# Benchmarking and Dissecting the Nvidia Hopper GPU Architecture

Weile Luo[1], Ruibo Fan[1], Zeyu Li[1], Dayou Du[1], Qiang Wang[2,†], Xiaowen Chu[1,3,†]

***Abstract*—Graphics processing units (GPUs) are continually evolving to cater to the computational demands of contemporary general-purpose workloads, particularly those driven by artificial intelligence (AI) utilizing deep learning techniques. A substantial body of studies have been dedicated to dissecting the microarchitectural metrics characterizing diverse GPU generations, which helps researchers understand the hardware details and leverage them to optimize the GPU programs. However, the latest Hopper GPUs present a set of novel attributes, including new tensor cores supporting FP8, DPX, and distributed shared memory. Their details still remain mysterious in terms of performance and operational characteristics. In this research, we propose an extensive benchmarking study focused on the Hopper GPU. The objective is to unveil its microarchitectural intricacies through an examination of the new instruction-set architecture (ISA) of Nvidia GPUs and the utilization of new CUDA APIs. Our approach involves two main aspects. Firstly, we conduct conventional latency and throughput comparison benchmarks across the three most recent GPU architectures, namely Hopper, Ada, and Ampere. Secondly, we delve into a comprehensive discussion and benchmarking of the latest Hopper features, encompassing the Hopper DPX dynamic programming (DP) instruction set, distributed shared memory, and the availability of FP8 tensor cores. The microbenchmarking results we present offer a deeper understanding of the novel GPU AI function units and programming features introduced by the Hopper architecture. This newfound understanding is expected to greatly facilitate software optimization and modeling efforts for GPU architectures. To the best of our knowledge, this study makes the first attempt to demystify the tensor core performance and programming instruction sets unique to Hopper GPUs.**

***Index Terms*—Instruction Latency, Tensor Core, PTX, Hopper, DPX, Asynchronous Execution, Distributed Shared Memory**

## I. Introduction

Graphics Processing Units (GPUs) have experienced a significant leap in their capacity to accelerate a wide array of applications, spanning from neural networks to scientific computing. This growth has been particularly propelled by the emergence of large language models (LLMs), where models like GPT-3, boasting over 150 billion parameters, stand as prime examples [1]. Modern GPU architectures, such as Ampere, Ada, and Hopper, embody cutting-edge features like tensor cores and high-bandwidth memory, meticulously crafted to elevate artificial intelligence applications. These GPUs have now firmly established themselves as the bedrock of computing infrastructure in high-performance clusters.

Nvidia consistently introduces new GPU architectures every two years, incorporating advanced features. However, detailed micro-architecture information about these features is often limited, making precise quantification challenging. In-depth studies are increasingly essential to understand the impact of these advancements on application performance. The tensor core (TC) unit was initially introduced with the Volta architecture, focusing on accelerating deep neural networks with FP16 and FP32 precision operations. Subsequent Ampere architectures expanded TC capabilities to include sparsity and a broader range of data precisions such as INT8, INT4, FP64, BF16, and TF32. The Hopper architecture extended this further, introducing support for FP8 precision, significantly enhancing LLM training and inference acceleration. While a recent study [2] discussed TC programmability on Hopper, assembly code analysis and microbenchmarks were still conducted on Ampere and Turing, highlighting the need for further research specifically on Hopper tensor cores.

In addition to the new tensor core, as shown in Fig. 1, Hopper introduces innovative features: Dynamic Programming X (DPX) instructions, distributed shared memory (DSM), and an enhanced asynchronous execution mechanism (Tensor Memory Accelerator) for diverse scenarios. DPX instructions accelerate a wide range of dynamic programming algorithms, often involving numerous minimum/maximum operations for comparing previously computed solutions. DSM enables direct SM-to-SM communications, including loads, stores, and atomics across multiple SM shared memory blocks. Hopper supports asynchronous copies between thread blocks within a cluster, enhancing efficiency. However, detailed implementation and performance specifics remain undisclosed in existing literature. Unveiling these technical details is crucial for programmers to optimize AI applications effectively and leverage the new features of modern GPUs.

In this study, we conduct a comprehensive benchmarking of the latest GPU architectures (Ampere, Ada, and Hopper), focusing on key features like tensor cores and asynchronous operations. To the best of our knowledge, our research presents a pioneering analysis of the new programming interfaces specific to the Hopper architecture, offering a unique horizontal performance comparison among these cutting-edge GPU architectures. Many of our findings are novel and being published for the first time, providing valuable insights. We highlight the contributions of our work as follows.

- We conduct detailed instruction-level testing and analysis

[1] The Hong Kong University of Science and Technology (Guangzhou), Guangzhou, China, {wluo976, rfan404, zli755, dda487}@connect.hkust-gz.edu.cn, xwchu@ust.hk

[2] School of Computer Science and Technology, Harbin Institute of Technology (Shenzhen), Shenzhen, China, qiang.wang@hit.edu.cn

[3] The Hong Kong University of Science and Technology, Hong Kong SAR, China.

[†] Corresponding authors.

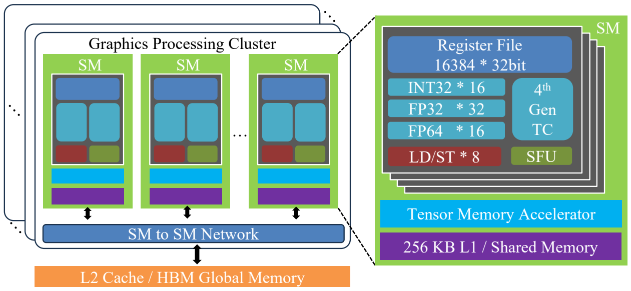
Graphics Processing Cluster
SM
SM
SM
...
SM to SM Network
L2 Cache / HBM Global Memory
SM
Register File
16384 * 32bit
INT32 * 16
FP32 * 32
FP64 * 16
4th Gen TC
LD/ST * 8
SFU
Tensor Memory Accelerator
256 KB L1 / Shared Memory

Fig. 1: Hopper architecture

on memory architecture and tensor cores across three GPU generations with different architectures. Our analysis highlights the unique advantages and potential of the Hopper architecture.

- We compare AI performance across recent GPU generations, examining latency and throughput of tensor cores at the instruction level, transformer engines at the library level, and real LLM generation at the application level. This comprehensive analysis provides valuable insights for AI system and application design and optimization, aiding informed decisions by researchers, developers, and manufacturers for next-generation AI solutions.
- Our research represents the inaugural exploration of Hopper architecture's distinctive features, encompassing DPX, asynchronous memory operations, and distributed shared memory. These innovations hold significant potential for enhancing GPU programming methodologies. Our study, by evaluating the performance of these features, contributes to performance modeling and algorithm design in dynamic programming and scientific computing. This contribution may fully unlock the GPU performance potential, driving advancements in the field.

## II. Related Work

Analyzing GPU microarchitectures and instruction-level performance is crucial for modeling GPU performance and power [3]–[10], creating GPU simulators [11]–[13], and optimizing GPU applications [12], [14], [15]. Early research [16]–[20] extensively dissected undisclosed GPU microarchitecture characteristics, particularly in older architectures like Fermi, Kepler and Maxwell. Jia et al. [19], [20] further evaluated the Volta and Turing GPU architectures, adding some special measurements for the new tensor cores.

Fused Matrix Multiplication Accumulation (MMA), a critical operation in AI, is predominantly accelerated by tensor cores (TCs) in Nvidia GPUs since the Volta architecture. To harness TCs' full potential, extensive research has dissected TCs across Volta, Turing, and Ampere architectures, focusing on compute throughput and register mapping. Early studies [21], [22] examined the first-generation tensor cores on Volta GPUs, emphasizing legacy *wmma* APIs and benchmarks using vendor CUBLAS and CUTLASS libraries. Subsequent work by Jia et al. [19], [20] expanded this analysis to the Turing architecture, providing preliminary assembly code analysis of *wmma* APIs for TCs. However, comprehensive instruction-level microbenchmarks to fully exploit TC performance and numeric behaviors were lacking. Yan et al. and Md Aamir et al. [15], [23], [24] delved into assembly code (SASS level) benchmarking, demystifying Volta and Turing TCs. Their focus was on assembly-level optimizations for matrix multiplication performance. Additionally, they optimized half-precision matrix multiplication on TCs. A study by Fasi et al. [25] scrutinized the numeric behaviors of TCs, exploring rounding modes and subnormal behaviors of TF32, BF16, and FP16 data types. However, it's noted that *wmma* APIs have limitations in operand shapes and cannot fully leverage the new sparse matrix multiplication features on Ampere and Hopper GPUs.

On the contrary, the new *mma* programming interface was proposed since the Turing architecture and has evolved to support sparse matrix multiplication (*mma.sp*) on the architecture above Ampere. Sun et al. [2] conducted a comprehensive study of instruction-level microbenchmarks on the current *mma* APIs (*ldmatrix, mma and mma.sp*) instead of legacy *wmma* APIs, exploring the full performance of tensor cores, the computation numeric behaviors of low-precision floating points, and the new sparse matrix multiplication feature of Turing and Ampere GPUs. As for the newest Hopper, the SASS codes of *wgmma* and *mma* can be even more diverse to support a wide range of computational precisions. The usage as well as their performance is still uncovered.

In addition to performance, energy efficiency is also an often discussed factor. In addition to the above literature that models GPU power, some work also benchmarks the application level. [26] empirically investigated the impact of GPU Dynamic Voltage and Frequency Scaling (DVFS) on energy consumption and performance during deep learning, testing various GPU architectures, DVFS settings, and DNN configurations. [27] performed an empirical comparison of performance and energy efficiency across different AI accelerators from multiple vendors when training DNNs.

Despite the popularity of tensor cores and AI, another trend is the support of DPX, asynchronous operation support and distributed shared memory. To this end, benchmarking and dissecting the performance details of the modern GPU architectures is necessary and emerging. Revealing them helps programmers investigate more optimization opportunities.

## III. Methodology

### A. Memory Unit Access Latency and Throughput

In this subsection, we focus on two memory performance metrics: latency and throughput. Our memory test in this subsection uses a method similar to P-chase microbenchmark, which was first introduced on [28] [29]. Below we will introduce how we test these two metrics.

*1) L1 Cache:* For the latency test, we first load the data from global memory to the L1 cache using the *ca* modifier. Then we use a thread to access this L1 cache to obtain the latency. For the throughput test, we also first load the memory into the L1 cache using the *ca* modifier. Since the L1 cache is exclusive to SM, we only issue a block with 1024 threads to repeatedly access the L1 cache. We record the time consumed and the amount of data accessed to calculate the bandwidth of the L1 cache.

*2) Shared memory:* Testing the shared memory is similar to testing the L1 cache. The only difference is that there is no need to specify modifiers to explicitly warm up shared memory. We can test it directly by declaring shared memory. Since shared memory can only be accessed within a block (distributed shared memory is not considered in this subsection), we use one thread to test latency and a block with 1024 threads to test bandwidth just like testing the L1 cache.

*3) L2 Cache:* For the latency test, we use the same method as for L1 cache testing. The only difference is that the *cg* modifier is used instead of *ca*, ensuring that the cache we load is L2. For the throughput test, we first load the memory into the L2 cache using the *cg* modifier. Since the L2 cache is shared by all SMs, we use a large number of blocks to access the L2 cache. We then calculate the bandwidth of the L2 cache based on the amount of data accessed and the time consumed.

*4) Global memory:* For the latency test, we first allocate a global memory that exceeds the L2 size to avoid L2 prefetching, and then initialize the global memory. Initialization has two purposes. The first is to enable the test to be performed at a fixed stride, and the second is to warm up the TLB to avoid the occurrence of cold misses. When we started the test, we launched four consecutive threads, each of which was responsible for reading 8 bytes, thus forming a 32-byte memory read transaction. Finally, we can calculate the memory access latency of each thread. For the throughput test, we allocate a much larger memory space than L2. We set up each thread to use vectorized memory access to read float4. Each thread reads 5 times and writes 1 time. Finally, we calculate the memory bandwidth based on the time consumed and the amount of data.

### *B. Tensor Core Latency and Throughput*

*1) Tensor Core's Evolution:* Table I illustrates the progression of TCs, encompassing enhancements in precision, operand shapes, programming modes, and execution modes. In the initial Volta Architecture, first-generation TCs exclusively supported FP16 as the input data type. Subsequent architectures, including Ampere, Ada, and Hopper, introduced support for a broader range of data types such as BF16, TF32, FP64, INT8, INT4, Binary, and more. The programming of TCs has also seen continuous improvement. Ampere and Ada Lovelace GPUs provide users with the flexibility to utilize either the legacy C-level *wmma* APIs or PTX-level *mma* instructions. Notably, the *wmma* APIs had limitations in fully harnessing TCs' capabilities, whereas *mma* instructions could leverage advanced sparse matrix multiplication capabilities introduced since Ampere. In the case of Hopper GPUs, new warp-group-level *wgmma* instructions were introduced. Both *wmma* and *mma* APIs remain supported in Hopper, but we find that the complete potential of Hopper TCs can only be realized through *wgmma* instructions.

Fig. 2 provides examples of both *mma* and *wgmma* instructions, demonstrating mixed-precision capabilities. An *mma* instruction computes $D(m \times n) = A(m \times k) \times B(k \times n) + C(m \times n)$ and is executed by one CUDA warp (i.e., 32 threads) synchronously. In contrast, *wgmma* for Hopper computes $D(m \times n) = A(m \times k) \times B(k \times n) + D(m \times n)$ and is asynchronously executed by one CUDA warp group (i.e., four CUDA warps). The matrix shapes for *mma* instructions can be $m16n8k16$ or $m16n8k8$, while *wgmma* supports $m64nNk16$, where $N$ can be 16, 32, 64, 128, 256, and so on, with the complete valid range listed in [30]. Notably, *wgmma* has the advantage of directly loading matrices $A$ and $B$ from shared memory, unlike *mma*, which requires storing all matrices in the register file before execution. We use the term "SS" to denote the *wgmma* instruction that loads both $A$ and $B$ from shared memory, while "RS" is used for the instruction that loads $A$ from the register file. Additionally, *wgmma* offers support for certain useful arguments not required for *mma*. Further details are provided in [30].

TABLE I: Properties of the latest generations of Tensor Cores

| Arch | Precision | Programmability | Mode |
|---|---|---|---|
| Ampere | FP16,BF16,<br>TF32,FP64,<br>INT8,INT4,Binary | C: wmma<br>PTX: mma, mma.sp | Sync |
| Ada | FP16,BF16,FP8,<br>TF32,FP64,<br>INT8,INT4,Binary | C: wmma<br>PTX:mma, mma.sp | Sync |
| Hopper | FP16,BF16,FP8,TF32,<br>FP64,INT8,Binary | C: wmma<br>PTX: mma, mma.sp | Sync |
| | | PTX: wgmma, wgmma.sp | ASync |

*2) Benchmarking Levels and Performance Metrics:* We conduct micro-benchmarking of TCs at the PTX level, as it strikes a suitable balance between granularity and complexity. Additionally, we disassemble PTX instructions to SASS codes to achieve a deeper understanding of the operations. We focus on assessing two critical indicators: latency and throughput. Latency signifies the elapsed time, measured in clock cycles, starting from the initiation of instruction issuance to the execution pipeline and concluding when the results become accessible for subsequent usage. This measurement is specifically labeled as "completion latency." To elaborate further, we issue a single synchronous TC instruction (i.e., *mma*) using one CUDA warp per SM, whereas one asynchronous TC instruction (i.e., *wgmma*) is issued utilizing four CUDA warps (comprising a warp group) on one SM. We execute the instruction 1024 times within a CUDA kernel. Throughput is quantified as $Total\_OPS/Duration$, where $OPS$ represents multiplication or addition operations. It's important to emphasize that, unlike the approach described in [2], we abstain from utilizing total clock cycles to compute throughput due to potential variations in GPU frequencies during the execution of different TC instructions.

### *C. Transformer Engine*

The Transformer Engine (TE) [31] is a library specifically designed to accelerate Transformer models [32], following

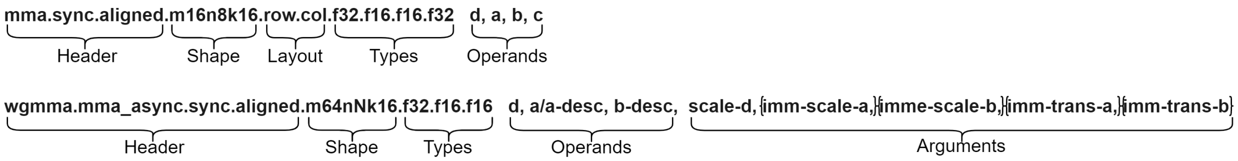
mma.sync.aligned.m16n8k16.row.col.f32.f16.f16.f32
d, a, b, c
Header
Shape
Layout
Types
Operands
wgmma.mma_async.sync.aligned.m64nNk16.f32.f16.f16
d, a/a-desc, b-desc,
scale-d, {imm-scale-a,}{imme-scale-b,}{imm-trans-a,}{imm-trans-b}
Header
Shape
Types
Operands
Arguments

Fig. 2: The *mma* and *wgmma* instructions that perform $D = A \times B + C$ and $D = A \times B\{+D\}$, respectively.

the introduction of the Hopper architecture. It is capable of leveraging the FP8 precision offered by both the Hopper and Ada architectures. In particular, it provides a variety of optimized modules for Transformer layers that can be utilized within the widely-used deep learning framework, PyTorch [33]. In this subsection, we describe the benchmark details of the Transformer Engine on different modules, as well as the inference performance of FP8 in Large Language Models.

*1) Linear Layer:* In the Transformer architecture, most of the computational overhead comes from the linear layers, specifically matrix multiplication while the Transformer Engine provides the `te.Linear` implementation to perform matrix multiplication with higher throughput on FP8 Tensor Cores. When employing the Transformer Engine with `te.Linear` for matrix multiplication in FP8 precision, TE converts both the input and weights in the linear layer to FP8. This conversion process involves data transformation and quantization operations. For example, as the dynamic range of FP8 may not encompass the maximum value of the input tensor, TE identifies the maximum absolute value of the input data as the scaling factor. It then adjusts the input data to fit the representation range of FP8 using $inp_{fp8} = inp_{fp16}/scale$, followed by matrix multiplication in FP8 Tensor Core $out_{fp8} = inp_{fp8} \times w_{fp8}$. Finally, it scales the result with $out_{fp16} = out_{fp8} \times scale$. This operation would introduce some overhead.

As depicted in Fig. 3, when performing matrix multiplication with relatively small matrix sizes, the proportion of overhead attributed to conversion is significantly larger than that resulting from the GEMM kernel computation in FP8 Tensor Cores. To evaluate and investigate the support and optimization of TE for linear layers, we measure the throughput (GFLOPS) of `te.Linear` for two identical matrices $D(n \times n) = A(n \times n) \times B(n \times n)$.

Overheads dominate execution time for small matrices

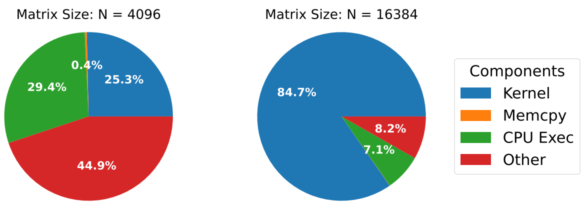

Fig. 3: Proportion of execution time for different operators when performing FP8 matrix multiplication using `te.Linear`.

*2) TransformerLayer:* The Transformer Engine (TE) capitalizes on the efficiency improvements provided by FP8 through specific operator fusion optimizations for transformer layer structures. For example, `te.LayerNormMLP` combines layernorm and MLP within the transformer structure, allowing data transmission between layernorm and the subsequent MLP layer to adopt the FP8 format. This approach not only eliminates data format conversion overhead but also effectively leverages FP8 memory transfer advantages.

TE offers a `te.TransformerLayer` module that encompasses all operator optimizations for transformer layer structures, facilitating the implementation of various Large Language Model (LLM) structures by adjusting its parameters. However, some operators, such as `Softmax` and `GeLU`, have not been quantized to FP8 by TE, resulting in significant data format conversion overhead. Additionally, the DotProductAttention operator uses flash-attention [34] rather than FP8 Tensor Cores.

↳ Not FP8

The computational overhead of the transformer layer's linear layer primarily depends on the hidden size, raising the question of which hidden state (dimension of embedding) will yield better performance for TE with FP8 compared to FP16. We investigate this by examining the open-source LLM, Llama [35], [36], modifying the activation function to SwiGLU [37] and normalization to RMSNorm [38]. We set layer structure parameters based on the hidden state's size, with hidden states 4096, 5120, and 8192 corresponding to Llama configurations 7b, 13b, and 70b, respectively.

TABLE II: Parameter settings of `te.TransformerLayer` for various *hidden_size* values

| hidden_size | 1024 | 2048 | 4096 | 5120 | 8192 |
|---|---|---|---|---|---|
| ffn_hidden_size | 2816 | 5632 | 11008 | 13824 | 22016 |
| num_attention_heads | 8 | 16 | 32 | 40 | 64 |

We fixed the input as (4, 512, *hidden_size*), where 4 is the *batch_size*, 512 is the *sequence_length*, and the attention mask is set to None. We then calculated the latency (ms) required for encoding a single layer once, focusing on the encoding task for a single layer.

*3) LLM Generation:* Currently, the Transformer Engine has not provided optimal support for mainstream decode-only casual language models. In order to test the inference performance of the Transformer Engine on this type of model like, Llama [35], we replaced the `nn.Linear` and `RMSNorm` in the original model structure with `te.Linear` and `te.RMSNorm`, respectively, to ensure that most modules in the model utilize the Transformer Engine.

To evaluate the effectiveness of TE in generating text for Llama, we followed [39] using the ShareGPT dataset as input for the LLM. The ShareGPT dataset comprises conversations between users and ChatGPT [40], which have been shared by the users. We tokenize these datasets and, based on their input and output lengths, generate synthesized client requests.

In order to test and ensure compatibility with different hardware architectures (different memory capacities), we set the maximum input length to 128 and the maximum text generation length to 128. Furthermore, to meet the dimension requirements of `Te.Linear`, we set the batch size to 8.

We use throughput as the evaluation metric, which represents the total text length that can be processed per second: $Throughput = (input\_len + output\_len)/time$.

### D. New CUDA Programming Features

*1) DPX:* Nvidia offers DPX functions[1] from CUDA 12 onward to accelerate dynamic programming code, enhancing programming ease. On the latest Hopper architecture, these functions are hardware-accelerated. Our test of DPX functions focuses on the instruction latency and throughput. For latency assessment, we utilize a thread to iteratively issue DPX functions, calculating their average latency. In the throughput test, we employ a block to repeatedly issue DPX functions, determining the DPX instruction throughput for each SM. To pinpoint the location of DPX acceleration hardware, we vary the number of launched blocks and observed the relationship between DPX throughput and the launched block count.

*2) Asynchronous Data Movement:* The introduction of asynchronous execution is a highlight of the Ampere architecture. This feature allows for non-blocking data transfers between GPU global memory and shared memory using *cuda::memcpy_async*, avoiding thread occupation during data movement. It facilitates the overlap of computations with data transfers, effectively reducing overall execution time. Building upon Ampere's asynchronous copies, the Hopper architecture enhances this with a more advanced Tensor Memory Accelerator (TMA) for sophisticated asynchronous copying.

To assess this feature's efficiency, we conduct empirical studies using the *globalToShmemAsyncCopy* application from official CUDA samples[2]. The application implements matrix multiplication and leverages asynchronous data copy from global to shared memory for compute capability 8.0 or higher. We compare two implementations: "SyncShare", employing synchronous copy to shared memory for conventional tiled matrix multiplication, and "AsyncPipe", enhancing tiling with asynchronous data movement. The asynchronous version uses a two-stage pipeline with doubled shared memory buffer size, enabling computation and data copy overlap across different execution streams. Matrix A's width and Matrix B's height are set to 2048, determining each thread's computational workload. We vary the block size from $8\times 8$ to $32\times 32$ to assess asynchronous operations' effects on warp concurrency. Additionally, we benchmark different block numbers to optimize computational throughput by adjusting Matrix A's height and Matrix B's width.

*3) Distributed Shared Memory:* The Hopper architecture features a direct SM-to-SM communication network within clusters, enabling threads from one thread block to access shared memory of another block, known as distributed shared memory (DSM). According to the official documentation, this network can reduce data transfer overhead between blocks on different SMs by up to 7×. Additionally, for cases where shared memory demand restricts active block numbers on an SM, DSM can partition data within the same cluster, alleviating shared memory demand per block. The programmability of DSM is facilitated through the CUDA C function, `cluster.map_shared_rank(SMEM, DST_BLOCK_RANK)`, returning the shared memory address of the target block. Here, `SMEM` represents the shared memory pointer, and `DST_BLOCK_RANK` is the target block rank in the cluster. This is compiled into PTX code *mapa*, which maps the address of the shared variable in the target block.

We assess DSM using three benchmarks.

(1) Latency Measurement: To measure inter-SM data transfer latency, we launch two blocks, each with one thread. Using the *clock()* function, we record the latency of adding register values of the first block to the second one. We utilize *mov.u32 %0, %%smid* PTX code to record the SM ID of each block, ensuring they run on different SMs.

(2) Throughput Measurement by RBC (Ring-Based Copy): For DSM data communication throughput, we propose the ring-based copy (RBC) scheme. We launch one block per SM and gather them into clusters. In each cluster, we arrange threads in each block ranked by $R$ to add their register values to those of the block ranked by $(R+1)\%CS$, where $CS$ refers to the cluster size. Instruction-level parallelism (ILP) is employed to maximize bandwidth. We tune cluster size, block size, and ILP to measure their impact on DSM throughput.

(3) Histogram Application with DSM: We redesign the histogram[3] application using DSM, distributing bins across blocks in the same cluster. During histogram counting via shared memory, each thread loads the element and determines the DSM address for the target bin, followed by an atomic increment operation. We adjust cluster size, block size, and bin count, measuring element processing throughput.

## IV. Experimental Results

In this section, we first introduce the GPUs used in this study. Then, we introduce and analyze the performance of memory, Tensor Core, AI, and new CUDA features. Due to space limitations, we hereby present the most meaningful findings. The complete experimental results can be found in the public preprint version or reproduced by the open-sourced codes after the review process.

[1] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#dpx

[2] https://github.com/NVIDIA/cuda-samples/Samples/3_CUDA_Features/globalToShmemAsyncCopy

[3] https://github.com/NVIDIA/cuda-samples/Samples/2_Concepts_and_Techniques/histogram

TABLE III: Comparison of the Properties of the Ampere, Ada Lovelace and Hopper Devices

| Device | A100 PCIe | RTX4090 | H800 PCIe |
|---|---|---|---|
| Comp. Capability | 8.0<br>(Ampere) | 8.9<br>(Ada Lovelace) | 9.0<br>(Hopper) |
| SMs * cores/SM | 108 * 64 | 128 * 128 | 114 * 128 |
| Max Clock rate | 1410 MHz | 2520 MHz | 1755 MHz |
| Mem. Size | 40GB | 24GB | 80GB |
| Mem. Type | HBM2e | GDDR6X | HBM2e |
| Mem. Clock rate | 1215 Mhz | 10501 Mhz | 1593 Mhz |
| Mem. Bus | 5120-bit | 384-bit | 5120-bit |
| Mem. Bandwidth | 1555 GB/s | 1008 GB/s | 2039 GB/s |
| Tensor Core | 432<br>(3rd Gen.) | 512<br>(4th Gen.) | 456<br>(4th Gen.) |
| DPX hardware | No | No | Yes |
| Distributed shared memory | | | |

TABLE IV: Latency clocks of different memory scopes

| Type | RTX4090 | A100 | H800 |
|---|---|---|---|
| L1 Cache | 43.4 | 37.9 | 40.7 |
| Shared | 30.1 | 29.0 | 29.0 |
| L2 Cache | 273.0 | 261.5 | 263.0 |
| Global | 541.5 | 466.3 | 478.8 |

### A. Experimental Setup

In this work, we select the most representative GPUs of the Ampere, Ada Lovelace, and Hopper architectures, which are A100 PCIe, RTX4090, and H800 PCIe respectively. Their basic hardware properties are shown in Table III. On the RTX4090, the driver version we use is 530.30.02 and the CUDA version is 12.1. On A100 and H800, the driver version we use is 535.104.05, and the CUDA version is 12.2.

### B. Memory Latencies and Throughputs

Table IV shows the memory access latency at different memory levels. We can observe that on devices with different architectures, their memory access latencies are close, indicating that the memory levels are similar. But we can find that the global memory latency of A100 and H800 using HBM2e will be slightly lower than RTX4090. In the comparison of latency at different memory levels, it can be observed that on the three devices, the average latency of the L2 cache is 6.5 times that of the L1 cache, and the average latency of the global memory is 1.9 times that of the L2 cache.

Table V shows the memory access throughput at different memory levels. In the cache throughput test, we use different data types for memory access. We can observe that using vectorized memory access (FP32.v4, equivalent to CUDA's float4) can always achieve better performance. It is worth noting that in FP64 Cache access, the throughput of RTX4090 and H800 will be much smaller than the normal value. This is because we need to perform calculations after accessing memory to avoid the elimination of our memory access instructions by the compiler. However, the FP64 addition throughput of both RTX4090 and H800 we measured is 16byte/clk/SM. Therefore, the bottleneck of the memory access test here lies in the FP64 computing unit, which does not represent the actual throughput of the cache memory.

The maximum throughput of L1 Cache and shared memory of the three devices are similar. However, for the throughput of L2 Cache, H800 is 2.6 times and 2.2 times that of RTX4090 and A100 respectively.[4] In the memory throughput test, our results reach 92%, 90%, and 91% of the theoretical performance on RTX4090, A100, and H800 respectively. In the comparison of L2 and Global, the L2 cache throughput of RTX4090, A100, and H800 is 4.67, 2.01, and 4.23 times the global memory throughput respectively.

### C. Tensor Core Latencies and Throughputs

**SASS analysis.** We perform the disassembly of *mma* and *wgmma* instructions specifically for Hopper Tensor Cores, and the results are presented in Table VI. The *mma* instructions undergo compilation into SASS instructions, with the naming convention following the established patterns: HMMA (for floating-point types), IMMA (for integer types), and BMMA (for binary types). It is worth noting the existence of two specialized types within *mma*: INT4 and FP8.

For INT4, on Ampere and Ada Tensor Cores, *mma* instructions are compiled into IMMA.16832.S4.S4 instructions. However, a noteworthy deviation occurs on Hopper, where INT4 *mma* instructions are compiled into a series of IMAD instructions, which eventually run on the CUDA cores. This deviation results in performance that may fall short of the expected performance levels achievable with Tensor Cores. Additionally, there are no *mma* instructions available for FP8, a new data type introduced in Ada.

INT4 on
Ada & Ampere
→ Tensor core
H100
↓
Not Tensor
cores

The latest *wgmma* instructions are currently exclusive to Hopper Tensor Cores, despite Nvidia's assertion that both Ada and Hopper feature 4th generation Tensor Cores. Unlike *mma*, *wgmma* instructions are compiled into the new GMMA SASS instructions. Users have the capability to program two variations of FP8 (E5M2 and E4M3) Tensor Cores using *wgmma*. It's important to note that *wgmma* does not offer support for INT4 Tensor Cores.

***mma* results.** Table VII provides an overview of the latency and throughput measurements for *mma* instructions across A100, RTX4090, and H800 Tensor Cores GPUs. Note that the sparse shapes in the table represent compressed shapes. In other words, the $k$ of the actual instruction modifier is twice that in the table.

For A100 and H800, the same-precision *mma* instructions with the larger shapes commonly achieve better throughputs. But this phenomenon disappears on RTX4090. Sparse and dense *mma* instructions exhibit equivalent latency, with sparse *mma* instructions achieving higher throughputs. On the RTX4090, sparse *mma* instructions can achieve up to double the throughput compared to their corresponding dense counterparts, aligning with the speedup claims stated in the vendor's documentation. However, for the A100, only the

For most
→ shapes

[4] Note that for the L1 cache, the amount of data they transfer per clock is almost the same. However, since the order of clock frequency from high to low is RTX4090, H800, and A100, the order of throughput per unit time from high to low is also RTX4090, H800, and A100. The same calculation method also applies to L2 cache.

TABLE V: Throughput at different memory levels

| Type | RTX4090 | | | A100 | | | H800 | | |
|---|---|---|---|---|---|---|---|---|---|
| L1 Cache (byte/clk/SM) | FP32 | FP64 | FP32.v4 | FP32 | FP64 | FP32.v4 | FP32 | FP64 | FP32.v4 |
| | 63.7 | 13.3 | 121.2 | 99.5 | 120.0 | 106.8 | 125.8 | 16.0 | 124.1 |
| L2 Cache (byte/clk) | FP32 | FP64 | FP32.v4 | FP32 | FP64 | FP32.v4 | FP32 | FP64 | FP32.v4 |
| | 1622.2 | 1500.8 | 1708.0 | 1853.7 | 1990.4 | 2007.9 | 4472.3 | 1817.3 | 3942.4 |
| Shared Memory (byte/clk/SM) | 127.9 | | | 128.0 | | | 127.9 | | |
| Global Memory (GB/s) | 929.8 | | | 1407.2 | | | 1861.5 | | |
| L2 vs. Global | 4.67× | | | 2.01× | | | 4.23× | | |

TABLE VI: SASS Instructions for Different Hopper Tensor Core PTX Instructions

| A/B | C/D | mma | wgmma |
|---|---|---|---|
| FP16 | FP16 | HMMA.16816.F16 | HGMMA.64x256x16.F16 |
| FP16 | FP32 | HMMA.16816.F32 | HGMMA.64x256x16.F32 |
| TF32 | FP32 | HMMA.1688.F32.TF32 | HGMMA.64x256x8.F32.TF32 |
| FP8 | FP16 | × | QGMMA.64x256x32.F16.E5M2.E5M2<br>QGMMA.64x256x32.F16.E4M3.E4M3 |
| FP8 | FP32 | × | QGMMA.64x256x32.F32.E4M3.E4M3<br>QGMMA.64x256x32.F32.E5M2.E5M2 |
| INT8 | INT32 | IMMA.16832.S8.S8 | IGMMA.64x256x32.S8.S8 |
| INT4 | INT32 | IMAD.MOV.U32 | × |
| Binary | INT32 | BMMA.168256.AND.POPC | BGMMA.64x256x256.AND.POPC |

For A100, 2x for larger matrices

For H100, only 1.42x ☆

sparse *mma* instructions with larger shapes can realize the theoretical speedups. In the case of the H800, sparse *mma* instructions can only achieve an average speedup of 1.42 times over the dense ones. This highlights that on Hopper Tensor Cores, sparse *mma* instructions may not fully harness the capabilities of the sparse tensor cores.

The achieved throughput on A100 exceed 95% of their theoretical peak performance. The achieved throughput of RTX4090 is higher than the official theoretical peak performance. This is because we find that our RTX4090 runs at a higher frequency than the officially announced boost frequency. However, on Hopper Tensor Cores, *mma* instructions can only attain an average of 62.9% of the theoretical peak performance. When developing high-performance applications tailored for Hopper GPUs, such as matrix multiplication and convolution, users should exercise caution in their utilization of *mma* instructions.

***wgmma* results.** As a set of warp-group-level Tensor Core instructions designed specifically for Hopper GPUs, *wgmma* instructions are the pioneering instructions to be executed asynchronously. Table VIII and IX show the measured latency and throughputs of dense and sparse *wgmma* instructions, respectively. When initializing matrices with zeros, we achieve throughputs exceeding 95% of the theoretical peak performance. We observe a decrease in Tensor Core performance when initializing matrices with random values, especially pronounced when utilizing FP16 as the computation type and FP32 for accumulation. This phenomenon is primarily attributed to the power consumption nearing the 350W power limit of the H800-PCIe, subsequently causing a reduction in frequency. Users working with Tensor Cores on the H800-PCIe GPU should take into full consideration the power constraints when performing computations.

In the case of dense *wgmma*, with $N$ set to 128, we observe that the latency for all data types corresponding to the instructions is 128.0. Interestingly, under both "RS" and "SS" modes, the latency and throughputs for the same instruction remain relatively consistent. We attribute this phenomenon to the efficient concealment of shared memory access latency due to the substantial computational workload and asynchronous nature of the process.

In the context of sparse *wgmma* instructions, the latencies for "RS" and "SS" modes are 128.0 and 144.0, respectively. Additionally, we observe that the achieved throughputs in "SS" modes are lower compared to "RS" modes, which is notably different from dense *wgmma*. We find that in sparse *wgmma*, the "SS" mode retrieves data from the shared memory of size $m \times k$ and performs a 2:4 sparse pruning based on metadata during the execution of the sparse *wgmma* instruction. In contrast, the "RS" mode directly accesses data from the pruned register file of size $m \times k/2$. The high shared memory access demand (twice as much) may lead to latency that cannot be effectively concealed by Tensor Core computation, resulting in sparse *wgmma* instructions in "SS" modes failing to achieve the expected peak performance. ☆

***wgmma* results with different $N$ values.** We conduct tests using the example of wgmma.m64nNk16.f32.f16.f16, varying the value of $N$, and the results are presented in Table X. When $N$ is greater than or equal to 64, all *wgmma* instructions can achieve throughputs that closely approach peak performance. However, when $N$ is less than 64, the achieved throughputs decrease, and the "SS" mode of instructions exhibits higher latency than the "RS" mode, while the achieved throughputs are lower than those of the "RS" mode. As $N$ decreases, the computational density of *wgmma* instructions gradually diminishes, making it challenging to conceal the latency associated with shared memory access, leading to the aforementioned phenomena. Therefore, when utilizing wgmma instructions, it is advisable to opt for larger values of $N$ ($>= 64$) whenever possible to attain superior performance.

**Energy efficiency.** Although Tensor Cores have impressive performance, their energy efficiency should also be considered. Excellent energy consumption ratio will bring benefits both economically and environmentally. We choose mma for energy consumption testing because mma is the current compatible instruction, and previous codes can run directly on the latest hopper architecture. We test the largest operational shape in Table VII. The energy efficiency of *mma* instructions is shown in Table XI. In terms of dense instructions, the average energy efficiency of H800 is 1.60 times and 1.69 times that of A100 and RTX4090 respectively. In terms of sparse instructions, the

TABLE VII: Different dense and sparse *mma* instructions on A100, RTX4090 and H800 Tensor Cores. Latency (LAT) is measured in clock cycles. Throughput is measured in TFLOPS or TOPS/s. Peak performance (A100): FP16 (312 TFLOPS); TF32 (156 TFLOPS); INT8 (624 TOPS). Peak performance (RTX4090): FP16 (330.3 TFLOPS); TF32 (82.6 TFLOPS); INT8 (660.6 TOPS). Peak performance (H800): FP16 (756.5 TFLOPS); TF32 (378 TFLOPS); INT8 (1513 TOPS).

| **A/B** | **C/D** | **Shape** | **LAT/Throughput** | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | **A100** | | **RTX4090** | | **H800** | |
| | | | Dense | Sparse | Dense | Sparse | Dense | Sparse |
| FP16 | FP16 | m16n8k8 | 17.7/310.0 | 17.3/408.4 | 17.7/355.3 | 17.3/713.2 | 16.0/368.6 | 16.0/493.8 |
| FP16 | FP16 | m16n8k16 | 24.6/310.6 | 24.5/622.8 | 24.6/357.6 | 24.5/711.8 | 24.1/494.4 | 24.0/722.8 |
| FP16 | FP32 | m16n8k8 | 17.5/299.6 | 18.0/394.1 | 18.8/177.8 | 18.8/357.4 | 16.0/363.7 | 16.0/488.7 |
| FP16 | FP32 | m16n8k16 | 26.0/303.4 | 24.5/603.3 | 33.0/178.9 | 33.0/356.0 | 24.1/490.7 | 24.0/721.8 |
| TF32 | FP32 | m16n8k4 | 17.8/149.5 | 18.2/196.8 | 19.2/89.0 | 19.0/178.0 | 16.5/180.6 | 16.4/240.7 |
| TF32 | FP32 | m16n8k8 | 26.3/151.5 | 26.7/301.5 | 33.4/89.0 | 33.3/178.7 | 24.5/246.4 | 24.4/363.3 |
| INT8 | INT32 | m16n8k16 | 17.6/594.8 | 18.0/788.5 | 17.3/707.6 | 17.3/1412 | 16.1/730.3 | 16.1/970.0 |
| INT8 | INT32 | m16n8k32 | 26.0/607.6 | 26.6/1210 | 24.5/711.7 | 24.6/1423 | 24.0/977.9 | 24.2/1435 |

TABLE VIII: Variations in Dense *wgmma* Instructions for H800 Tensor Cores. Latency (LAT) is quantified in clock cycles, while throughput is expressed in TFLOPS or TOPS. The peak throughputs for FP16, TF32, FP8, and INT8 are 756.5, 373, 1513, and 1513, correspondingly. "Zero" or "Rand" signifies that all matrices are initialized with either zero or randomly generated values. "SS" implies that both matrix A and B are stored in shared memory, while "RS" signifies that matrix A is stored in the register file, whereas B is stored in shared memory. "Rand" has the same latency as "Zero".

| **A/B** | **C/D** | **Instruction** | **LAT/Throughput (SS,Zero)** | **LAT/Throughput (RS,Zero)** | **Throughput (SS,Rand)** | **Throughput (RS,Rand)** |
|---|---|---|---|---|---|---|
| FP16 | FP16 | m64n256k16 | 128.0/729.3 | 128.0/729.2 | 704.5 | 703.7 |
| FP16 | FP32 | m64n256k16 | 128.0/728.5 | 128.0/731.9 | 665.4 | 667.5 |
| TF32 | FP32 | m64n256k8 | 128.0/364.4 | 128.0/364.6 | 357.1 | 357.3 |
| FP8 | FP16 | m64n256k32 | 128.0/1448.4 | 128.0/1448.0 | 1439.2 | 1440.3 |
| FP8 | FP32 | m64n256k32 | 128.0/1447.5 | 128.0/1455.0 | 1417.2 | 1419.8 |
| INT8 | INT32 | m64n256k32 | 128.0/1448.7 | 128.0/1447.9 | 1442.3 | 1442.2 |

TABLE IX: Different sparse *wgmma* Instructions on H800 Tensor Cores. The definitions of LAT and throughput can be found in the caption of Table VIII. "Rand" has the same latency as "Zero".

| **A/B** | **C/D** | **Sparse Instruction** | **LAT/Throughput (SS,Zero)** | **LAT/Throughput (RS,Zero)** | **Throughput (SS,Rand)** | **Throughput (RS,Rand)** |
|---|---|---|---|---|---|---|
| FP16 | FP16 | sp.m64n256k32 | 144.0/1308.0 | 128.0/1472.0 | 1257.8 | 1362.3 |
| FP16 | FP32 | sp.m64n256k32 | 144.0/1312.3 | 128.0/1476.2 | 1194.3 | 1277.5 |
| TF32 | FP32 | sp.m64n256k16 | 144.0/656.8 | 128.0/735.4 | 644.9 | 721.7 |
| FP8 | FP16 | sp.m64n256k64 | 144.0/2619.9 | 128.0/2945.0 | 2588.6 | 2782.4 |
| FP8 | FP32 | sp.m64n256k64 | 144.0/2622.8 | 128.0/2931.0 | 2588.7 | 2722.3 |
| INT8 | INT32 | sp.m64n256k64 | 144.0/2612.4 | 128.0/2933.0 | 2593.9 | 2898.3 |

average energy efficiency of H800 is 1.33 times and 1.39 times that of A100 and RTX4090 respectively. We can find that the H800 has significantly higher energy efficiency.

### *D. Transformer Engine Performance*

**`Te.Linear` analysis.** We extensively assess the Linear performance across diverse shapes, data types, and hardware setups (Fig. 4). Leveraging the Transformer Engine, we expedit matrix multiplications for the Linear layer using FP8 Tensor Cores. Our findings reveal an increase in GPU utilization and throughput with larger matrix sizes. FP8 performance is influenced by the overhead from data format conversion and quantization operators. For smaller matrix sizes, FP8 throughput is lower compared to FP16 or FP32. However, with $N$=8192, FP8's performance gains become evident. When $N$=16384, H800 and 4090 utilizing FP8 achieve almost twice the throughput of FP16. This underscores FP8's high throughput potential but underlines the need for specific conditions to attain optimal computing density.

**`Te.TransformerLayer` analysis.** The Transformer Engine condenses the entire Transformer Layer structure into `te.TransformerLayer`. Fig. 5 illustrates the latency for the same input text, offering a performance comparison across various hardware setups and data types with `te.TransformerLayer`. As computational density increases, the advantage of H800 in computation becomes evident. Notably, FP16 shows nearly twice the speed compared to FP32. FP8 outperforms FP16 for hidden_size>4096 but does not achieve double FP16 performance. This is because some modules within the Transformer Layer still do not utilize FP8 precision for calculations and data movement.

**LLM Inference Throughput Results.** We test the state-of-the-art decode-only models on inference with different data types, as shown in Table XII. We set the input length and text generation length to be relatively short, and the decode-only model is memory-bound during inference, so the computational advantages of FP8 Tensor Cores are not significant. Moreover, since the current Transformer Engine

TABLE X: Different *wgmma* instructions with different $N$ values on H800 tensor cores. The definitions of LAT and throughput can be found in the caption of Table VIII. “Rand” has the same latency as “Zero”.

| N | Dense | | | | Sparse | | | |
|---|---|---|---|---|---|---|---|---|
| | LAT/Throughput (SS,Zero) | LAT/Throughput (RS,Zero) | Throughput (SS,Rand) | Throughput (RS,Rand) | LAT/Throughput (SS,Zero) | LAT/Throughput (RS,Zero) | Throughput (SS,Rand) | Throughput (RS,Rand) |
| 256 | 128.0/728.5 | 128.0/731.9 | 665.4 | 667.5 | 144.0/1312.3 | 128.0/1476.2 | 1194.3 | 1277.5 |
| 128 | 64.0/728.5 | 64.0/725.4 | 659.8 | 661.7 | 80.0/1176.4 | 64.0/1463.3 | 1109.6 | 1270.5 |
| 64 | 32.0/719.6 | 32.0/719.7 | 648.3 | 649.9 | 48.0/977.4 | 32.0/1450.1 | 969.9 | 1263.4 |
| 32 | 24.0/477.3 | 16.0/710.3 | 471.5 | 634.4 | 32.0/727.1 | 18.0/1272.4 | 723.4 | 1135.7 |
| 16 | 20.0/287.0 | 13.0/434.2 | 283.5 | 426.2 | 24.0/482.3 | 18.0/638.6 | 479.8 | 636.3 |
| 8 | 18.0/158.2 | 13.0/216.7 | 157.6 | 215.2 | 20.0/289.0 | 16.0/359.4 | 286.1 | 356.7 |

TABLE XI: Power consumption and energy efficiency of maximum shape under mma instructions. T, D, and S represent Type, Dense, and Sparse respectively. P stands for energy, measured in Watts. E stands for efficiency, measured in TFLOPS/watt.

| A/B | C/D | T | A100 | | H800 | | 4090 | |
|---|---|---|---|---|---|---|---|---|
| | | | P | E | P | E | P | E |
| FP16 | FP16 | D | 173.4 | 1.79 | 188.6 | 2.62 | 189.1 | 1.89 |
| | | S | 198.8 | 3.13 | 187.2 | 3.86 | 214.0 | 3.33 |
| FP16 | FP32 | D | 188.5 | 1.61 | 196.7 | 2.49 | 154.1 | 1.16 |
| | | S | 216.1 | 2.79 | 194.9 | 3.70 | 165.9 | 2.15 |
| TF32 | FP32 | D | 214.7 | 0.71 | 254.9 | 0.97 | 174.3 | 0.51 |
| | | S | 235.7 | 1.28 | 232.5 | 1.56 | 187.9 | 0.95 |
| INT8 | INT32 | D | 178.4 | 3.41 | 165.3 | 5.92 | 201.4 | 3.53 |
| | | S | 193.9 | 6.24 | 163.3 | 8.79 | 219.8 | 6.47 |

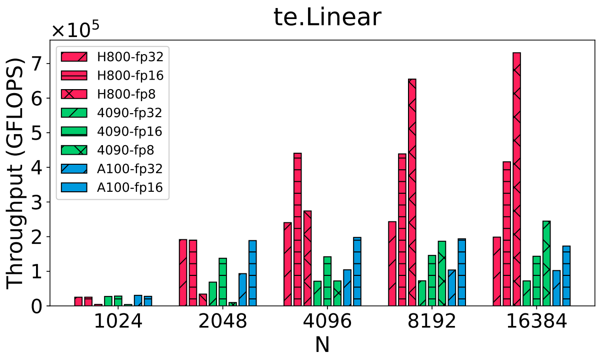

Fig. 4: Comparison of throughput for matrix multiplication with two same-size matrices $D(N*N) = A(N*N) \times B(N*N)$ in different hardware configurations and data types using `te.Linear`.

does not provide comprehensive support, data transmission between modules still occurs in FP16/FP32 without operator fusion. It is possible that when the model size and input data length increase, and with good operator fusion support, a certain improvement can be achieved.

### *E. New Features of Hopper*

**DPX.** Fig. 6 and Fig. 7 show the latency and throughput of the DPX functions on three tested GPUs. Since the DPX of RTX4090 and A100 are software emulation, their performance is almost the same. What can be observed is that for `relu` instructions, the performance of H800 is significantly better than the other two. For 16-bit operations, H800 also has significant acceleration, up to 13 times.

DPX on Hopper
is a hardware

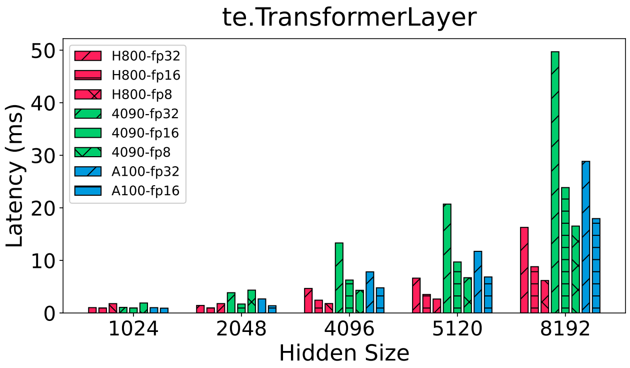

Fig. 5: Comparison of latency for the same input text in different hardware configurations and data types using `te.TransformerLayer`.

TABLE XII: Inference Throughput (Tokens / s) for different model sizes on different GPUs and different data types

| GPU | Model | FP32 | BF16 | FP8 |
|---|---|---|---|---|
| 4090 | llama-3B | 414.08 | 425.19 | 429.31 |
| | llama-2-7B | OOM | 350.69 | OOM |
| A100 | llama-3B | 674.50 | 670.87 | - |
| | llama-2-7B | 400.88 | 548.57 | - |
| | llama-2-13B | OOM | 420.81 | - |
| H800 | llama-3B | 679.45 | 624.10 | 537.92 |
| | llama-2-7B | 568.91 | 502.65 | 474.42 |
| | llama-2-13B | 357.57 | 399.38 | 356.11 |

However, not all functions have acceleration effects on Hopper. For some simple operations (e.g. `__viaddmax_s32`, which accepts 3 signed integers `(s1, s2, s3)` and returns `max(s1+s2,s3))`, we find that the performance of the three devices is close. In fact, by observing the SASS code, we find that new instructions (`VIMNMX`) were used on Hopper. Compared with previous `IMNMX`, performance does not seem to improve significantly. But in general, the Hopper architecture with DPX hardware acceleration has better performance than the previous generation architecture.

Additionally, `__vibmax_s32` data is not available on RTX4090 and A100. The reason is that compilation optimization optimizes this function into a max instruction. If we want to prevent this optimization, throughput measurements will be greatly affected.

Another finding is that on H800, when the number of launched blocks is less than the number of SMs, the throughput of DPX functions is proportional to the number of blocks. When the number of blocks just exceeds an integral multiple of the number of SMs, the throughput plummets, gradually returning to the maximum level as the number of blocks increases. Maximum throughput occurs when the number of blocks is an integer multiple of the number of SMs. Therefore, we have enough reason to infer that the DPX acceleration unit is located at the SM level.

Wave Quantization

**Asynchronous Data Movement.** Tables XIII and XIV illustrate the throughput comparison between *AsyncPipe* and *SyncShare* implementations on H800 and A100, respectively. Notably, *AsyncPipe* generally outperforms *SyncShare* with smaller block sizes (e.g., 8×8 and 16×16) on both GPUs. For instance, at a block size of 8×8, *AsyncPipe* shows an average performance improvement of 39.5% on H800 and 19.6% on A100. The reason is that under small block sizes, insufficient warp numbers hinder hiding synchronous shared memory copy latency. On the contrary, the two-stage pipeline in *AsyncPipe* allows simultaneous data movement and computation across different stages. However, as block size increases, the benefits diminish. Even with a block size of 32×32 on H800, the throughputs of *AsyncPipe* are often worse than those of *SyncShare*. Larger block sizes result in high warp concurrency, effectively concealing shared memory copy latency.

TABLE XIII: Benchmarking results of *globalToShmemAsyncCopy* on H800. The value of each cell represents the computational throughput measured in GFlops/s.

| block size: 8×8 | | | | | | | |
|---|---|---|---|---|---|---|---|
| Blocks/SM | 1 | 2 | 4 | 8 | 16 | 32 | Perf↑ |
| AsyncPipe | 516.69 | 998.45 | 1808.5 | 2931.29 | 3315.38 | 3615.99 | 39.5% |
| SyncShare | 327.86 | 646.58 | 1191.48 | 2117.56 | 2736.06 | 2861.75 | |
| **block size: 16×16** | | | | | | | |
| Blocks/SM | 1 | 2 | 4 | 8 | 16 | 32 | Perf↑ |
| AsyncPipe | 2650.06 | 4531.02 | 5038.26 | 5510.76 | 5728.71 | 5929.61 | 9.7% |
| SyncShare | 2372.41 | 3821.71 | 4713.84 | 5147.53 | 5309.23 | 5512.41 | |
| **block size: 32×32** | | | | | | | |
| Blocks/SM | 1 | 2 | 4 | 8 | 16 | 32 | Perf↑ |
| AsyncPipe | 5570.17 | 6112.92 | 6372.73 | 6496.21 | 6592.66 | 6592.87 | -1.8% |
| SyncShare | 5782.03 | 6280.8 | 6465.53 | 6600.58 | 6649.46 | 6631.11 | |

TABLE XIV: Benchmarking results of *globalToShmemAsyncCopy* on A100. The value of each cell represents the computational throughput measured in GFlops/s.

| block size: 8×8 | | | | | | | |
|---|---|---|---|---|---|---|---|
| Blocks/SM | 1 | 2 | 4 | 8 | 16 | 32 | Perf↑ |
| AsyncPipe | 379.03 | 798.5 | 1544.15 | 2429.93 | 2825.64 | 2888.84 | 19.6% |
| SyncShare | 379.03 | 742.93 | 1325.88 | 1982.38 | 2112.6 | 2256.17 | |
| **block size: 16×16** | | | | | | | |
| Blocks/SM | 1 | 2 | 4 | 8 | 16 | 32 | Perf↑ |
| AsyncPipe | 2198.21 | 2566.83 | 3821.09 | 4205.72 | 4413.69 | 4527.82 | 4.9% |
| SyncShare | 1754.73 | 2974.9 | 3724.42 | 4015.96 | 4207.57 | 4316.63 | |
| **block size: 32×32** | | | | | | | |
| Blocks/SM | 1 | 2 | 4 | 8 | 16 | 32 | Perf↑ |
| AsyncPipe | 4453.52 | 4863.73 | 5020.21 | 5106.74 | 5150.78 | 5129.68 | 1.7% |
| SyncShare | 4428.55 | 4917.25 | 5024.77 | 5025.45 | 4996.66 | 5028.47 | |

**Distributed Shared Memory.** SM-to-SM network latency is 180 cycles, a 32% reduction compared to L2 cache. This validates the advantages of the network, facilitating efficient data exchange from producers to consumers.

In Fig. 8, SM-to-SM throughput is illustrated for varying cluster and block sizes. As typically observed in similar benchmarks, larger block sizes and more parallelizable instructions result in higher throughputs. A peak throughput of nearly 3.27 TB/s is observed with a cluster size of 2, reducing to 2.65 TB/s with a cluster size of 4. Interestingly, as more blocks in the cluster compete for SM-to-SM bandwidth, the overall throughput gets lower and lower. While a larger cluster size can reduce data movement latency for more blocks, it intensifies throughput competition. Balancing this tradeoff by selecting optimal block and cluster sizes is an important direction for exploration.

Fig. 9 displays the histogram throughput with distributed shared memory. First, the optimal cluster size differs for various block sizes (CS=4 for block size 128, CS=2 for block size 512). Increasing block and cluster sizes can saturate SM-to-SM network utilization, potentially degrading overall performance due to resource contention. Second, a notable performance drop occurs from 1024 to 2048 *Nbins* when CS=1. Larger *Nbins* demand more shared memory space and limit active block numbers on an SM. Employing the cluster mechanism to divide *Nbins* within the same cluster enhances block concurrency, mitigating this issue. Lastly, although shared memory is not a limiting factor for active block numbers with block size = 512, choosing an appropriate cluster size ease the on-chip shared memory traffic by leveraging the SM-to-SM network resource, ultimately improving overall performance.

## V. Conclusion

This paper delves into memory hierarchy and tensor core performance of the newest three Nvidia GPU architectures using instruction-level benchmarks. We found that the hopper architecture shows advantages in both memory bandwidth and tensor core that are consistent with official claims. It is worth noting that on tensor core, we need to use the latest wgmma instructions to take advantage of all the performance of the fourth generation tensor core. We analyze AI performance across diverse architectures at library and application levels, emphasizing the impact of varied precisions. Experiments show that when the operation scale is relatively large, low-precision data types will show greater advantages. Additionally, we explore key features of the Hopper architecture: DPX, asynchronous data movement, and distributed shared memory. Our research enhances comprehension of the latest architecture's traits and performance, aiding in optimized algorithm design and application performance.

## Acknowledgments

This research was also supported by the National Natural Science Foundation of China (No. 62302126), the Shenzhen Science and Technology Program (No. RCBS20221008093125065, No. JCYJ20220818102414030, No. JSGGKQTD20221101115655027), a Hong Kong RGC

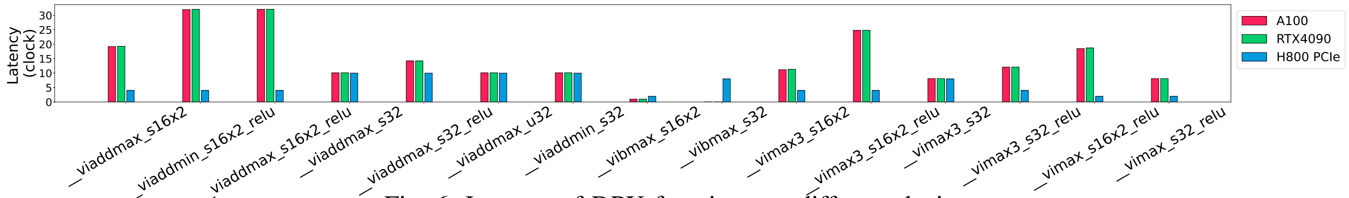

Fig. 6: Latency of DPX functions on different devices

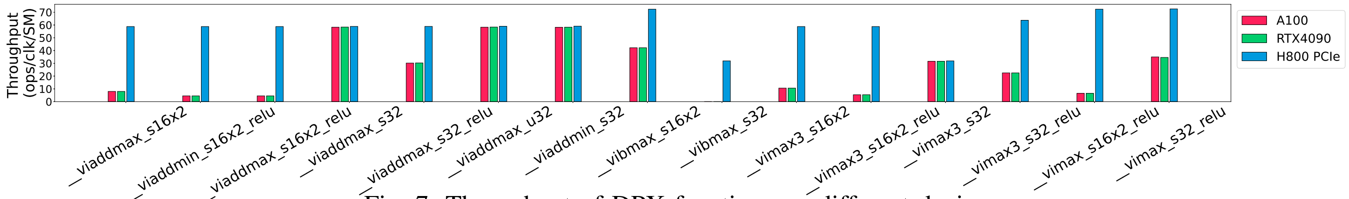

Fig. 7: Throughput of DPX functions on different devices

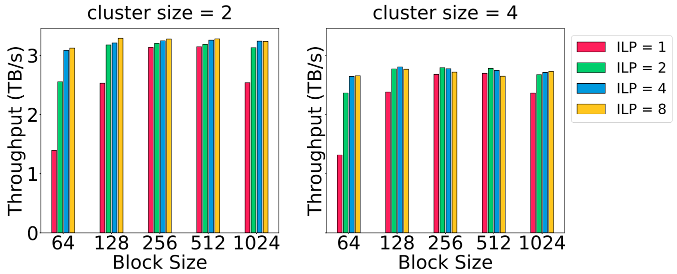

Fig. 8: The data communication throughput of the SM-to-SM network. "ILP" refers to the number of parallelizable data movement instructions.

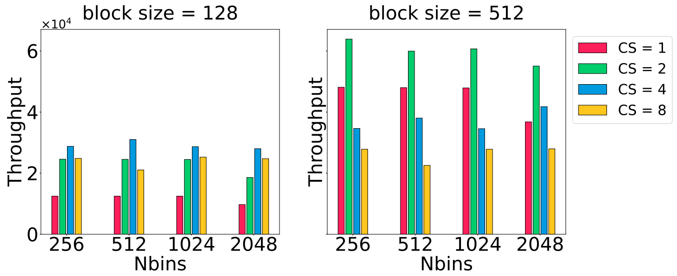

Fig. 9: Performance of the histogram application with distributed shared memory. The throughput is measured by the number of processing elements per second. "CS" refers to the cluster size. "Nbins" refers to the number of histogram bins.

RIF grant under the contract R6021-20, and Hong Kong RGC CRF grants under the contracts C7004-22G and C1029-22G.

## References

[1] L. Floridi and M. Chiriatti, "Gpt-3: Its nature, scope, limits, and consequences," *Minds and Machines*, vol. 30, no. 4, pp. 681–694, 2020.

[2] W. Sun, A. Li, T. Geng, S. Stuijk, and H. Corporaal, "Dissecting tensor cores via microbenchmarks: Latency, throughput and numeric behaviors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 1, pp. 246–261, 2023.

[3] S. Hong and H. Kim, "An integrated GPU power and performance model," in *International Symposium on Computer Architecture (ISCA)*, 2010.

[4] L. Braun, S. Nikas, C. Song, V. Heuveline, and H. Fröning, "A simple model for portable and fast prediction of execution time and power consumption of GPU kernels," *ACM Transactions Architecture and Code Optimization*, vol. 18, no. 1, dec 2021.

[5] X. Wang, K. Huang, A. Knoll, and X. Qian, "A hybrid framework for fast and accurate GPU performance estimation through source-level analysis and trace-based simulation," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 506–518.

[6] Q. Wang and X. Chu, "GPGPU performance estimation with core and memory frequency scaling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 12, pp. 2865–2881, 2020.

[7] X. Mei, Q. Wang, and X. Chu, "A survey and measurement study of gpu dvfs on energy conservation," *Digital Communications and Networks*, vol. 3, no. 2, pp. 89–100, 2017.

[8] Y. Arafa, A. ElWazir, A. ElKanishy, Y. Aly, A. Elsayed, A.-H. Badawy, G. Chennupati, S. Eidenbenz, and N. Santhi, "Verified instruction-level energy consumption measurement for nvidia gpus," in *Proceedings of the 17th ACM International Conference on Computing Frontiers*, 2020, pp. 60–70.

[9] R. van Stigt, S. N. Swatman, and A.-L. Varbanescu, "Isolating gpu architectural features using parallelism-aware microbenchmarks," in *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, 2022, pp. 77–88.

[10] Y. Arafa, A.-H. A. Badawy, G. Chennupati, N. Santhi, and S. Eidenbenz, "Low overhead instruction latency characterization for nvidia gpgpus," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2019, pp. 1–8.

[11] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-sim: An extensible simulation framework for validated GPU modeling," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 473–486.

[12] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *2009 IEEE international symposium on performance analysis of systems and software*. IEEE, 2009, pp. 163–174.

[13] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: Enabling energy optimizations in GPGPUs," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13, 2013, p. 487–498.

[14] N.-M. Ho and W.-F. Wong, "Exploiting half precision arithmetic in nvidia gpus," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2017, pp. 1–7.

[15] D. Yan, W. Wang, and X. Chu, "Optimizing batched Winograd convolution on GPUs," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '20, 2020, p. 32–44.

[16] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying gpu microarchitecture through microbenchmarking," in *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE, 2010, pp. 235–246.

[17] X. Mei, K. Zhao, C. Liu, and X. Chu, “Benchmarking the memory hierarchy of modern gpus,” in *Network and Parallel Computing: 11th IFIP WG 10.3 International Conference, NPC 2014, Ilan, Taiwan, September 18-20, 2014. Proceedings 11*. Springer, 2014, pp. 144–156.

[18] X. Mei and X. Chu, “Dissecting GPU memory hierarchy through microbenchmarking,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 72–86, 2017.

[19] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, “Dissecting the NVIDIA Volta GPU architecture via microbenchmarking,” *arXiv preprint arXiv:1804.06826*, 2018.

[20] Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza, “Dissecting the NVidia Turing T4 GPU via microbenchmarking,” *arXiv preprint arXiv:1903.07486*, 2019.

[21] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter, “NVIDIA tensor core programmability, performance & precision,” in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2018, pp. 522–531.

[22] M. Martineau, P. Atkinson, and S. McIntosh-Smith, “Benchmarking the NVIDIA V100 GPU and tensor cores,” in *Euro-Par 2018: Parallel Processing Workshops*. Cham: Springer International Publishing, 2019, pp. 444–455.

[23] D. Yan, W. Wang, and X. Chu, “Demystifying tensor cores to optimize half-precision matrix multiply,” in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 634–643.

[24] M. A. Raihan, N. Goli, and T. M. Aamodt, “Modeling deep learning accelerator enabled GPUs,” in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019, pp. 79–92.

[25] M. Fasi, N. J. Higham, M. Mikaitis, and S. Pranesh, “Numerical behavior of NVIDIA tensor cores,” *PeerJ Computer Science*, vol. 7, p. e330, 2021.

[26] Z. Tang, Y. Wang, Q. Wang, and X. Chu, “The impact of gpu dvfs on the energy and performance of deep learning: An empirical study,” in *Proceedings of the Tenth ACM International Conference on Future Energy Systems*, 2019, pp. 315–325.

[27] Y. Wang, Q. Wang, S. Shi, X. He, Z. Tang, K. Zhao, and X. Chu, “Benchmarking the performance and energy efficiency of ai accelerators for ai training,” in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 2020, pp. 744–751.

[28] R. Saavedra and A. Smith, “Measuring cache and TLB performance and their effect on benchmark runtimes,” *IEEE Transactions on Computers*, vol. 44, no. 10, p. 1223–1235, Jan 1995. [Online]. Available: http://dx.doi.org/10.1109/12.467697

[29] R. Saavedra-Barrera, “CPU performance evaluation and execution time prediction using narrow spectrum benchmarking,” Jan 1992.

[30] N. Corporation. (2023) CUDA documentation: Parallel thread execution. [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html

[31] NVIDIA. (2022) Transformerengine. [Online]. Available: https://github.com/NVIDIA/TransformerEngine

[32] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” *Advances in neural information processing systems*, vol. 30, 2017.

[33] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, “Pytorch: An imperative style, high-performance deep learning library,” *Advances in neural information processing systems*, vol. 32, 2019.

[34] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, “Flashattention: Fast and memory-efficient exact attention with IO-awareness,” *Advances in Neural Information Processing Systems*, vol. 35, pp. 16 344–16 359, 2022.

[35] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, “Llama: Open and efficient foundation language models,” *arXiv preprint arXiv:2302.13971*, 2023.

[36] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, “Llama 2: Open foundation and fine-tuned chat models,” *arXiv preprint arXiv:2307.09288*, 2023.

[37] N. Shazeer, “GLU variants improve transformer.” *arXiv: Learning,arXiv: Learning*, Feb 2020.

[38] B. Zhang and R. Sennrich, “Root mean square layer normalization,” *Neural Information Processing Systems,Neural Information Processing Systems*, Dec 2019.

[39] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, “Efficient memory management for large language model serving with pagedattention,” in *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[40] OpenAI. Introducing ChatGPT. [Online]. Available: https://openai.com/blog/chatgpt