatmap of the Instruction Execution Factor for the 170HX relative to the A800 across different precisions, as shown in Figure 6-1:

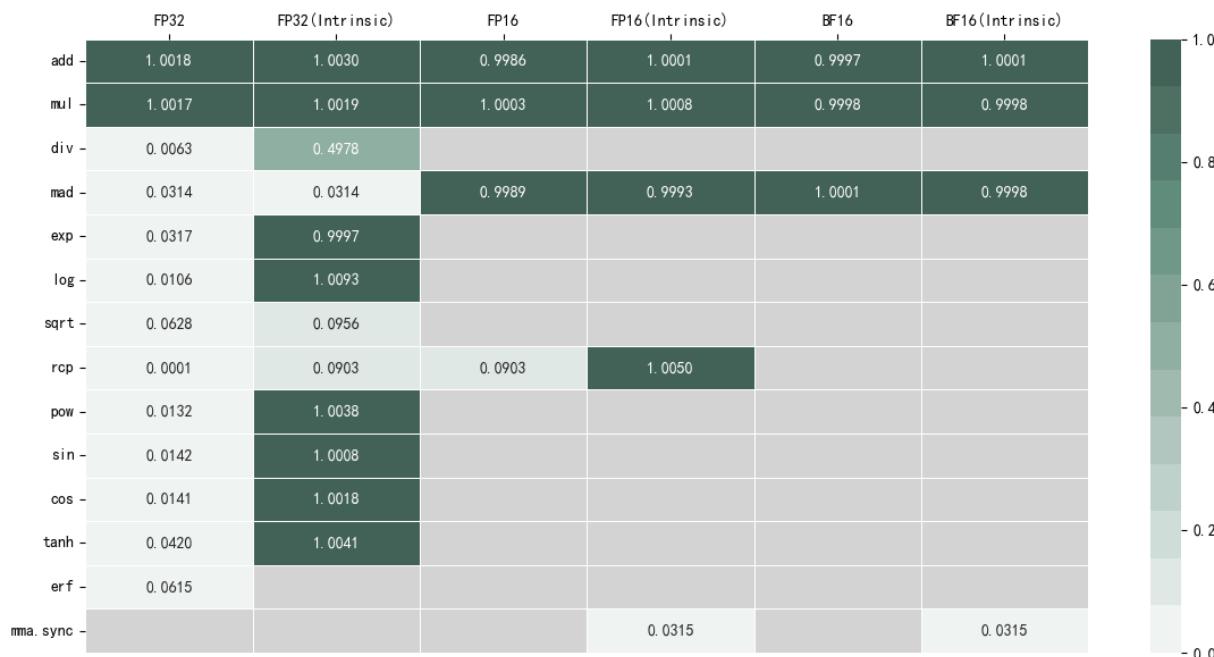


Figure 6-1: Instruction Execution Factor - Precision Graph for 170HX

Figure 6-1 visually illustrates the instruction execution efficiency of the restricted AI accelerator 170HX under the combined influence of instruction path, numerical precision, and execution units. It presents a heatmap distribution of the Effective Execution Factor for the CMP 170HX across different numerical precisions (FP32 / FP16 / BF16), different instruction invocation paths (standard library vs. intrinsic), and different instruction types.

The numerical values in the figure represent the achievable throughput ratio of the 170HX relative to the corresponding instruction on the standard GA100 (A800) after normalizing for the number of SMs. Lower values indicate stronger performance inhibition for that instruction under that specific precision and invocation path. Grayed-out areas indicate that the instruction path does not

exist or is not comparable at that precision.

In stark contrast to basic arithmetic instructions, numerous composite arithmetic and transcendental function instructions exhibit a significant collapse in execution factors under the FP32 standard path. For instance, the effective execution factors for functions such as `mad`, `div`, `exp`, `log`, `sin`, `cos`, and `tanh` generally fall within the range of 0.01–0.06, indicating a systematic suppression of their throughput capability.

Notably, this inhibition does not apply uniformly across all invocation paths. Under the intrinsic path, the execution factors for many of these instructions can recover to levels close to 1.0, indicating that the corresponding hardware functional units are not physically disabled but are subject to restriction policies related to the invocation path.

Furthermore, the figure reveals several counter-intuitive characteristics. For example, the `mad` instruction is significantly inhibited under FP32 precision but appears fully released under FP16/BF16 precisions. The `mma.sync` instruction shows extremely low effective execution factors in both FP16 and BF16 paths.

These phenomena suggest that the performance-limiting mechanisms of the 170HX are not based solely on instruction categories or execution unit types. Instead, they involve complex coupling with numerical precision, execution paths, and underlying scheduling policies. Quantitative analysis of these characteristics and inferences regarding potential mechanisms will be discussed in subsequent sections.

## 6.2 FP32 Instruction Throughput Characteristics

Based on the Instruction Execution Factor heatmap (Figure 6-1) constructed in Section 6.1, this section focuses on the specific instruction behavior characteristics of the CMP 170HX under FP32 precision. It analyzes the performance differences among basic arithmetic instructions, composite operation instructions, and special function instructions, combined with the energy density metric to provide a deeper exploration of the actual utilization of its hardware execution units.

Figure 6-2 shows the precision under FP32, using the Precision Execution Factor to characterize the degree of limitation experienced by various arithmetic instructions.

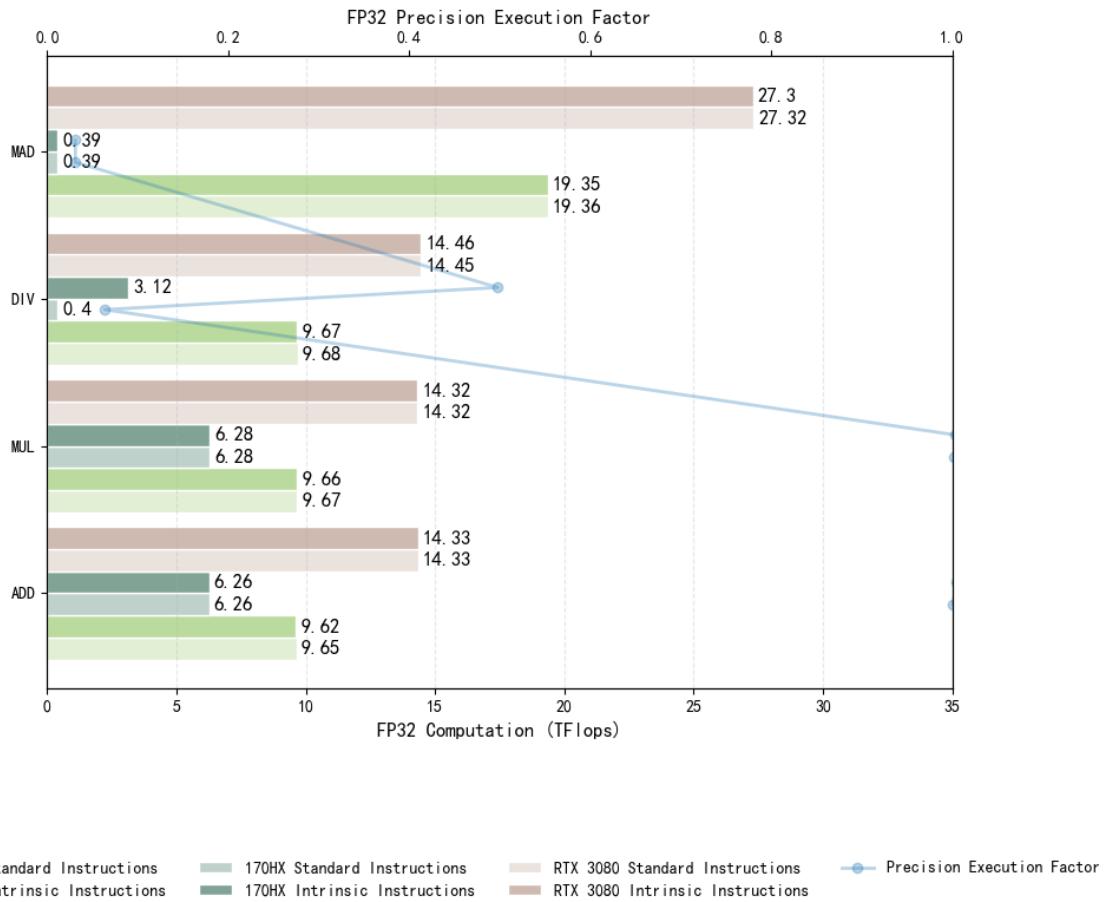


Figure 6-2: FP32 Arithmetic Floating-Point Performance Comparison: 170HX-A800-RTX3080

Experimental data indicates that the CMP 170HX imposes differentiated performance constraints on different types of FP32 arithmetic instructions. The effective Instruction Execution Factor for basic addition and multiplication instructions shows no significant throughput suppression, approaching the theoretical levels of non-restricted devices. However, composite operation instructions compliant with the IEEE 754 standard—particularly fused multiply-add (FMA) instructions—exhibit extremely low Instruction Execution Factors, suffering from severe performance throttling. This phenomenon not only corroborates the community observation that 'disabling FMA can improve performance' but also clarifies, at the instruction level, that the primary performance bottleneck of the CMP 170HX lies in the blocking of the FMA instruction execution path.

Simultaneously, we observed that mathematical functions under FP32 face even stricter limitations, as shown in Figure 6-3:

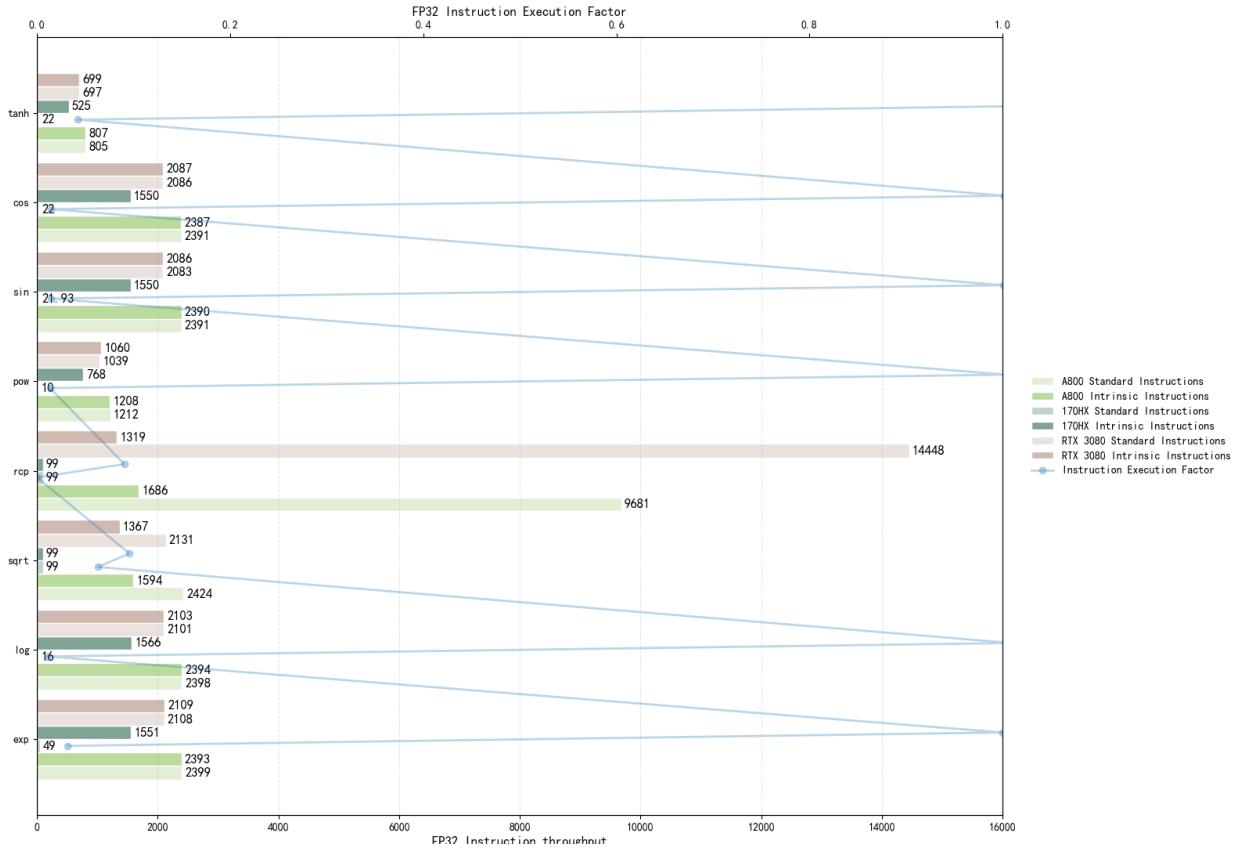


Figure 6-3: FP32 Mathematical Floating-Point Instruction Performance Comparison: 170HX-A800-RTX3080

Beyond core arithmetic instructions, mathematical library functions under FP32 precision (typically mapped to the GPU's Special Function Units, SFUs) are subjected to more stringent restrictions. The throughput of transcendental function instructions such as exp, log, sin, and cos demonstrates a significant collapse. During comparative analysis, we noted that the non-restricted control groups (RTX 3080 and A800) exhibited exceptionally high values for the RCP (reciprocal calculation) instruction under the standard library call path (as high as 69,000 Ginst/s). This suggests its underlying execution likely utilizes a different, optimized path compared to standard arithmetic instructions. To ensure the rigor and comparability of the analytical baseline, this section will employ DIV instruction data as a substitute benchmark for comparisons involving division-related operations.

To further reveal the underlying hardware state associated with the restricted instructions, this section conducted a side-channel analysis using the energy density metric. On non-restricted GPUs (such as the RTX 3080 and A800), the performance of instructions under the two paths of intrinsic

compilation and standard library calls tends to be consistent, with differences primarily stemming from compiler-level calling overhead. However, on the CMP 170HX, instruction restrictions under the standard library call path are far stricter than under the intrinsic path, making the energy density of standard library instructions a sensitive indicator reflecting the utilization rate of its hardware units. We present the relevant information in Figure 6-4.

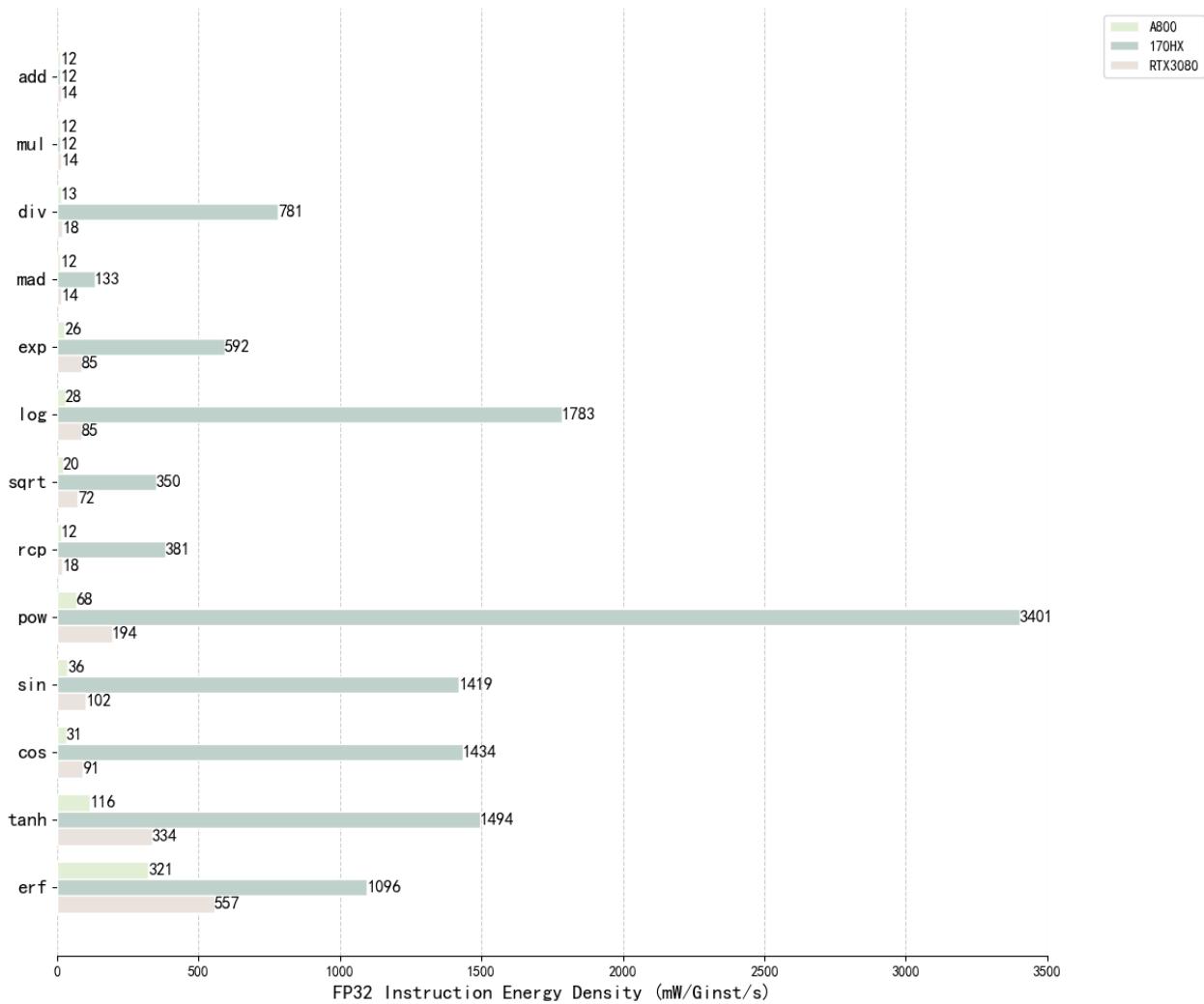


Figure 6-4: FP32 Instruction Energy Density Comparison: 170HX-A800-RTX3080

By comparing the standard instruction energy density charts of the CMP 170HX and the A800, a clear observation can be made: despite sharing the same GA100 architecture, the CMP 170HX incurs significantly higher energy cost per billion instructions completed (Ginst/s) when executing restricted instructions (such as FMA and various SFU functions) compared to the A800. This nonlinear characteristic of "high energy density, low throughput" indicates that when executing restricted instructions, the physical computing units of the CMP 170HX may not be operating in

an efficiently saturated state. Instead, they are likely forced into a dormant state or a low-efficiency operating mode through some gating mechanism, leading to a severe degradation in energy efficiency.

### 6.3 FP16 Instruction Throughput Characteristics

Building upon the instruction execution factor heatmap constructed in Section 6.1, this section systematically analyzes the instruction-level performance characteristics of the CMP 170HX under FP16 precision. Given that FP16 precision is primarily used for linear operations in contemporary AI inference workloads, with relatively infrequent calls to mathematical functions, this study focuses on the performance evaluation of arithmetic instructions, following the testing strategy outlined in Section 5.4. As illustrated in Figure 6-5, the FP16 arithmetic floating-point performance of this device demonstrates a distinct selective restriction pattern.

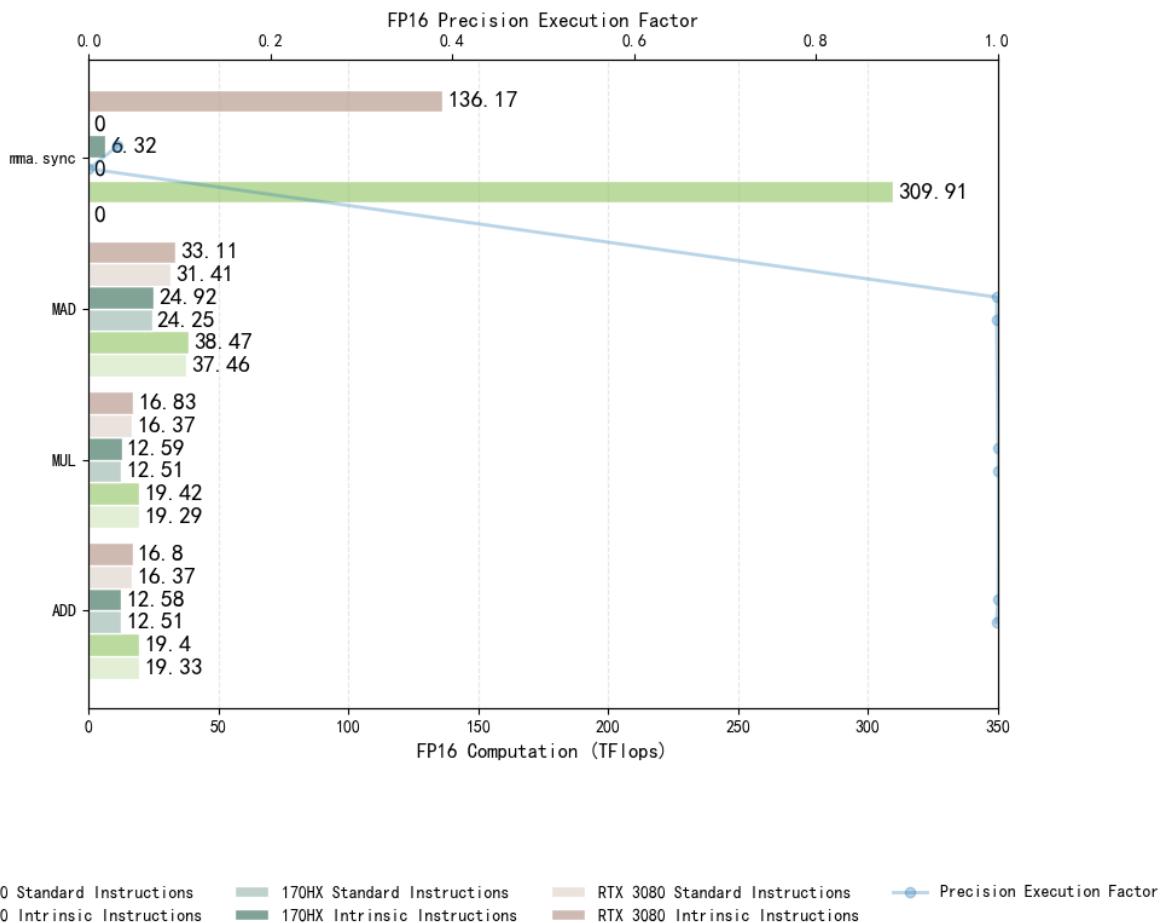


Figure 6-5: FP16 Arithmetic Floating-Point Performance Comparison: 170HX-A800-RTX3080

The experimental results indicate that the CMP 170HX does not impose significant performance restrictions on the majority of basic arithmetic instructions (e.g., **ADD**, **MUL**) and composite arithmetic instructions (e.g., **MAD**) under FP16 precision. Their effective Instruction Execution Factor approaches the theoretical level of the unrestricted reference device (A800). However, the matrix multiply-accumulate instruction **mma.sync**, based on Tensor Cores, exhibits significant performance suppression, with its Effective Execution Factor dropping below 0.05. This suggests the instruction path is subject to systematic restrictions.

To further investigate the hardware execution state of the restricted instruction, this study introduces the instruction-level energy density metric for side-channel analysis. As shown in Figure 6-6, leveraging the characteristic of minimal restrictions under FP16 precision, this experiment employs inline vectorized instructions to maximize computational throughput and establish a lower baseline for energy density. The analysis reveals that, except for **mma.sync**, the energy density of other FP16 arithmetic instructions remains highly consistent with that of the A800 device, validating the normal operational state of their hardware execution units.

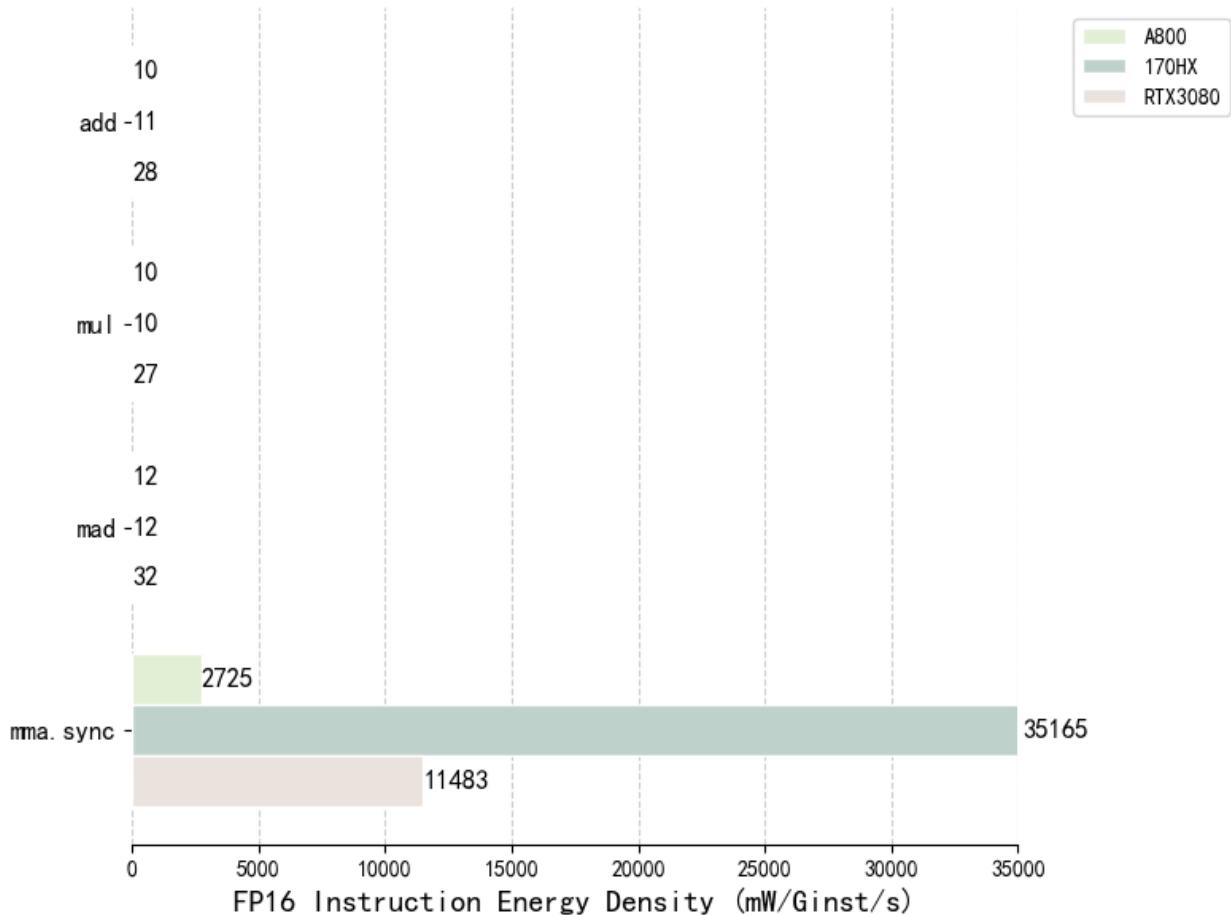


Figure 6-6: FP16 Arithmetic Instruction Energy Density Comparison: 170HX-A800-RTX3080

To quantitatively analyze the anomalous behavior of the `mma.sync` instruction, this study further employs the Computational Energy Density metric (unit: W/Tflops) defined in Chapter 3. This metric more directly reflects the energy efficiency characteristics of the execution units.

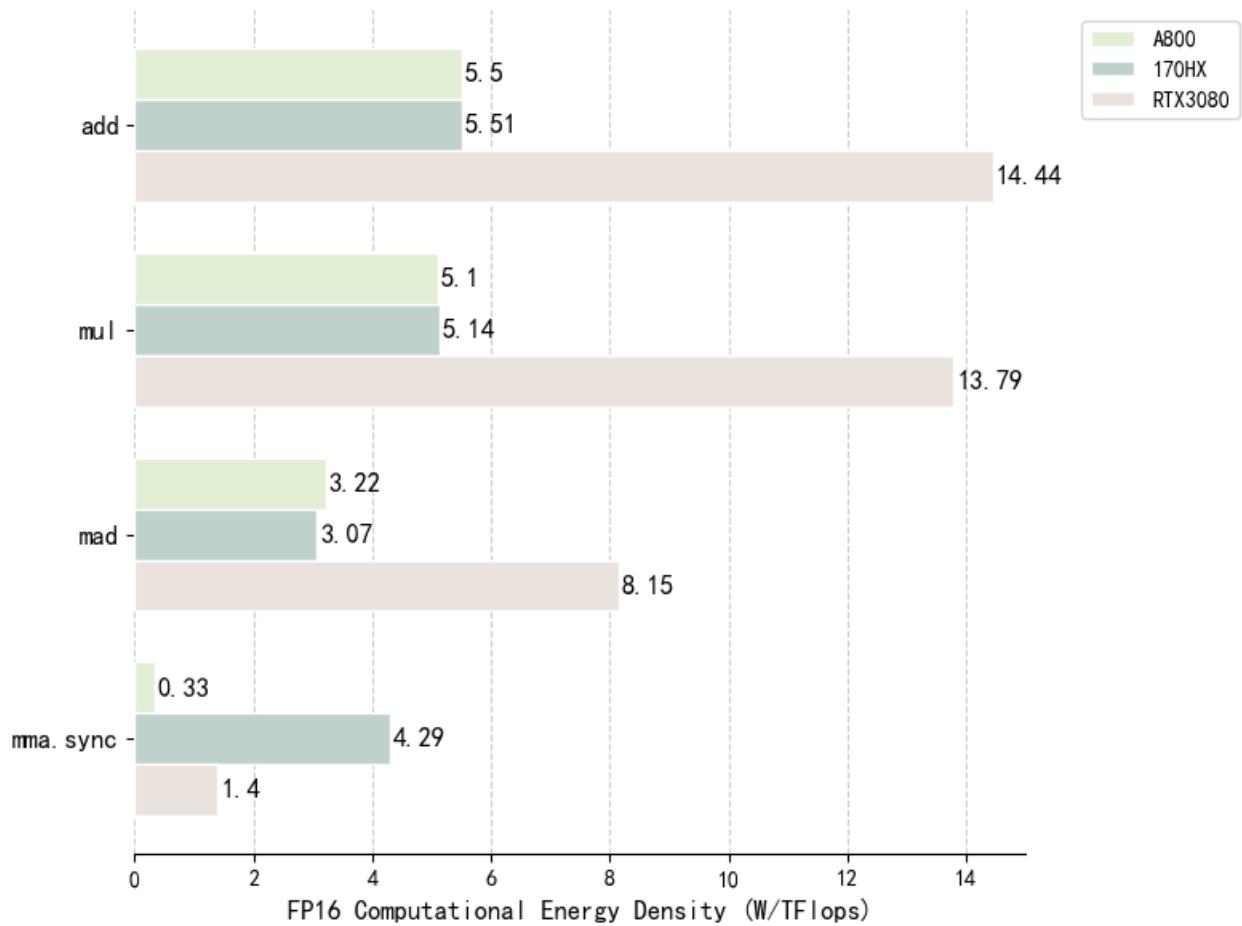


Figure 6-7: FP16 Computational Energy Density Comparison: 170HX-A800-RTX3080

As shown in Figure 6-7, although the CMP 170HX shares the same GA100 microarchitecture foundation as the A800, the former exhibits an abnormally high Computational Energy Density (approximately 4.29 W/Tflops) when executing the FP16 `mma.sync` instruction. This value is significantly higher than the corresponding value for the A800 (approximately 0.33 W/Tflops). This phenomenon indicates that the Tensor Cores, which theoretically should possess superior energy efficiency, not only have restricted computational output on the CMP 170HX but also suffer from severely degraded energy efficiency characteristics. Their actual performance is even lower than that of the traditional CUDA Core path. This result strongly suggests that the Tensor Core

execution units may be actively suppressed by hardware-level gating or scheduling policies.

#### 6.4 BF16 Instruction Throughput Characteristics

This section further examines the instruction-level performance characteristics of CMP 170HX under BF16 precision, as shown in Figures 6-8 and 6-9.

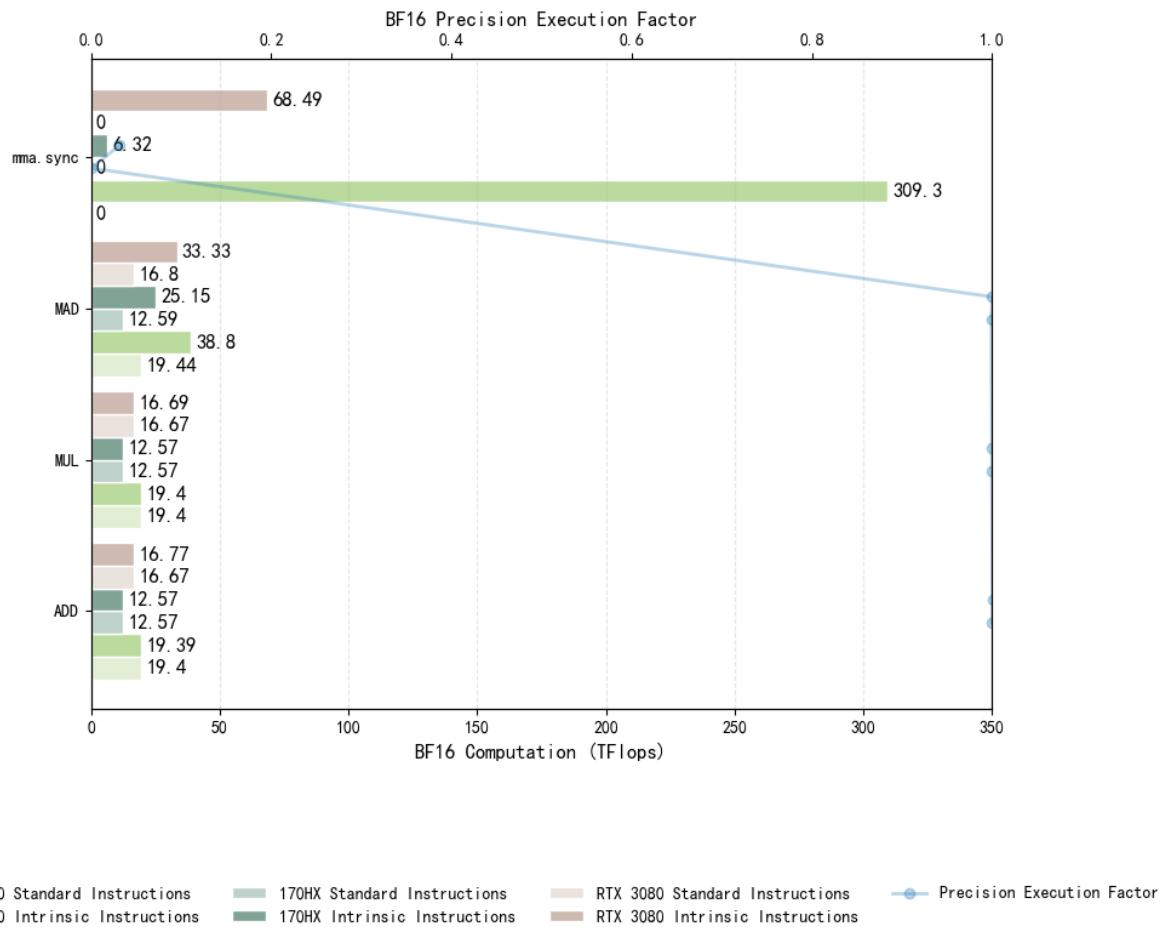


Figure 6-8: BF16 Arithmetic Floating-Point Performance Comparison: 170HX-A800-RTX3080

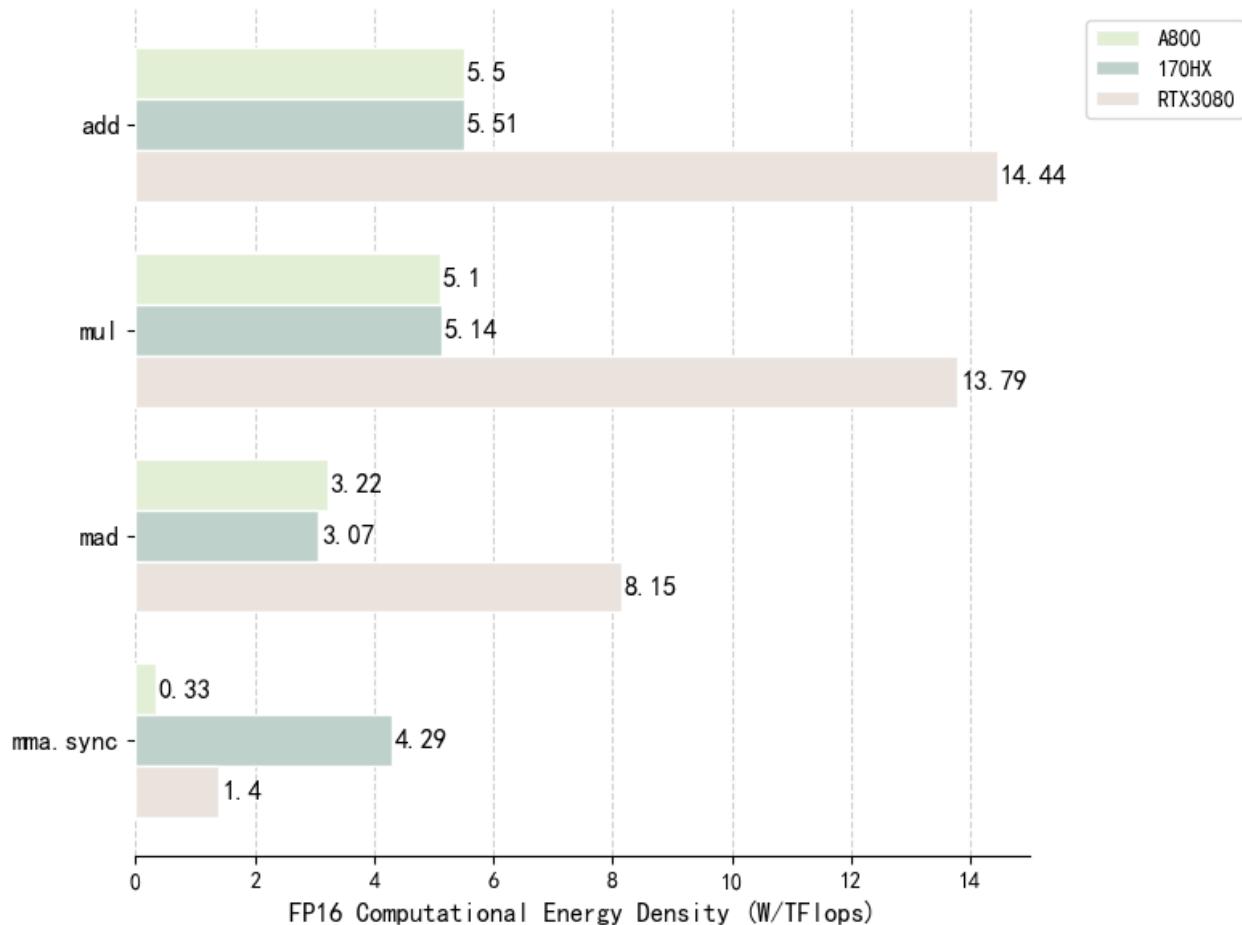


Figure 6-9: BF16 Computational Energy Density Comparison: 170HX-A800-RTX3080

Through comparative analysis of arithmetic floating-point performance and computational energy density achieved via intrinsic functions, the following key findings are derived:

First, the constraint patterns under BF16 precision are highly consistent with those under FP16. Except for the `mma.sync` instruction based on Tensor Cores, other arithmetic instructions show no significant performance throttling. The effective execution factor of this instruction under BF16 precision also drops below 0.06, indicating that the constraint mechanism maintains cross-precision consistency across different low-precision formats, likely due to a unified hardware control strategy for Tensor Core units.

Second, the experimental results reveal that the actual computational performance of BF16 standard library `FMA` functions only reaches approximately 50% of that achieved by intrinsic vectorized implementations. This phenomenon may stem from the Ampere architecture's special handling of BF16 data types: the standard library call path may trigger additional data format

conversions or precision compensation operations, resulting in reduced effective computational throughput. Notably, this behavior is also observed on A800 devices, indicating it is an architecture-level design feature rather than a limitation specific to CMP 170HX.

Third, cross-device comparative analysis reveals interesting architectural differences: the RTX 3080 (based on the GA102 core) achieves about 50% of the FP16-equivalent performance when executing BF16 mma.sync instructions, whereas the A800 (based on the full GA100 core) maintains a BF16/FP16 performance ratio close to 1:1. This discrepancy may reflect NVIDIA's differentiated feature-gating strategy across product lines, where consumer-grade GPUs impose additional performance constraints on BF16 Tensor Core operations to achieve product-tier market segmentation.

In summary, BF16 and FP16 precisions exhibit highly similar constraint characteristics on CMP 170HX, with the core restrictions concentrated on the Tensor Core execution path. This finding provides a critical basis for designing subsequent operator-level mitigation strategies: in low-precision inference scenarios, priority should be given to computational paths that bypass Tensor Core dependencies, adopting alternative implementations that can be efficiently executed by CUDA Cores.

## 7 Analysis and Discussion

### 7.1 Analysis of the Multi-Level Restriction Mechanism in Restricted AI Accelerators

Based on the experimental data presented earlier, this study first systematically verified the research hypotheses proposed in Chapter 4. The experimental results strongly confirm the existence of extensive and profound instruction-level performance restrictions within the restricted AI accelerator, and these restrictions exhibit a strong correlation with the low computational performance observed at the macro level. Specifically, while the throughput of basic arithmetic instructions (such as `ADD`, `MUL`) for the CMP 170HX at FP32 precision largely meets specifications, the significantly degraded floating-point computational performance at the application layer is primarily attributed to the severe suppression imposed on compound arithmetic instructions, especially `FMA`. This finding not only validates the community observation that "disabling `FMA` can improve performance" but also theoretically confirms that the core performance bottleneck lies in the blocking of specific instruction paths.

Further analysis indicates that this instruction-level restriction exhibits a significant strong coupling characteristic among "instruction type, numerical precision, and execution unit." Experimental data show that the compute path based on CUDA Cores primarily suppresses `FMA` instructions at FP32 precision, while the path based on Tensor Cores demonstrates a more comprehensive performance block on the `mma.sync` instruction at FP16/BF16 precision. This suggests that when implementing restriction strategies, the manufacturer has formulated differentiated control logic for different microarchitectural execution units, rather than adopting a global compute frequency downclock.

More importantly, by comparing the performance differences between intrinsic instructions and standard library instructions, this study reveals the hierarchical nature of the restriction mechanism. For certain instructions (such as `exp`, `sin`, `cos`, `tanh`), the Effective Execution Factor approaches 1.0 under the intrinsic path but collapses to extremely low levels under the standard library path; whereas other instructions (such as `fma`, `mma.sync`) are subject to strong suppression regardless of the calling path. This disparity suggests that the restriction mechanism is not single-dimensional but may encompass multiple layers, including compiler front-end scheduling, driver-level instruction interception, and microarchitectural unit functional gating. Combined with the side-channel analysis results of Energy Density, the CMP 170HX exhibits a non-linear characteristic of "high Energy Density, low throughput" when executing restricted

instructions, which deviates from the efficiency curve of normal hardware under full load. This anomalous energy consumption characteristic indicates that the hardware compute units might not be physically damaged but are instead forced into an inefficient operating mode or dormant state by a certain logical gating mechanism, leading to severe waste of computational resources and degradation of energy efficiency.

Analyzing the complex composition of a single Streaming Multiprocessor (SM) within the GA100 microarchitecture—which includes Warp Schedulers, INT32/FP32 CUDA Cores, 3rd-generation Tensor Cores, Special Function Units (SFUs), and various memory units—it can be inferred that there are at least four layers of restriction in the CMP 170HX at the CUDA programming visible level. The first is the instruction emission restriction at the compiler (`nvcc`) level, used to explain the performance difference between intrinsic and standard library paths. The second is the performance restriction at the Floating-Point Unit (FPU) level, directly corresponding to the throughput collapse of FP32 `FMA`. The third is the restriction at the Special Function Unit (SFU) level, leading to low performance of instructions like `RCP` and `SQRT`. Finally, there is the restriction at the Tensor Core level, preventing `mma.sync` instructions from delivering their intended acceleration. This hierarchical restriction mechanism means that while the CMP 170HX possesses datacenter-grade potential in terms of physical hardware, it is strictly constrained to low-end application scenarios at the software scheduling level.

## 7.2 Empirical Refinement and Applicability Assessment of the Theoretical Model

Based on the experimental results presented above, this study reflects on and refines the theoretical model proposed in Chapter 3. First, the introduction of the Effective Instruction Execution Factor and the Effective Computational Execution Factor successfully deconstructs the macroscopic computational performance disparity into a utilization issue at the microarchitecture level, providing an effective mathematical tool for quantifying the performance gap in limited GPUs. However, the empirical analysis reveals that, given the current landscape where mainstream AI inference workloads are predominantly composed of compute-intensive operators, the numerical values of the Effective Computational Execution Factor show a high degree of consistency with those of the Effective Instruction Execution Factor for arithmetic-type instructions. Consequently, in practical engineering applications, these two can be regarded as equivalent metrics, thereby streamlining the evaluation process.

Secondly, at the operational level, the "Precision Ratio" in the theoretical model, which is calibrated based on non-limited devices, often proves difficult to obtain precise baseline data for.

The experiments indicate that, in the absence of official theoretical precision ratios as references, directly calculating the "Precision Execution Factor" using the actual measured computational performance of the target device under different precision formats is not only more convenient and efficient but also more intuitively reflects the device's relative performance tier at a specific precision. This normalization approach based on empirical measurement data reduces reliance on non-limited reference devices and enhances the model's independent applicability.

Finally, regarding the selection of energy consumption metrics, this study posits that "Computational Energy Density" is more closely aligned with real-world application scenarios and better aligns with the traditional definition of "energy efficiency" than "Instruction Energy Density." While Instruction Energy Density can assist in dissecting microarchitectural states, its unit (mW/Ginst/s) has a weaker correlation with macroscopic, system-level decision-making. In contrast, Computational Energy Density (W/Tflops) directly relates the power cost incurred per unit of computational throughput. It provides direct decision-making support for thermal design and power supply selection in edge computing and low-cost computational utilization scenarios. Therefore, in model applications oriented toward engineering practice, Computational Energy Density should be prioritized as the core metric for evaluating the benefits of instruction-avoidance strategies.

### **7.3 Interpretation of the Motivations Behind Possible Limiting Mechanisms**

For the complex limiting mechanisms exhibited by the CMP 170HX, their underlying motivations can be systematically explained from two dimensions: technical requirements and commercial strategy. In essence, the CMP 170HX is specialized hardware for cryptocurrency mining, such as Ethereum. The ETHash algorithm, a typical memory-intensive workload, relies on repeated access to a large Directed Acyclic Graph (DAG) dataset for its core computational logic. It involves extensive 32-bit unsigned integer (uint32) operations, as well as integer multiplication and XOR operations within the FNV hash mixing process. This algorithm has no need for single-precision, double-precision, or half-precision floating-point multiply-accumulate (FMA) operations whatsoever. Therefore, from a purely technical adaptation perspective, disabling all floating-point compute units, mathematical functions, and Tensor Cores would not impact its mining efficiency. On the contrary, such a move could potentially reduce chip power consumption and thermal output. However, the manufacturer did not adopt this extreme approach of physical hardware disabling. Instead, software-level restrictions were implemented while preserving the underlying hardware foundation.

The core driver behind this design choice lies in market segmentation and Stock Keeping Unit (SKU) control strategies for the product line. Considering that the GA100 core is fundamentally a high-end product intended for data centers, its packaging, testing, and manufacturing processes are extremely stringent, disallowing any post-factory physical rework. Consequently, implementing performance limitations through "functional gating" at the firmware level, such as via vBIOS, drivers, or microcode, became the most cost-effective and flexible solution. Through this strategy, the manufacturer ensured that chips produced from the same wafer could be precisely differentiated into the miner-oriented CMP series and the high-end, data center-focused A100 series. This effectively prevents lower-cost, restricted chips from entering the high-margin AI training market, thereby safeguarding market share and commercial interests for the premium product line.

#### **7.4 Inference on the State of Tensor Core**

Regarding the state of the Tensor Core, experimental data indicate that it is not completely physically disabled on the CMP 170HX but rather exists in a state of deep logical suppression. Compared to the physically heterogeneous design of the H100 architecture, where only a portion of SMs are equipped with rasterization capabilities, the `mma.sync` instruction on the 170HX remains executable but demonstrates an extremely low energy efficiency ratio.

This phenomenon is analogous to the instruction flow encountering a form of logical gating mechanism, resulting in significantly low overall utilization of the Tensor Core, while the scheduling queues appear fully occupied. This "soft-lock" limitation ensures that mining tasks—which do not utilize the Tensor Core—remain unaffected while completely preventing its potential value as an AI Accelerator. This verifies the manufacturer's strategic rigor in business operations.

#### **7.5 Discussion on the Generalization of the Proposed Model**

The analytical model proposed in this study possesses significant potential for generalization. Although the instruction-level analysis and precision ratio analysis rely on architectural transparency, the precision model based on power side-channel—including the Precision Execution Factor and Computational Energy Density—remains applicable even in black-box device scenarios.

This model can be effectively applied to domestic AI accelerators such as Huawei Ascend NPU, Moore Threads GPU, and Hygon DCU, as well as export-restricted models like the NVIDIA RTX

4090D and 5090D. Through this model, researchers and engineers can effectively characterize the instruction features, precision management strategies, and theoretical performance boundaries of these devices without relying on vendor-specific performance counters. This provides a universal theoretical foundation for computational optimization in constrained environments.

## 8 Operator-level Circumvention Strategies and Engineering Practice

Based on the results of the instruction-level testing described above, we will attempt to formulate circumvention strategies and apply this methodology in specific engineering practices. This circumvention strategy directly guides concrete application.

### 8.1 Overview of Circumvention Strategies

The preceding microarchitectural analysis at the instruction level indicates that the performance limitations of the CMP 170HX exhibit a pronounced coupling characteristic between instruction type, numerical precision, and execution unit. To bypass restricted instruction paths within the legitimate CUDA programming model, this paper proposes four categories of operator reconstruction strategies, forming a hierarchical performance recovery scheme:

#### 8.1.1 Instruction Decomposition Strategy

To address the throughput bottleneck of Fused Multiply-Add (FMA) instructions at FP32 precision, a single `fma.rn.f32` instruction is explicitly decomposed into a sequence of independent multiplication (`mul`) and addition (`add`) instructions. Although this transformation introduces double rounding error, for AI inference tasks represented by Transformer models, existing research indicates that such precision loss has a negligible impact on the numerical stability of the model's final output.

#### 8.1.2 Intrinsic Instruction Substitution Strategy

For special function instructions (such as `div`, `exp`, `pow`, and trigonometric functions) restricted by the standard library path, CUDA built-in functions (intrinsics, such as `__fdividef`, `__expf`) are used to replace standard math library calls. These intrinsic functions bypass potential instruction interception mechanisms at the driver or runtime library level, mapping directly to underlying SFU or ALU hardware instructions, thereby restoring the suppressed execution unit throughput capability.

#### 8.1.3 Algebraic Transformation Strategy

Based on numerical analysis theory, mathematically equivalent transformations are applied to

specific operations to circumvent restricted instructions. For example, for reciprocal (`rcp`) and square root (`sqrt`) operations, approximation algorithms based on Taylor expansion or polynomial approximation are employed; for mean calculation within reduction operations, explicit summation and scaling are used to replace implicit aggregation instructions. These transformations restructure the algebraic graph of the computation, avoiding performance-restricted instruction paths while maintaining numerical equivalence.

#### *8.1.4 Execution Unit Migration Strategy*

To address the logic gating limitations of Tensor Cores at FP16/BF16 precision, tensor core instructions such as `mma.sync` are completely avoided. Matrix multiply-accumulate operations are explicitly mapped to the SIMT execution path of CUDA Cores. This trade-off is made to obtain stable throughput performance and predictable energy efficiency characteristics.

## **8.2 Implementation of Custom Operators**

Based on the instruction dependency analysis of key operators in Transformer and Stable Diffusion from Section 5.1, this section selects three representative operators—Linear Transformation (Linear), Sigmoid Linear Unit (SiLU), and Root Mean Square Layer Normalization (RMSNorm)—to elaborate on the specific implementation mechanisms of the aforementioned mitigation strategies.

### *8.2.1 FMA Deconstruction of the Linear Transformation Operator (Linear/GEMM)*

The core computation of a standard linear transformation is matrix multiplication and accumulation, most commonly implemented as General Matrix Multiply (GEMM). Its fundamental operation relies on the Fused Multiply-Add (FMA) instruction, which performs the following calculation:

$$O_{ij} = FMA(A_{ik}, B_{kj}, C_{ij}) = A_{ik} \times B_{kj} + C_{ij}$$

At the CUDA level, this is typically compiled into the `fma.rn.f32` instruction. To circumvent the hardware limitations of the FP32 FMA pathway, this paper adopts an instruction splitting strategy, explicitly reconstructing it as a composite operation of independent multiplication and addition:

$$O_{ij} = FMA(A_{ik}, B_{kj}, C_{ij}) = ADD(MUL(A_{ik}, B_{kj}), C_{ij})$$

In engineering implementation, this is achieved by manually writing a CUDA kernel that explicitly calls the `fmul.rn.f32` and `fadd.rn.f32` instructions. This is coupled with optimizations such as register-level data reuse and shared memory tiling to ensure the computational bottleneck strictly resides at ALU throughput. If further optimization is required, considerations should extend to memory caching optimization and parallelism enhancement, among other optimization measures.

### 8.2.2 Intrinsic Instruction Optimization for the SiLU Activation Function

The SiLU operator is an important activation function. Mathematically, it is differentiable, has a lower bound but no upper bound, and its computation is more complex than ReLU. The SiLU operator's calculation is as follows:

$$SiLU(x) = \frac{x}{1 + \exp(-x)}$$

Its standard implementation relies on standard library calls for the exponential function and division (`expf`, `fdiv`). We employ an instruction substitution strategy, replacing the standard `exp` with the intrinsic function `_expf` and the standard division (`DIV`) with the intrinsic function `_fdividef`. That is:

$$SiLU(x) = _fdividef(x, 1 + _expf(-x))$$

### 8.2.3 Algebraic Reconstruction and Reciprocal Square Root Approximation for RMSNorm

RMSNorm is a layer normalization method that does not require mean centering, making computation more efficient. It is commonly used in Transformer architectures. Its mathematical expression is:

$$RMSNorm(x) = \frac{x}{\sqrt{mean(x^2) + c}} \odot g$$

The standard implementation relies on implicit mean reduction and the square root instruction

(**sqrt**). This paper implements a dual algebraic transformation:

First, the mean operation is explicitly deconstructed into parallel reduction summation and scalar multiplication:

$$\text{mean}(x^2) = \frac{1}{d} \sum_{i=1}^d x_i^2$$

Second, the reciprocal square root instruction **rsqrt** is utilized to replace **sqrt** and the subsequent division, thereby avoiding the performance limitations of the **sqrt** instruction:

$$\text{RMSNorm}(x) = x \cdot \text{rsqrt}\left(\frac{1}{d} \sum_{i=1}^d x_i^2\right) \odot g$$

### 8.3 Operator-Level Theoretical Performance Evaluation

Based on the constraint avoidance strategies proposed earlier, this study implements end-to-end custom CUDA kernels for three major categories of core operators, including basic computations, activation functions, and normalization functions. These are integrated into a PyTorch extension for incorporation into standard inference frameworks. To ensure objectivity and reproducibility of the evaluation, we construct standardized testing scripts to compare the computational throughput and energy consumption characteristics between the custom operators and PyTorch native operators in a controlled environment.

		Pytorch.FP32	Patch.FP32	Pytorch.FP16	Patch.FP16	Pytorch.BF16	Patch.BF16
Tanh [8192x8192]	-	2.300	0.513	2.054	0.445	2.055	0.444
ERF [8192x8192]	-	2.395	0.551	2.393	0.259	2.394	0.527
Linear [4096x4096] @ [4096x4096].T	-	351.112	24.318	22.339	6.750	23.876	6.723
BMM [Batch=16, 1024x1024 @ 1024x1024]	-	89.255	7.630	5.595	1.964	5.595	2.226
Conv2d [N=64, C=128, 128x128]	-	119.912	41.531	26.032	26.170	26.030	124.047
ConvTranspose2d [N=64, Cin=64, Cout=64, 64x64, Stride=2]	-	49.953	23.111	4.014	100.796	4.015	265.892
Attention [B=8, H=8, L=1024, D=128]	-	34.082	82.316	5.986	8.922	5.989	686.199
Embedding [Vocab=32000, Dim=1024, Input=(512, 1024)]	-	3.379	3.269	1.638	1.908	1.642	5.924
GELU [8192x8192]	-	2.349	0.487	2.394	0.210	2.393	0.280
Softmax [8192x8192]	-	5.120	0.682	5.127	0.465	5.134	0.232
SiLU [8192x8192]	-	3.924	0.412	3.418	0.209	3.418	0.208
Swish [8192x8192] (beta=10.0)	-	3.762	0.446	2.912	0.214	3.912	0.229
Mish [8192x8192]	-	7.161	0.412	7.156	0.212	7.157	0.211
Softplus [8192x8192] (beta=1.0, threshold=20.0)	-	6.823	0.552	6.820	0.218	6.819	0.446
Softsign [8192x8192]	-	2.542	0.417	2.267	0.374	2.268	0.375
GroupNorm [N=64, C=512, 128x128]	-	21.166	5.098	20.645	3.186	20.654	2.697
LayerNorm [In=64x128x1024, Norm=(1024, )]	-	0.355	0.092	0.353	0.085	0.355	0.066
RMSNorm [In=64x128x1024, Norm=(1024, )]	-	0.225	0.104	0.163	0.104	0.164	0.108

Figure 8-1: Custom Operator vs. Native Operator Throughput Comparison with Applied 170HX Optimization Strategy

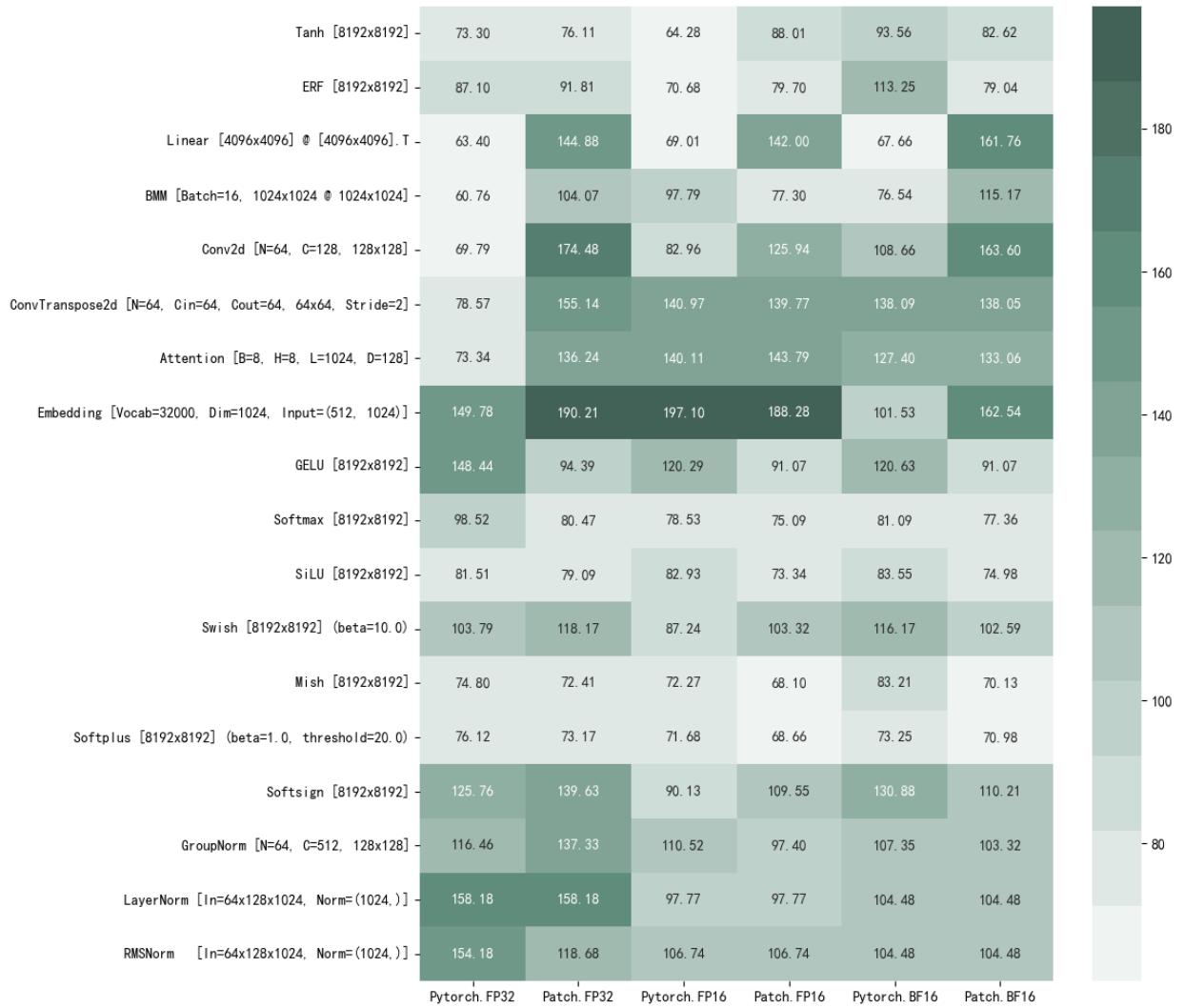


Figure 8-2: Custom Operator vs. Native Operator Power Consumption Comparison with Applied 170HX Optimization Strategy

The heatmap visualization results in Figures 8-1 and 8-2 clearly reveal the performance differentiation before and after optimization. Firstly, for computationally intensive operators (such as Linear, SiLU, and Softmax under FP32 precision), the custom operators demonstrate significant performance improvements compared to the official PyTorch implementations. This empirically validates the effectiveness of the avoidance strategies in overcoming instruction-level performance bottlenecks. Further analysis combining real-time power consumption comparisons reveals more in-depth microarchitectural behavioral characteristics: Some custom operators that achieved acceleration (e.g., FP32 Conv2D and Linear) exhibited significantly higher real-time power consumption during execution compared to their native counterparts. Considering the previous discussion on power side-channels, this phenomenon indicates that these operators successfully

activated more idle computational circuit resources on the CMP 170HX, thereby achieving the physical release of computational capability. Concurrently, another subset of operators (e.g., FP32 SiLU and RMSNorm) showed acceleration without a corresponding increase in power consumption, and in some cases, even a reduction. This suggests that the optimization path enhanced the hardware's energy efficiency ratio by reducing idle cycles in the instruction pipeline.

Furthermore, the experiments also exposed that certain operators (e.g., FP32 Attention and Embedding) did not achieve the expected acceleration, and their operational power consumption was comparable to or even higher than the native implementations. This indicates potential room for improvement in the current optimization implementations regarding engineering details such as cache spill handling, memory access patterns, or thread block scheduling, which may not yet fully exploit the hardware's potential.

#### **8.4 Performance in Operator Engineering Applications**

To validate the end-to-end effectiveness of the aforementioned instruction-level mitigation strategies in real-world AI inference tasks, this study selected three typical scenarios—image generation, Large Language Model (LLM) inference, and Diffusion Transformer (DiT) model inference—for systematic testing. The overall speedup data indicates that the proposed solution can effectively improve the response speed of the constrained GPU in practical applications while maintaining numerical precision.

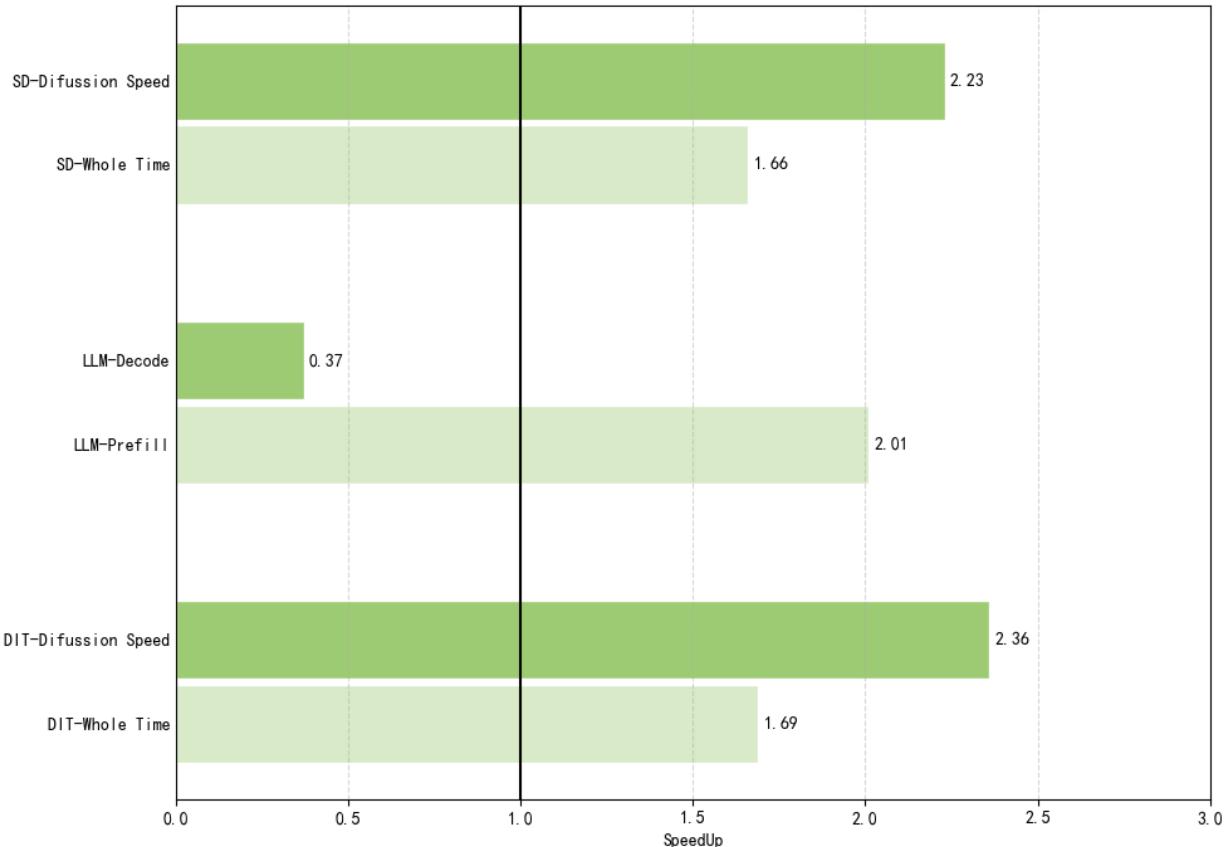


Figure 8-3: Speedup Ratio of Custom Operator with 170HX Optimization Strategy vs. Native Operator Across AI Inference Application Stages

#### *8.4.1 Stable Diffusion Image Generation Inference*

For the text-to-image task, a testing environment was constructed based on the ComfyUI framework. The operators involved in the complete inference pipeline—including text encoding, Diffusion denoising, and VAE decoding stages—such as Linear, Conv2d, GroupNorm, LayerNorm, and various activation functions, were replaced via custom extensions. The test employed the Stable Diffusion XL model with a resolution set to  $1024 \times 1024$ , using the Euler sampler for 35 iterations. The computationally intensive denoising stage utilized FP16 precision, while the text encoding and decoding stages maintained FP32 precision to ensure accuracy. Experimental results show that after applying the optimized operators, the total time for generating a single image was significantly reduced from 112.39 seconds to 67.62 seconds, validating the strategy's applicability in complex generative models.

#### *8.4.2 Large Language Model Inference*

For Transformer-based large language model inference, we selected the BF16 weights of the Qwen2.5-1.5B model and conducted replacement tests on key operators such as Linear, BMM, LayerNorm, and Softmax. The experimental results revealed phased performance differences: during the Prefill stage, the optimized operators brought significant throughput improvements; however, during the Decode stage, performance decreased by approximately 11.68 t/s. We speculate that this counterintuitive phenomenon might stem from inadequacies in the current custom operator's memory management mechanisms. This could introduce additional kernel launch overhead or cache miss penalties in the Decode stage, which is extremely sensitive to memory bandwidth and latency, thereby failing to effectively circumvent the performance bottleneck.

#### *8.4.3 DiT Image Diffusion Model Inference*

In the inference test of the DiT architecture, considering the 8GB VRAM limitation of the CMP 170HX, we introduced the ComfyUI-GGUF node and loaded the quantized Z-image-Turbo diffusion model using the Q6\_K quantization. Despite the constrained memory resources, we fully patched core operators such as linear layers, convolutional layers, and normalization layers throughout the entire pipeline. Under a lightweight configuration with a resolution of 1024×1024 and 8 steps using the Euler sampler, the entire process uniformly performed computations in BF16 precision. Test results show that the optimized inference time was reduced from 121.08 seconds to 71.62 seconds, demonstrating that this strategy can still provide stable and significant performance gains even in edge computing scenarios with limited VRAM and quantized models.

## 9 Deficiencies and Limitations of the Research

Through systematic instruction-level microbenchmarking and operator restructuring practices, this study successfully revealed the coupled limitation mechanisms among instructions, precision, and execution units within constrained GPUs, and verified the effectiveness of various mitigation strategies in end-to-end inference tasks. However, constrained by the research scope, technical complexity, and resource limitations, this work still has several areas requiring improvement, which are analyzed in detail below:

### 9.1 Partial Instructions Subject to Underlying Hardware Constraints, Difficult to Fully Overcome via Software Means

This study has systematically verified that the CMP 170HX exhibits hierarchical, multi-dimensional performance limitation mechanisms and has accordingly proposed four categories of mitigation strategies: instruction splitting, intrinsic substitution, algebraic transformation, and execution unit migration. However, experimental data indicate that the limitations affecting certain critical instructions (such as `FMA` instructions under FP32 precision and the `mma.sync` tensor core instructions under FP16/BF16 precision) are notably inherent to the underlying hardware and persistent. As analyzed in Chapter 7, regardless of whether the standard library invocation path or the intrinsic compilation path is used, the Effective Execution Factor for these instructions consistently remains at extremely low levels (below 0.05), accompanied by anomalously high Computational Energy Density (e.g., 4.29 W/TFLOPS for FP16 `mma.sync`, approximately 13 times that of the A800). This phenomenon strongly suggests that the limiting mechanism likely resides at the microcode or hardware feature gating level, rather than being confined solely to the driver or compiler front-end.

Such underlying constraints pose fundamental challenges to performance recovery: On one hand, FP32 FMA, as the core arithmetic unit of the IEEE 754 standard, suffers from performance collapse, directly preventing traditional FP32-intensive tasks (such as non-linear transformations in Transformer inference) from achieving even 50% of the theoretical computational power. On the other hand, the logical inhibition of the Tensor Core path prevents low-precision matrix operations from leveraging the 4x throughput gain intended by the architecture design, forcing the system to fall back to the SIMT execution mode of CUDA Cores. Although this study achieved approximately a 15-fold performance recovery (from 0.39 to 6.25 TFLOPS) for FP32 linear operations through instruction splitting strategies, the overall computational capability remains

unable to exceed 50% of the theoretical peak of the GA100 architecture due to the hardware design ceiling. Future research needs to combine firmware reverse engineering and hardware signal analysis to further elucidate the specific implementation mechanisms of these underlying constraints and their potential bypass boundaries.

## 9.2 High Engineering Cost of Manual Optimization for Complex Operators (e.g., Attention Mechanism)

Based on the instruction-level analysis results, this study implemented custom CUDA kernels for fundamental operators such as Linear, SiLU, and RMSNorm in Chapter 8, and validated their effectiveness in typical tasks including SDXL, LLM, and DiT. However, for highly complex composite operators such as Attention, their implementation faces significant engineering challenges. The Attention mechanism involves multiple computational stages—QKV projection, scaled dot-product, Softmax normalization, and output projection—characterized by tight data dependencies and coupling of memory access patterns between stages. To circumvent constrained instruction paths, fine-grained instruction-level reconstruction is required for each sub-operation while maintaining the numerical equivalence of the computational graph and the locality of memory accesses.

Although large language model-assisted CUDA kernel generation techniques (e.g., KernelBench[18]) have significantly lowered the development barrier for fundamental operators in recent years, such tools remain insufficient when handling advanced optimization techniques like multi-stage dependency management, shared memory tiling strategy optimization, and warp-level synchronization. This study attempted a full-chain reconstruction of the Attention operator. However, constrained by the team's limited accumulated experience in GPU microarchitecture optimization and human resource limitations, competitive performance was not achieved (as shown in Section 8.3, the optimized FP32 Attention operator exhibited no performance improvement). This limitation highlights the gap between "theoretical feasibility" and "practical realizability" in constrained hardware optimization. Future work could explore automatic instruction replacement techniques based on compiler intermediate representation (IR) or construct domain-specific languages (DSL) to reduce the cognitive load of complex operator reconstruction.

## 9.3 Kernel Launch Overhead Introduced by Fine-grained Operator Replacement

When implementing operator-level avoidance strategies in an end-to-end inference pipeline, it is inevitable to face the trade-off of Kernel Launch overhead. In modern GPU architectures, each

Kernel Launch involves fixed overhead such as host-device data transfer, thread block scheduling, register allocation, and Streaming Multiprocessor (SM) context switching, typically on the order of several microseconds. When the inference task involves a large number of short-duration, small operators (e.g., LayerNorm, activation functions), frequent Kernel Launches can significantly dilute the performance gains from compute-intensive operations.

The performance degradation observed in the decoding phase of LLMs in this study (Section 8.4) is partially attributed to this issue. In the token-by-token autoregressive generation process, each decoding iteration involves dozens of small operator calls. The independent launching of custom operators leads to an increased proportion of scheduling overhead. Experimental data show that during the Decode phase of the Qwen2.5-1.5B model, the optimized throughput actually decreased to approximately 11.68 tokens/s, forming a stark contrast to the significant improvement observed in the Prefill phase. This phenomenon confirms the importance of the "operator fusion" strategy discussed in Section 2.2.2 – by fusing multiple operators with data dependencies into a single kernel, the overhead associated with writing intermediate results back to global memory and subsequently reading them back can be effectively eliminated. However, constrained by the dynamic computational graph characteristics of frameworks like PyTorch, achieving automatic cross-operator fusion faces the contradiction between graph optimization complexity and framework intrusiveness. Future research could leverage static graph techniques such as TorchScript or FX Tracing to develop specialized operator fusion compilers targeting constrained hardware.

#### 9.4 Practical Challenges in Integrating Patches into Large-Scale Deep Learning Frameworks

Although modern deep learning frameworks like PyTorch provide mechanisms for extending custom operators (e.g., `torch.autograd.Function`) and runtime patching capabilities (commonly known as "monkey patching"), they still face multiple challenges in real-world deployment.

First, mainstream inference frameworks (such as vLLM, TensorRT-LLM), in pursuit of peak performance, heavily employ advanced techniques like operator fusion, kernel specialization, and memory layout optimization. Their internal computational graph structures are highly non-standardized, making it difficult for external patches to precisely locate and replace the target computational path.

Second, frequent framework version iterations lead to significant changes in low-level APIs and

internal implementation details, resulting in high maintenance costs for custom operator compatibility. For instance, when integrating SDXL-optimized operators into the ComfyUI framework in this study, adaptations were required for the node executors of different versions, significantly increasing engineering complexity.

More fundamentally, the "monkey patching" method is inherently an intrusive modification that violates software engineering principles of encapsulation and maintainability. When multiple optimization strategies (e.g., quantization, operator fusion, custom kernels) are applied simultaneously to the same framework, unexpected interaction effects and debugging difficulties can easily arise. The ideal solution should be to build a Hardware Abstraction Layer (HAL) decoupled from the framework, enabling transparent acceleration for constrained hardware through standardized interfaces.

However, developing such infrastructure requires deep involvement in the core architecture of the framework, which exceeds the resource capabilities of a single research project. Future work could explore collaboration with open-source inference framework communities to promote the design of standardized interfaces for constrained hardware support.

## 9.5 Lack of In-Depth Instruction Behavior Analysis at the PTX/SASS Level

The instruction-level analysis in this study primarily relies on control and observation at the CUDA C++ source code level, without delving deeper into the PTX (Parallel Thread Execution) intermediate representation or the SASS (Streaming ASSEMBly) native device instruction level. This limitation may lead to misinterpretations of the restriction mechanisms: the compiler backend (e.g., NVCC) might rewrite or replace specific instruction sequences during the PTX generation phase, causing the "instruction evasion" at the source code level to be remapped to restricted paths at the PTX/SASS level. For instance, the CUDA compiler might re-fuse an explicit `mul` + `add` instruction sequence into an `fma` instruction during the PTX optimization stage, thereby circumventing the user-intended evasion strategy.

Furthermore, the instruction scheduling and execution unit allocation on NVIDIA GPUs heavily depend on microarchitectural details at the SASS level (e.g., warp scheduler policies, functional unit contention mechanisms). Relying solely on high-level instruction throughput measurements makes it difficult to precisely infer the actual allocation state of hardware resources. Combining disassembly analysis of the generated SASS code using NVIDIA's `cuobjdump` tool with instruction-level sampling via performance counters (e.g., the `inst_executed` event in

`nvprof`) would help more accurately pinpoint the level (compiler/driver/microcode/hardware) at which restrictions occur. Limited by the research timeline and forbidden toolchain, this work did not conduct such in-depth analysis, which constitutes an important direction for future research. In particular, for the restriction mechanisms on the Tensor Core path, PTX/SASS-level analysis might reveal the specific reasons why `mma.sync` instructions are redirected to inefficient execution paths, providing a basis for designing more robust evasion strategies.

In summary, while this study has achieved preliminary results in instruction-level modeling and operator-level optimization for restricted GPUs, significant limitations remain in areas such as breaking through underlying restrictions, engineering complex operators, the feasibility of system integration, and the depth of analysis. These shortcomings not only reflect the inherent challenges in the field of constrained hardware optimization but also point out potential breakthrough directions for subsequent research. Future work will focus on compiler-assisted automatic evasion strategy generation, the design of cross-operator fusion optimization frameworks, and deep analysis of restriction mechanisms combined with hardware reverse engineering, with the aim of constructing a more comprehensive technical system for reusing constrained computing power.

## 10 Conclusion

### 10.1 Core Finding: Constraints are Instruction-Level and Hierarchical

Through a systematic instruction-level microarchitecture analysis of the NVIDIA CMP 170HX, a representative constrained AI accelerator, this study confirms that the performance suppression mechanisms in commercial constrained GPUs exhibit significant instruction-level granularity and multi-level coupling characteristics. Specifically, performance limitations do not stem from global frequency throttling or power limit constraints but manifest as a selective suppression strategy targeting specific combinations of instruction types, numerical precisions, and execution units. Experimental data demonstrates that the CMP 170HX imposes near-total throughput blocking on Fused Multiply-Add (FMA) instructions at FP32 precision (Effective Execution Factor as low as 0.06), while basic arithmetic instructions (ADD/MUL) maintain near-theoretical peak execution efficiency. At FP16/BF16 precision, Tensor Core-based matrix multiply-accumulate instructions (mma.sync) are similarly subjected to systematic suppression, whereas arithmetic instructions via the CUDA Core pathway remain largely unaffected. This tripartite coupled constraint mode of "instruction type–numerical precision–execution unit" reveals the engineering practice of product line segmentation through multi-level feature gating mechanisms implemented across several software/firmware layers—such as the compiler front-end scheduling layer, driver instruction interception layer, and microarchitecture unit feature gating layer—rather than employing irreversible hardware masking methods like physical fuses. Crucially, power side-channel analysis reveals that constrained instructions exhibit a non-linear characteristic of "high Computational Energy Density, low throughput" during execution. This indicates that the restricted computational units are not physically defective but are instead forced by a logical gating mechanism into inefficient operational states or partial sleep modes. This strategy effectively blocks the potential of the device as a general-purpose AI accelerator while preserving its performance for targeted applications, such as cryptocurrency mining.

### 10.2 Practical Guidelines for CUDA Programming on the 170HX

Based on the empirical analysis of the instruction-level limiting mechanisms above, this study summarizes CUDA programming optimization guidelines for constrained accelerators such as the CMP 170HX, providing actionable guidance for engineering practice:

- (1) Precision Path Selection Principle: In compute-intensive tasks, preferentially execute linear

operations using FP16/BF16 precision to circumvent the performance bottleneck of the FP32 FMA instruction path. For nonlinear operations (e.g., activation functions, normalization), maintain FP32 precision to ensure numerical stability, as their fundamental arithmetic instructions are not significantly restricted.

- (2) Instruction-Level Circumvention Strategy: To address FP32 FMA restrictions, employ an instruction decomposition strategy (explicitly breaking down `fma.rn.f32` into separate sequences of `fmul` and `fadd` instructions). For special function instructions (e.g., `exp`, `div`), prioritize the use of CUDA intrinsic functions (e.g., `__expf`, `__fdividef`) over standard math library calls to bypass potential driver-level instruction interception.
- (3) Execution Unit Migration Principle: In low-precision matrix operations, actively avoid the Tensor Core-dependent path (e.g., `mma.sync` instructions). Explicitly map the computational workload to the SIMT execution model of the CUDA Cores to achieve stable and predictable throughput performance.

### **10.3 Power Side-Channel Analysis: A New Paradigm for Inferring Hardware Behavior in Constrained Environments**

This study innovatively employs power consumption measurement as a side-channel analysis tool for inferring the internal execution state of constrained hardware, establishing a coupled analysis framework of "Instruction Throughput–Energy Density." In scenarios where traditional performance counters are unavailable or blocked by vendor policies, this method effectively identifies the actual activity level and constrained state of computational units by monitoring the relationship between real-time power consumption and instruction throughput under saturated computational load: normally functioning execution units exhibit a linear characteristic of "high throughput–low energy density," while units constrained by logic gating display abnormal nonlinear behavior of "low throughput–high energy density."

This analytical paradigm overcomes reliance on vendor-specific performance monitoring tools and provides a universal methodology for inferring microarchitectural characteristics in black-box or semi-black-box hardware devices. Its value extends not only to performance evaluation of constrained GPUs but also to cross-platform performance modeling of domestic AI Accelerators (such as Ascend NPU and Hygon DCU), compliance verification of export-controlled devices, and dynamic scheduling decisions for heterogeneous hardware resources in edge computing scenarios.

In the future, this method is expected to integrate with machine learning technologies to develop an automated hardware constraint identification system based on power consumption time-series features, further enhancing resource utilization efficiency in constrained computing environments.

#### **10.4 Environmental Value and Economic Significance: Promoting the Sustainable Reuse of Low-Cost Computing Power**

The practical outcomes of this research highlight the unique value of restricted GPUs within the frameworks of green computing and the circular economy. With mainstream cryptocurrencies like Ethereum transitioning to Proof-of-Stake (PoS) mechanisms, a significant number of mining-specific GPUs (such as the CMP series) designed for Proof-of-Work (PoW) face market obsolescence, creating a potential risk of electronic waste. This study confirms that through instruction-level avoidance strategies and operator reconstruction techniques, the CMP 170HX can achieve end-to-end performance improvements of up to approximately  $2\times$  in typical AI tasks such as Transformer inference and Stable Diffusion image generation. This enables it to exhibit energy efficiency ratios approaching those of data center-grade GPUs in edge inference scenarios constrained by memory bandwidth. Combined with its significantly lower second-hand market price compared to new products, such devices possess prominent economic and environmental value in low-cost application scenarios such as community edge computing and lightweight AI service deployment. On one hand, by extending the hardware service life (from 1-3 years to over 5 years), approximately 60% of the demand for new hardware procurement can be reduced, significantly lowering lifecycle resource consumption and carbon emissions. On the other hand, it provides high-cost-performance AI computing power access solutions for resource-constrained regions, promoting technological inclusivity and digital equity. The analytical framework and optimization methods proposed in this research provide a technical foundation for establishing a sustainable computing power utilization paradigm that integrates "restricted hardware, software adaptation, and scenario-specific customization" in a trinity. This aligns with global trends in green AI and the principles of the circular economy, holding positive practical significance for promoting the low-carbon transition of the computing industry.

## Acknowledgments

This research spanned a period of one year. Initially (early 2025), I purchased this 170HX graphics card purely out of admiration for the GA100 architecture and conducted some simple experiments and data collection. However, due to my limited understanding of the underlying mechanisms of AI inference at the time, my efforts amounted to only superficial reproductions of existing work. Moreover, having just resigned from my job, I was physically and mentally exhausted, and under considerable pressure from my family, so that early attempt never progressed into thorough analysis.

Over the following six months, I continued to tinker with research in AI applications and occasionally wrote articles for public accounts. I soon realized, however, that I was not well-suited for this kind of work—most of what I produced merely replicated efforts already made by others, and even those reproductions were not always successful. It wasn't until one day last November, when I decided to test the newly released Gemini 3 Pro model and revisited my earlier paper, that a path related to "cuBLAS/cuTLAS" was brought to my attention. While experimenting with implementing GPU-Burn both with and without cuBLAS/cuTLAS, I noticed CUDA programming instructions—a more comprehensive and systematic area—and decided to shift my focus toward studying it. I completed all the experiments around mid-December, but unfortunately, my left eye suddenly developed severe floaters, forcing me to take a full month of rest before I could begin organizing the data and writing this paper.

Compared to my previous work, this paper received little external assistance. Even the driver installation process, which was unfamiliar to me at first, was learned entirely through trial and error. Nevertheless, I would like to extend my sincere gratitude to those who consistently offered me positive feedback and encouragement throughout this research journey. They are: "祈泪" "世界," and "昔音幻离"

I am also deeply grateful to my father for financially supporting my equipment purchases. In August last year, I acquired a dual-socket EPYC 7532 server with the intention of experimenting with CPU-GPU hybrid deployment, only to find the results unsatisfactory due to the low memory frequency (a mere 2133 MHz). That machine now serves as my workstation. The aforementioned 170HX graphics card was also funded by him (I actually sold the original card in June last year and repurchased a replacement in November).

Lastly, I thank my mother and grandmother for taking care of my daily life.

## References

1. Lu Weizheng, Zhang Feng, He Yinxuan, Chen Yueguo, Zhai Jidong, Du Xiaoyong, Performance Evaluation and Optimization of Huawei Ascend AI Processors, Chinese Journal of Computers, Vol. 45, No. 8, Aug. 2022.
2. Qiu Hongfei, Li Xianxu, Huang Chunguang, Performance Evaluation of GPU Servers in Cloud Infrastructure, Mobile Communications, 2018, 42(7): 1-6.
3. Wei Jizheng, Research on GPGPU Performance Optimization in Multi-kernel Environment Based on Thread Scheduling, East China University of Technology, June 2024.
4. Xing Kangwei, Exploration of Cryptocurrency Mining-Specific GPUs in AI Applications: A Case Study of CMP 170HX, arXiv, 2025.
5. Lu Lin, The Impact of Cryptocurrency Regulatory Policies on Mining Activities and the Semiconductor Industry, Guangxi University, May 2024.
6. Matt Wuebbling, GeForce Is Made for Gaming, CMP Is Made to Mine, February 18, 2021, <https://blogs.nvidia.com/blog/geforce-cmp/>.
7. NVIDIA, A Crypto Mining GPU for Professionals, 2021, <https://www.nvidia.com/en-us/cmp/>.
8. NVIDIA, NVIDIA Products \_ AI, Data Center, Cloud, GeForce Laptops, and More, 2026, <https://www.nvidia.cn/products/>.
9. Zhang Jun, Research on Performance Optimization of General Purpose Graphics Processing Unit based on Thread Scheduling, Wuhan University, May 2016.
10. NVIDIA, NVIDIA Ampere GA102 GPU Architecture, 2020.
11. Li Yanli, Wu Qiang, Advanced Lithography Process Development Methods and Workflows in Modern Integrated Circuit Fabs, Tsinghua University Press, ISBN 9787302664185, Sep. 2024.
12. Wang Yaohua, Guo Yang, The Review of State-of-the-Art Processor Architecture for High Performance Computing, Computer Engineering & Science, Vol. 42, No. 10, Oct. 2020.
13. NVIDIA, NVIDIA A100 Tensor Core GPU Architecture, 2020.
14. Zhang Jianding, Chen Genlang, Ming Zongyu, A CUDA Kernel Time Prediction Model Based on Instruction Pipeline, Software Engineering, Vol. 27, No. 10, Oct. 2024.
15. Lei Xinli, Research on Parallel Solution Method of Small and Medium Sized Linear Equations Based on GPU.
16. Hamdy Abdelkhalik, Yehia Arafa, Nandakishore Santhi, Abdel-Hameed Badawy, Demystifying the Nvidia Ampere Architecture through Microbenchmarking and Instruction-level Analysis, Aug. 2022.
17. Md Aamir Raihan, Negar Goli, and Tor M. Aamodt, Modeling Deep Learning Accelerator Enabled GPUs, Electrical and Computer Engineering, University of British Columbia, Feb.

2019.

18. Anne Ouyang, et al., KernelBench: Can LLMs Write Efficient GPU Kernels, Stanford University, Princeton University, Feb. 2025.
19. Sunpyo Hong, Hyesoon Kim, An Integrated GPU Power and Performance Model, ISCA '10: Proceedings of the 37th Annual International Symposium on Computer Architecture, June 2010.
20. Weile Luo, Ruibo Fan, Zeyu Li, Dayou Du, Qiang Wang, Xiaowen Chu, Benchmarking and Dissecting the Nvidia Hopper GPU Architecture, Feb. 2024.
21. Sophia Falk, et al., From FLOPs to Footprints: The Resource Cost of Artificial Intelligence, Bonn University, Dec. 2025.
22. Mingjin Zhang, EdgeShard: Efficient LLM Inference via Collaborative Edge Computing, IEEE Internet of Things Journal, Vol. 12, No. 10, 15 May 2025.
23. Lu Lu, Zhao Rong, Liang Zhihong, Suo Siliang, Design and Optimization of Small Batch Matrix Multiplication Based on Matrix Core, Journal of South China University of Technology (Natural Science Edition), Sep. 2025.
24. Guido Juckeland, et al., SPEC ACCEL: A Standard Application Suite for Measuring Hardware Accelerator Performance.
25. NVIDIA cuBLAS Library, Tensor Core Operations.
26. Md Aamir Raihan, Negar Goli, and Tor M. Aamodt, Modeling Deep Learning Accelerator Enabled GPUs, University of British Columbia, Feb. 2019.
27. NVIDIA, NVIDIA A800 Tensor Core GPU, 2022.
28. Ainur R. Zamanov, Vladimir V. Erokhin, Pavel S. Fedotov, ASIC-resistant Hash Functions, Department of Computer Systems of Technologies, Department of Industrial Economics and Management, National Research Nuclear University MEPhI (Moscow Engineering Physics Institute).
29. Chen Daokun, Yang Chao, Liu Fangfang, Ma Wenjing, Parallel Structured Sparse Triangular Solver for GPU Platform.
30. consensus/ethash/algorithm.go - GitLab.

## Appendix

### Appendix 1 Experimental Environment Configuration and System Parameters

To ensure the rigor and reproducibility of the instruction-level microbenchmarking and operator-level performance evaluation in this study, we constructed experimental platforms in both local and cloud-based high-performance computing (HPC) benchmarking environments, implementing strict consistency controls for the software stack and compilation parameters.

#### *1.1 Local Environment Evaluation Platform*

This platform was primarily used for deep instruction-level profiling of a restricted GPU (NVIDIA CMP 170HX) and validation of optimized end-to-end inference tasks.

Central Processing Unit (CPU): AMD Ryzen 7 7840HS, with integrated Radeon 780M graphics, configured as an 8-core, 16-thread host CPU responsible for instruction dispatch and task scheduling.

Target Accelerator (Target GPU): NVIDIA CMP 170HX (based on the GA100 architecture), equipped with 8GB of HBM2e memory. This device served as the primary restricted unit under test (UUT) in this research.

System Memory: 64GB of DDR5 RAM, ensuring no system-level memory bottlenecks during data preprocessing and model loading phases.

Operating System: Debian 12 (codenamed Bookworm), with Kernel version 6.1 LTS, providing a stable Linux-based runtime environment.

Driver and Runtime Environment: Proprietary NVIDIA driver version 590.44, paired with CUDA Toolkit 12.8. A relatively recent driver version was chosen to ensure full support for Ampere architecture features and to exclude performance anomalies caused by driver bugs.

#### *1.2 Cloud Baseline Environment*

To compare the performance of restricted GPUs against a non-restricted data center baseline, we deployed a cloud-based HPC environment.

Central Processing Unit (CPU): Intel Xeon Gold 6138, configured as 20 cores, 40 threads (limited to 8 threads), providing ample computational support.

Baseline Accelerator (Baseline GPU): NVIDIA A800 80GB. This device is based on the full GA100 architecture. Apart from interconnect bandwidth, its compute units and architectural features are identical to those of the CMP 170HX, but it lacks any functional restrictions, making it an ideal unrestricted control group.

System Memory: 32GB of ECC DDR4 RAM.

Operating System: Ubuntu 22.04 LTS, offering mature compatibility within the cloud computing ecosystem.

Driver and Runtime Environment: NVIDIA Data Center driver version 572.44 and CUDA Toolkit 12.8 were used to ensure version consistency in the compiler backend with the local environment, thereby eliminating measurement bias introduced by toolchain differences.

### *1.3 Compilation Environment Configuration and Optimization Parameters*

For all experiments in this study, kernel compilation strictly adhered to the following parameter configurations to generate machine code optimized for the target architecture:

Optimization Level: **-O3**. This enables the highest level of compiler optimizations, including loop unrolling, vectorization, and intrinsic function substitution, to maximize single-thread execution efficiency and reduce instruction dispatch overhead.

Architecture Target: **-arch=sm\_80**. This specifies the compiler to generate PTX intermediate representation and SASS assembly code optimized for the NVIDIA Ampere architecture (compute capability 8.0). This parameter ensures the code can correctly invoke architecture-specific instruction sets (e.g., Tensor Core operations and third-generation Tensor Core instructions).

## **Appendix 2 Analysis of Temperature Drift Characteristics of Static Power Consumption in GA100 Architecture**

During the process of constructing the instruction-level energy density model, we observed that the GA100 core exhibits significant coupling characteristics between static power consumption and temperature. This physical phenomenon impacts the accuracy of the "Computational Energy Density" metric proposed in this study, and thus, a separate experimental record and analysis are provided here.

### *2.1 Experimental Observation Data*

By controlling the operational state of the cooling system of the CMP 170HX to achieve thermal equilibrium steady-state at different core temperatures, we recorded the power consumption data in an idle state, as shown in Table EX-1 below:

Idle Temperature (°C)	20	35	53	71
Idle Power (W)	31	40	49	60

Table EX-1: GA100 Temperature-Power Table at Idle State

### 2.2 Explanation of Physical Mechanism

The aforementioned phenomenon can be attributed to the transistor leakage effects in deep submicron process technologies. The GA100 core is manufactured using TSMC's 7nm process, with a chip area as large as 826 mm<sup>2</sup>. As the process node shrinks, the gate oxide thickness decreases, leading to extreme sensitivity of subthreshold leakage and gate leakage to temperature variations.

Specifically, according to semiconductor physics models, the reverse saturation leakage current of a transistor exhibits an exponential relationship with temperature. When the GPU core temperature rises from 20°C to 71°C, the static power consumption (primarily composed of leakage) nearly doubles. This high leakage baseline constitutes a non-negligible component of the GPU's total power consumption.

### 2.3 Impact on Energy Efficiency Modeling

The core model of this study relies on calculating the "Instruction Energy Density," i.e.,  $\lambda_{inst} = \frac{W_{inst} - W_s}{E_{inst}} \times 1000$ . The significant fluctuation of  $W_s$  with temperature will introduce errors in the calculation of the dynamic power consumption  $W_{inst}$ .

Therefore, in high-load experiments, although a large number of computing units are activated, causing voltage to increase and thereby intensifying leakage effects (elevating the baseline), we must account for the nonlinear contribution of temperature to the baseline power consumption during data processing. This also explains why, during prolonged stress tests, the energy efficiency ratio of the device often degrades as the temperature approaches the Thermal Design Power (TDP)

threshold. This degradation is not merely a result of increased dynamic power consumption but is also a physical manifestation of the sharp rise in static leakage.

### **Appendix 3 Multi-Platform Operator-Level Performance Comparative Analysis**

To quantify the actual gains of the custom operator optimization strategy proposed in this study on the restricted GPU and situate it within a broader hardware ecosystem, we conducted a systematic horizontal comparison between the optimized performance on the NVIDIA CMP 170HX and baseline, unrestricted devices (A800, RTX 3080).

This section selects key operators defined in Chapter 3 (including Linear/GEMM, Conv2D, Attention, activation functions, and normalization layers) and tests them on the following platforms:

Experimental Group: NVIDIA CMP 170HX (equipped with the custom CUDA operator library optimized in this study).

Control Group 1: NVIDIA A800 (equipped with the PyTorch native operator library utilizing cuBLAS/cuDNN, representing the performance ceiling for data center-grade hardware).

Control Group 2: NVIDIA GeForce RTX 3080 10GB (equipped with the PyTorch native operator library utilizing cuBLAS/cuDNN, representing the performance of peer consumer-grade products).

All tests uniformly adopted an independent strategy for three precision formats: FP32, FP16, and BF16.

Figure EX-1 below presents the relative performance of different operators across the three device types (the values represent the execution time for the corresponding task):

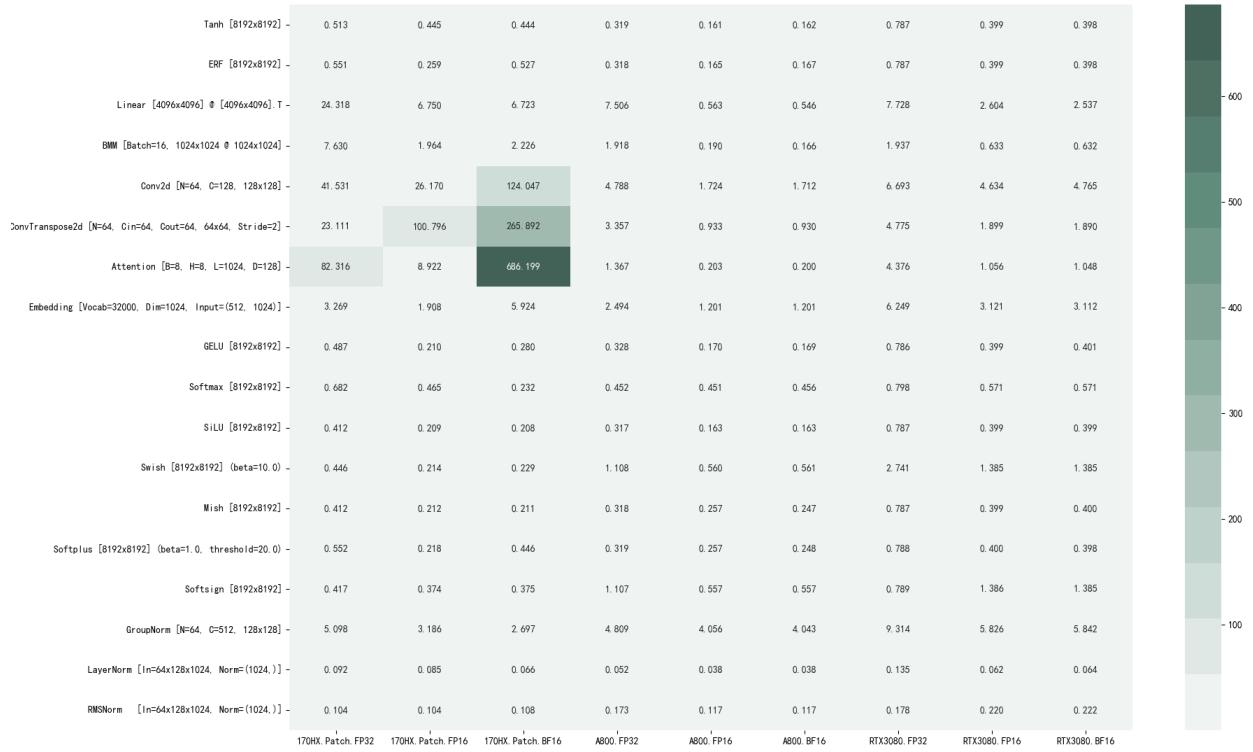


Figure EX-1: Custom Operator (170HX) vs. Native Operator (A800/RTX3080) Throughput Comparison

## Appendix 4 Numerical Precision Preservation Verification of Custom Operators

To ensure that the instruction-level bypass strategies and operator reconstruction schemes proposed in this study enhance performance without introducing significant numerical errors or degrading model output quality, we conducted rigorous numerical precision comparison experiments for two typical tasks: image generation and natural language processing.

### 4.1 Objective Quality Assessment for Image Generation Tasks

In the field of computer vision, the Structural Similarity Index (SSIM) and Peak Signal-to-Noise Ratio (PSNR) are objective standards for evaluating image reconstruction quality. SSIM focuses on measuring the perceptual similarity in structure, luminance, and contrast between images, while PSNR quantifies the ratio between the maximum possible power of a signal and the power of corrupting noise, based on Mean Squared Error (MSE).

We established a controlled test environment based on the ComfyUI framework. To eliminate

randomness and ensure the validity of the comparison, the following variables were strictly kept consistent during inference using the native operators and the optimized custom operators:

Model Configuration: The same pre-trained diffusion model weights were used.

Generation Parameters: Identical resolution, prompt text, random seed, classifier-free guidance scale, and sampling steps were set.

Execution Environment: Inference was performed on the same CMP 170HX hardware platform. Through a pixel-by-pixel comparative analysis of the generated images, the following metrics were obtained:

		
<b>SSIM</b>	0.8783	
<b>PSNR</b>	25.9602 dB	

Table EX-2: Result Error Table for Custom Operator in Stable Diffusion Inference

The experimental results indicate that the images generated using the custom operators maintain a very high structural similarity to the baseline images. An SSIM value approaching 0.88 and a PSNR exceeding 25 dB typically signify differences imperceptible to the human eye in image generation tasks. This data confirms that, despite employing strategies such as instruction splitting (e.g., FMA deconstruction) in FP32 linear operators and algebraic equivalence in nonlinear transformations, the introduced rounding errors remain within the model's tolerable numerical range after cumulative propagation, causing no substantial negative impact on generation quality.

#### 4.2 Perplexity Analysis for Language Model Inference

For Large Language Models (LLMs), Perplexity (PPL) is a core metric for measuring the accuracy of probability distribution predictions. It is essentially the geometric mean of the inverse predicted

probabilities of the test set by the model. A lower value indicates that the model is less "surprised" by the outcomes, signifying stronger language modeling capability.

We selected the text "Annie On My Mind" as the standard test corpus and conducted inference tests using the Transformer-based Qwen2.5-1.5B model under BF16 precision. The experiments compared the model's inference performance between two states: without optimization enabled (Patch OFF) and with custom operator optimization enabled (Patch ON).

测试状态	Perplexity (PPL)
Patch OFF (Baseline)	10.9782
Patch ON (Optimized)	11.0004

Table EX-3: Result Error Table for Custom Operator in LLM Inference

The test data shows that the optimized PPL only increased slightly from 10.9782 to 11.0004, representing a very minor relative error (<0.2%). This negligible fluctuation falls within the conventional error margin of LLM inference and may be attributed to changes in floating-point operation order. This result strongly demonstrates that the algebraic transformations and intrinsic instruction substitution strategies in this study do not compromise the numerical stability within the Transformer model while significantly boosting computational throughput, thereby preserving the coherence of semantic understanding and generation.

## Appendix 5 Analysis of Limitations in Custom Operator Design and Attribution of Performance Differences

Although this study achieved significant performance improvements on the constrained GPU, in engineering practice, some custom operators failed to achieve the expected acceleration and even exhibited performance regression under specific scenarios. Through in-depth micro-architecture profiling and performance analysis, we attribute these phenomena primarily to technical challenges at the following two levels.

### 5.1 Insufficient Operator-Level Optimization Depth and Resource Allocation Trade-offs

#### a. Lack of Fine-Grained Operator Fusion

As discussed in Section 8.3 of the main text, the current implementation of this study primarily focuses on instruction-level restructuring within individual operators and has not yet delved into cross-operator graph-level optimization. In stages such as LLM decoding, the inference pipeline

consists of a large number of fine-grained operators connected in series. Frequent Kernel Launch overhead and multiple rounds of memory read/writes (HBM access) become new performance bottlenecks. Particularly for operators involving complex dependencies, such as the Attention mechanism, the lack of operator fusion prevents full utilization of on-chip shared memory for intermediate data caching. This limits the benefits of optimization strategies during the Decode stage and can even lead to performance degradation due to increased scheduling overhead.

### **b. Priority Adjustment of BF16 Optimization Strategies**

In the initial phase of experimental design, the research focus was primarily on verifying the performance recovery potential of the constrained GPU under FP32 (via FMA circumvention) and FP16 (CUDA Core path). Therefore, for BF16 precision operator implementations, the initial goal focused mainly on functional correctness and interface compatibility, rather than extreme performance tuning. This resulted in some BF16 kernels implementing only basic computational logic, lacking deep vectorization and pipeline optimization tailored for the memory access characteristics and instruction scheduling of the BF16 data type. Consequently, they failed to fully unleash the hardware potential under this precision mode.

## *5.2 Cache Locality Failure Due to Micro-Architectural Differences*

### **a. Structural Differences in L2 Cache Capacity**

There is a significant difference in the design of the last-level cache (L2 Cache) between the GA100-105F core equipped in the CMP 170HX and the GA100-550F core used in the standard data center A800. Specifically, the A800 is equipped with a substantial 40MB L2 cache, whereas the 170HX has only 8MB of L2 cache, just one-fifth of the former's capacity.

### **b. Cache Spillover and Memory Access Bottleneck**

GPU's high-performance computing heavily relies on the principle of data locality. When designing compute-intensive operators such as matrix multiplication (GEMM) and convolution, the optimal tiling strategy is typically formulated based on the target device's L2 cache size to maximize data reuse. Since our initial design partially referenced optimization parameters (larger tile sizes) targeting A100/A800, directly transplanting these strategies to the 170HX with its smaller L2 cache led directly to severe cache spillover.

When the working set size exceeded the 8MB threshold, data was forced to be frequently transferred back and forth between the high-bandwidth memory (HBM2e) and the computing units, causing a sharp increase in memory access latency. This phenomenon of "cache thrashing" not

only masked the computational gains brought by instruction-level optimizations but, conversely, made memory bandwidth the primary bottleneck, significantly reducing the actual execution efficiency of some kernels. This also highlights the necessity for customized memory layout design tailored to constrained hardware.

## Appendix 6 Design Principles and Implementation Details of Instruction-Level Micro-Benchmarking Programs

To precisely measure the peak computational performance and energy efficiency characteristics of constrained GPUs under specific instruction sets, this study designed a dedicated micro-benchmarking framework. This framework adheres to three core design principles: **High Parallelism Saturation**, **Steady-State Operation**, and **Compiler Optimization Prevention**. The specific implementation mechanisms are elaborated in detail below, using typical instructions (FP32 FMA, FP32 EXP, FP16 mma.sync) as examples.

### 6.1 High Parallelism Architecture Design

The program employs a multi-level, multi-granularity parallel computational model to ensure all GPU computational resources (SM, SP, SFU, Tensor Cores) remain continuously saturated, thereby eliminating interference from hardware idleness on throughput measurements.

#### a. Thread-Level Parallelism (TLP)

TLP is the foundation of GPU parallel computation. To maximize hardware utilization, the program configures a massively large thread grid at the Grid level.

The design strategy for this approach involves dynamically calculating the required number of thread blocks based on the number of Streaming Multiprocessors (SMs) on the target device. By setting `numBlocks = props.multiProcessorCount * 64`, multiple active thread blocks are constantly maintained per SM. This masks the context-switching overhead of thread blocks and fully utilizes the scheduling capability of the SIMT (Single Instruction, Multiple Threads) architecture.

#### Implementation Example:

```
int blocksize = 256; // 每个线程块包含 256 个线程
```

```
int numBlocks = props.multiProcessorCount * 64; // 过度订阅 SM 资源  
int n = numBlocks * blockSize; // 启动总数万至数十万级别的线程
```

### b. Instruction-Level Parallelism (ILP)

ILP aims to exploit the processing capabilities within a single thread by issuing multiple independent instructions to the pipeline to hide execution latency.

The design strategy for this approach utilizes loop unrolling and register tiling techniques. Independent register variable groups are allocated within each thread to concurrently execute multiple sets of unrelated arithmetic operations. The `#pragma unroll` directive instructs the compiler to unroll loops, reducing stalls caused by branch misprediction and filling the floating-point pipeline.

#### Implementation Example:

```
#define ILP_FACTOR 8 // 定义指令级并行因子  
float val[ILP_FACTOR]; // 利用寄存器文件存储独立操作数  
#pragma unroll  
for (int i = 0; i < ILP_FACTOR; i++) {  
    val[i] = expf(val[i]); // 并行发射多条独立的浮点指数指令  
}
```

### c. Warp-Level Parallelism

Based on the characteristics of the SIMT architecture, the 32 threads within a Warp execute the same instruction stream.

The design strategy for this approach uses the Warp as the basic execution unit for logic mapping, particularly in Tensor Core-related tests. It ensures threads within the same Warp access consecutive memory addresses (coalesced access) and cooperatively execute matrix operation instructions to maximize memory bandwidth utilization.

### d. Tensor Core Matrix-Level Parallelism

For dedicated tensor computation units, their matrix-level parallel characteristics are leveraged to achieve exponential performance increases.

The design strategy for this approach involves using the NVIDIA Warp Matrix Multiply Accumulate (WMMA) API for programming. Each `mma.sync` instruction completes a  $16 \times 16 \times 16$  matrix multiplication and accumulation operation in a single clock cycle, equivalent to 8192 FLOPs, far exceeding scalar pipelines.

#### Implementation Example:

```
using namespace nvcuda::wmma;
fragment<matrix_a, 16, 16, 16, half, row_major> a_frag;
fragment<matrix_b, 16, 16, 16, half, col_major> b_frag;
fragment<accumulator, 16, 16, 16, half> c_frag[ILP_FACTOR];
// 显式调用 Tensor Core 指令
wmma::mma_sync(c_frag[i], a_frag, b_frag, c_frag[i]);
```

#### e. Data-Level Parallelism (DLP)

Ensures data independence between threads to avoid synchronization overhead.

The design strategy for this scheme maps data using a one-dimensional linear thread index `tid`, ensuring different threads process completely independent data streams. This eliminates data races and synchronization waits, confining the performance bottleneck solely to computational throughput rather than memory access conflicts.

### *6.2 Steady-State Long-Duration Operation Mechanism*

To obtain accurate power data (excluding transient power fluctuations) and stable average throughput, the testing program incorporates a continuous load mechanism.

**Continuous Full-Load Strategy:** A dead loop or long loop is constructed on the host (CPU) to continuously dispatch kernels to the GPU command queue. To avoid idle gaps caused by CPU-GPU synchronization, Batch Launching is employed. This involves consecutively launching 10-50 kernels before a single synchronization check. This approach ensures the GPU's command queue is never empty, keeping the hardware pipeline in a state of continuous saturation.

#### Implementation Example:

```
while (true) {
    int batch_size = 20; // 批量发射大小
```

```

for(int i = 0; i < batch_size; i++) {
    kernel<<<numBlocks, blocksize>>>(...); // 非阻塞发射
}
total_launches += batch_size;
// 仅在达到预设时间窗口后进行同步与时间检查
if (elapsed_ms >= duration_target * 1000.0) break;
}

```

### 6.3 Compiler Optimization Prevention Mechanisms

A core challenge in micro-benchmarking is preventing modern compilers (e.g., NVCC) from performing aggressive optimizations, which might identify test code as dead code or constants and eliminate it during compilation (e.g., Constant Folding). This study adopts the following technical measures to force hardware execution of the target instructions.

#### a. Runtime Parameter Dependency

To block compile-time constant propagation. We transform seemingly simple constants (e.g., 1.0) into parameters passed to the kernel function. Since the compiler cannot know the specific runtime values at compile time, it is forced to generate the corresponding computational instructions.

For example, in FMA tests, the multiplier and addend are set as parameters. Even if 1.0 and 0.0 are passed at runtime, the compiler cannot assume they are constant.

#### b. Dynamically Computed Initial Values

Generates unpredictable operands based on the thread ID (tid). Since tid is only determined at runtime, the compiler cannot precompute the result and must generate the load and initialization instructions.

#### Implementation Example:

```
val[i] = (float)tid * 0.0001f + (float)i * 0.1f;
```

#### c. Forcing Side Effects

Anchors the computation process by producing an unobservable side effect. The computed results are forcibly accumulated and written to global device memory. According to C++/CUDA

semantics, writing to global memory is considered a visible side effect. The compiler must preserve all intermediate calculation steps that produce this data and is prohibited from optimizing them away.

#### Implementation Example:

```
float res = 0.0f;  
#pragma unroll  
for (int i = 0; i < ILP_FACTOR; i++) {  
    res += val[i]; // 形成依赖链  
}  
out[tid] = res; // 关键: 强制写入显存
```

#### d. Tensor Core Explicit Dependency Chain

Purpose: A specific anti-elimination strategy for Tensor Cores.

Tensor Core computational throughput is extremely high, and results can easily be discarded if unused. Therefore, explicit data dependencies must be created. The results from multiple Matrix Fragments are accumulated into the first Fragment, forcing the compiler to retain all `mma.sync` instructions.

#### Implementation Example:

```
#pragma unroll  
for (int i = 1; i < ILP_FACTOR; i++) {  
    for (int t = 0; t < c_frag[0].num_elements; t++) {  
        // 显式累加, 产生数据依赖, 防止死代码消除  
        c_frag[0].x[t] = __hadd(c_frag[0].x[t], c_frag[i].x[t]);  
    }  
}
```

### Appendix 7 Analysis of Discrepancies Between Officially Advertised Computing Power Metrics and Actually Achievable Performance

During the empirical testing of various AI Accelerators, we have observed that the actual computational throughput of a device often deviates significantly from its officially advertised peak computing power. The basis for this numerical estimation should not be attributed to the

specific case of constrained AI Accelerators but rather to general-purpose AI accelerators, and is therefore listed separately. We also posit that such discrepancies are not solely due to hardware flaws but typically involve choices in marketing representation, applicability conditions of architectural features, and potential undisclosed hardware limitations. We categorize these phenomena into the following four main aspects for discussion.

### *7.1 Discrepancies in Numerical Precision Representation*

Modern AI Accelerators commonly adopt a heterogeneous computing unit architecture, designing distinct processing units or data paths for different numerical precisions (e.g., FP4, FP8, FP16, BF16, FP32, INT8). Since hardware design for low-precision computation (e.g., FP8, INT8) often allows for extremely high parallelism, its theoretical peak computing power is far greater than that of high-precision computation (e.g., FP32, FP64).

In market promotion, vendors tend to select the theoretical peak performance of the chip at the lowest precision or in a specific optimized mode as the core advertised metric. However, in practical AI inference or training scenarios, constrained by model requirements for numerical precision or algorithm compatibility, it is often impossible to directly use this lowest precision for computation. Consequently, there exists a substantial gap between the performance users measure under FP32 or FP16 workloads and the peak data advertised based on low precision.

### *7.2 Performance Differences between Dense and Sparse Matrices*

Starting with NVIDIA's Ampere architecture (such as the RTX 30 series), NVIDIA introduced the structured sparsity feature, allowing matrix operations to achieve a theoretical 2x acceleration in throughput compared to dense operations under specific sparsity conditions (2:4 sparsity).

However, this acceleration feature comes with strict prerequisites: the weight data of the inputs must conform to the sparsity structure, and the software stack must explicitly enable the sparse mode. The "peak computational performance" in official data sheets sometimes directly references values achieved with sparse acceleration enabled. Yet, most general AI models or operator libraries without specialized optimization are typically based on dense matrix operations. This mixing of "dense computational performance" and "sparse peak performance" often leads users to misjudge the actual general-purpose computing capability of the device.

### *7.3 Empirical Observations of FP16 Performance on GA100 Architecture: The Gap from 4x to 2x*

Regarding the GA100 architecture (widely used in A100, A800, and CMP 170HX), official whitepapers and GPU databases often claim that its CUDA CORE FP16 precision computational performance can reach 4 times that of FP32 (4x FP32). This theoretical value is based on idealized assumptions of fully saturated CUDA COREs and unobstructed data flow.

However, in our empirical tests, even under optimal memory access patterns, the actual achievable computational performance of the GA100 architecture at FP16 precision often stabilizes at only about 2 times that of FP32 (2x FP32). Through investigation, it has been found that this phenomenon is widespread in the community, but NVIDIA has not publicly released detailed design guidelines to help developers achieve the theoretical 4x peak in actual programming. In relevant discussions on official technical forums, engineers often avoid addressing this issue, suggesting that the 4x FP32 peak may only be attainable under very specific micro-architecture operating conditions or within closed optimization libraries (such as specific cuBLAS modes), rather than being the norm for general CUDA programming.

#### *7.4 Side-channel Analysis of BF16 Performance Limitation in RTX 3080 (GA102)*

During testing of the consumer flagship GPU NVIDIA GeForce RTX 3080 (based on the GA102 core), we observed an implicit performance limitation not documented in the official specifications. Specifically, when using Tensor Cores to perform matrix multiply-accumulate operations at Bfloat16 (BF16) precision, the actual throughput is only half of that at FP16 precision, rather than the theoretically expected performance parity between the two.

Concurrently, power monitoring data indicates that the energy consumption characteristics under the BF16 path also exhibit a certain suppressed state—that is, power consumption does not scale linearly with increasing computational intensity. Based on this, we speculate that the GA102 architecture in the consumer product line may have implicit hardware gating or driver-level throttling targeting BF16 Tensor Cores, intended for finer product segmentation. However, as of the completion of this study, NVIDIA has not released any official statement or documentation regarding this feature, and the specific implementation of this limitation mechanism remains a black box.