Efficiently Emulating High-Bitwidth Computation with Low-Bitwidth Hardware

Zixuan Ma
ma-zx19@mails.tsinghua.edu.cn
Tsinghua University
Beijing, China

Haojie Wang
wanghaojie@tsinghua.edu.cn
Tsinghua University
Beijing, China

Guanyu Feng
fgy18@mails.tsinghua.edu.cn
Tsinghua University
Beijing, China

Chen Zhang
zhang-c21@mails.tsinghua.edu.cn
Tsinghua University
Beijing, China

Lei Xie
xie-l18@mails.tsinghua.edu.cn
Tsinghua University
Beijing, China

Jiaao He
hja20@mails.tsinghua.edu.cn
Tsinghua University
Beijing, China

Shengqi Chen
csq20@mails.tsinghua.edu.cn
Tsinghua University
Beijing, China

Jidong Zhai
zhaijidong@tsinghua.edu.cn
Tsinghua University
Beijing, China

ABSTRACT

Domain-Specific Accelerators (DSAs) are being rapidly developed to support high-performance domain-specific computation. Although DSAs provide massive computation capability, they often only support limited native data types. To mitigate this problem, previous works have explored software emulation for certain data types, which provides some compensation for hardware limitations. However, how to efficiently design more emulated data types and choose a high-performance one without hurting correctness or precision for a given application still remains an open problem.

To address these challenges, we present Ape, which can 1) provide different strategies for emulating high-bitwidth data types using native data types with in-depth error analysis; 2) dynamically and automatically select proper data types and generate efficient code for a given computation in fine-granularity to achieve higher performance while maintaining both correctness and precision at the same time without human efforts. We implement Ape on both NVIDIA Tensor Core and Huawei Ascend. Results show that Ape can boost General Matrix Multiplication and convolution by up to 3.12× and 1.86× on Tensor Core over CUDA Core and accelerate various applications by up to 1.78× (1.65× on average).

CCS CONCEPTS

• **Computing methodologies** → **Parallel algorithms**.

KEYWORDS

Domain Specific Accelerator, Emulation, Tensor Core

ICS '22, June 28–30, 2022, Virtual Event, USA

ACM ISBN 978-1-4503-9281-5/22/06.
https://doi.org/10.1145/3524059.3532377

ACM Reference Format:
Zixuan Ma, Haojie Wang, Guanyu Feng, Chen Zhang, Lei Xie, Jiaao He, Shengqi Chen, and Jidong Zhai. 2022. Efficiently Emulating High-Bitwidth Computation with Low-Bitwidth Hardware. In *2022 International Conference on Supercomputing (ICS '22), June 28–30, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3524059.3532377

1 INTRODUCTION

Domain-Specific Accelerators (DSAs) have been increasingly developed in recent years [17, 20, 21]. To achieve high performance, most DSAs only support specific computation types based on a limited set of data types. These restrictions largely limit the extensive usage of DSAs. For example, NVIDIA introduced a widely known DSA called GPU Tensor Core [32]. Tensor Core can provide giant performance leaps on General Matrix Multiply (GEMM) computation over traditional CUDA Core (same GPU without using Tensor Core) for half-precision and double-precision floating point data types. However, Tensor Core does not support widely-used single-precision floating point data types (i.e., FP32). Therefore, users have to fall back to traditional CUDA Core or use double-precision floating point (FP64) Tensor Core instead when computing with FP32 type, hence hindering the huge performance boost brought by Tensor Core.

To mitigate this problem, previous works [11, 12, 29] have studied the feasibility of emulating high-bitwidth types using low-bitwidth types, e.g., emulating FP32 using FP16. These works can extend the usage of DSAs and enable better utilization of the computation capability. But these works do not take an in-depth error analysis of emulated data types, thus missing potential optimization opportunities in both terms of representation capability (representation range and precision) and performance. In fact, by carefully designing emulation algorithms, we can enable DSAs to support more data types besides limited native data types, including floating point data types and integer data types, as well as benefit from large performance enhancement. As more native data types are supported on DSAs, how to design more emulated data types and efficiently select a

Table 1: Native floating-point and emulated data types on NVIDIA A100 with their corresponding precision specifications (number of bits) and peak performances (TFLOPS).

	Native data types					Emulated data types			
	FP64	FP32	FP16	TF32	BF16	FP[11, 29]	FP32-F	FP32-T	FP32-B
Exponent	11	8	5	8	8	5*	5	8	8
Significand	52	23	10	10	7	21†	22	22	23
CUDA Core	9.7	19.5	78	N/A	39	-	-	-	-
Tensor Core	19.5	N/A	312	156	312	78	104	52	52

correct yet high-performance data type for a given data type still remains an open problem.

We take the floating point types on NVIDIA Tesla A100 Tensor Core GPU [32] as an example to give a more concrete illustration. Left part of Table 1 lists native floating-point data types with precision specifications and performance. The main difference between these data types is the bitwidth of exponent and significand. FP32 is a native data type with representation range of $[2^{-126}, 2^{+127}]$ and precision of 2^{-23}, and its peak performance is only 19.5 TFLOPS on CUDA Core. Previous works [11, 29] emulate FP32 using half-precision floating point (FP16), with representation range of $[2^{-2}, 2^{15}]$, precision of 2^{-21}, and peak performance of 78 TFLOPS. In our work, we take an in-depth analysis of numerical error and propose an optimized emulation method named FP32-F, which also emulates FP32 using FP16, but extends representation range, precision, and peak performance to $[2^{-14}, 2^{15}]$, 2^{-22} and 104 TFLOPS, respectively. Moreover, we design new emulation methods, FP32-T and FP32-B, which emulate FP32 using tensor floating point (TF32) [23] and brain floating point (BF16) [38], respectively. The error analysis also helps us optimize FP32-T and FP32-B for better representation capability and performance. The summarization of emulated data types is listed on the right of Table 1. Note that compared with native FP32, FP32-F provides a much narrower data representation range, and both FP32-F and FP32-T have 1-bit precision loss, while FP32-F has much higher performance than FP32-T and FP32-B.

To achieve efficient emulation on a DSA, we still face challenges of how to materialize various emulated data types. With both native and emulated data types of different representation capability and performance, we have more choices for a given application. During the execution of an application, the values of variables in a program are always changing. The requirements for the precision and data representation range of data types will also change. In addition, for a large data object, different regions of data may have diverse requirements for specific data types. Considering the dynamicity and space pattern of the input data, how to select the proper data type with low overhead is challenging.

To address these challenges, we design a lightweight emulation adapter with fine granularity. This adapter automatically analyzes data patterns for a given application at runtime and selects the most suitable type guaranteeing both correctness and performance with low overhead. The selection is performed on small blocks instead of the whole data space, which enables using mixed data types to fully utilize the computation capability for different data patterns.

Bringing up all together, we propose Ape, a system that presents several highly efficient data emulation methods for both floating point types and integer types, with a lightweight fine-granularity adapter to automatically select mixed data types from both native and emulated data types for a given application, to better utilize the computation capability of DSAs. Compared with previous works, our approach can enlarge the representation range of emulated data types and improve the computation performance while automatically choosing the most suitable data types. We implement Ape on both NVIDIA and Huawei accelerators, and support both floating point emulation and integer emulation, to verify our idea. Moreover, high-performance kernels with the support of different emulated data types are implemented and wrapped up into a BLAS-like library to make our system complete and available to end-users in different levels.

We make the following contributions in this work.

- We propose different algorithms to emulate high-bitwidth data types with low-bitwidth data types and perform in-depth analysis to optimize the representation capability and performance. Compared with previous works, our algorithms can expand data range, increase data precision, and significantly improve computation performance.
- We design a fine-grained lightweight data adaptation algorithm that can automatically select a proper data type according to the requirement of precision for a given application on specific hardware.
- We implement Ape on both NVIDIA Tensor Core GPUs and Huawei Ascend processors, supporting emulations of floating point and integer types, and provide a user-friendly API to facilitate porting legacy code with our library.
- The evaluation shows that Ape can achieve significant performance improvement. On NVIDIA Tesla A100 GPU, Ape can boost GEMM and convolution by up to 3.12× and 1.86×, and accelerate various applications by up to 1.77×.

2 BACKGROUND AND RELATED WORK

2.1 Domain-Specific Accelerator

In order to accelerate domain-specific applications, various Domain-Specific Accelerators (DSAs) are being designed. Compared with general-purpose chips, DSAs can achieve much higher performance on specific applications. For example, existing DSAs, including NVIDIA Tensor Core [33], Google Tensor Processing Unit (TPU) [21], and Huawei Ascend AI processor [25], are designed for AI applications, which mainly focus on linear operators with low-bitwidth data types like half-precision floating-point or low-bitwidth integer types.

NVIDIA Tensor Core NVIDIA Tensor Core is first introduced in NVIDIA Volta [33] GPUs. Tensor Core can provide powerful computing capability for Matrix Multiply-Accumulate (MMA) operations. For example, Tensor Core in Tesla A100 GPU has a floating-point computation throughput of up to 312 TFLOPS. Figure 1 shows half-precision floating point computation performed on Tensor Core: Tensor Cores read two half-precision input data and perform

*This emulated type presented by previous works has 5 exponent bits, but it only has a range of $[-2, +15]$. This range is much smaller than that can be represented by 5 exponents bit, i.e., $[-14, +15]$.

†[11] demonstrates that its emulation method has 21 significand bits, and [29]'s emulation method has 20 significand bits.

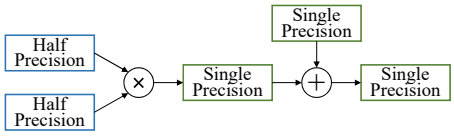

Figure 1: MMA operation on Tensor Core.

a half-precision matrix multiplication without loss of precision. Intermediate results are stored in single-precision registers, and accumulation is also performed with single-precision. Both No.2 and No.3 supercomputers on the Top500 [9] list, i.e., Summit and Sierra [37] are equipped with NVIDIA Tensor Core GPUs.

Huawei Ascend AI Processor Huawei Ascend AI processors are designed for AI applications, supporting both training and inference. We take Ascend 910A as an example. Ascend 910A has a peak performance of 256 TFLOPS on FP16 and 512 TOPS on INT8. 910A is equipped with 32 GB HBM, with a bandwidth of 1228 GB/s. Ascend 910A does not support FP64 operations. Peng Cheng Cloud Brain II [34] is a large AI computing platform equipped with 4,096 Ascend 910A, with a peak performance of 1 EFLOPS with FP16.

2.2 Low-Bitwidth Data Types

Floating-Point Data Types FP16 (IEEE 754 Half-precision Floating Point) [1], BF16 (Brain Floating Point) [38], and TF32 (Tensor Floating Point) [23] are commonly supported by DSAs. As shown in Figure 2, FP16 has 5 exponent bits and 10 significand bits. Compared with FP32, FP16 has a much narrower data range and lower precision. NVIDIA Tensor Core and Ascend processor support this type. BF16 is proposed by Google Brain, which has 8 exponent bits and 7 significand bits and can express floating point numbers with a similar range as FP32 but with much lower precision. NVIDIA Ampere GPUs and TPUs support this type. TF32 is proposed by NVIDIA. TF32 has 8 exponent bits and 10 significand bits, expressing floating point numbers with a similar range as FP32 and the same precision as FP16 but occupies the same storage space as FP32 (4 bytes). Now, only Nvidia Ampere GPUs support this data type.

Fixed-Point Data Types Fixed-point is another method to represent a real number, usually implemented by integers. Low-bitwidth fixed-point data types are commonly used in the AI domain. Not only for inference [22, 24] but training [18, 26], low-bitwidth datatype can reduce memory usage and improve performance without losing much accuracy. To meet the requirement of fixed-point, existing DSAs support acceleration for various integer types. For example, Edge TPU [15] and Ascend support INT8. NVIDIA Tensor Core on Ampere GPU supports INT8, INT4, and INT1. All these DSAs use INT32 to accumulate the production of these data types to avoid overflow on specific computation.

2.3 Related Work

Datatype Extension Several previous works [7, 16, 27, 36] emulate high-bitwidth data types with low-bitwidth types on GPUs and IoT devices. Due to lack of hardware support, they need to issue dozens of low-bitwidth instructions to emulate high-bitwidth operators, resulting in inefficiency. For example, Thall et al. [36] use 20 single-precision instructions on GPUs to emulate a double-precision addition.

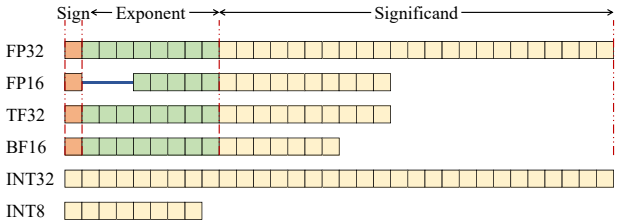

Figure 2: A comparison of different data types.

Markidis et al. [29] leverage FP32 accumulation on Tensor Core to emulate a single-precision GEMM based on four half-precision (FP16) GEMM. However, as described in Section 4.1, their emulated type only supports a limited range, i.e., $[0.25, 6.55 \times 10^4]$. EGEMM-TC [11] highly optimized Markidis's approach on Turing architecture by using SASS assembly-style instructions. However, EGEMM-TC also suffers the range limitation of Markidis's approach. Meanwhile, the SASS-based implementation cannot be directly used on other GPUs such as NVIDIA Tesla V100 and A100, which limits its usage. Our system provides a larger data representation range for FP16 emulated single-precision type with significant performance improvement, along with other emulated data types with an auto-adaptive approach.

Datatype Adaptation The trade-off of accuracy and performance makes selecting a proper data type necessary. Some recent efforts [5, 6, 35] focus on rigorous floating-point error analysis to down-cast precision for better performance while incurring a given error bound across all program inputs [5]. Although these works provide sufficient precision for any input, they are limited by lengthy analysis due to their static and formal methods. For example, the state-of-the-art system, Satire [6] takes about 10 minutes to complete analyzing a matrix multiplication on 128×128 matrices. Instead of costly analysis, our approach is lightweight and focuses on selecting a proper type based on representation ranges of floating-point data types at runtime.

3 OVERVIEW OF APE

3.1 APE Framework

APE is designed as a data type emulator and performance booster that can emulate high-bitwidth data types using native data types and automatically select data types with the best performance without hurting accuracy. By dynamically partitioning and analyzing input data for a given program, APE can optimize computation by selecting a proper data type for each fine-grained data block, thus improving computation efficiency.

Figure 3 gives an overview of APE. APE consists of two major modules, **Emulator** and **Adapter**. The emulator provides all available native and emulated data types of a DSA. The adapter is in charge of selecting a mixed data type, including both native and

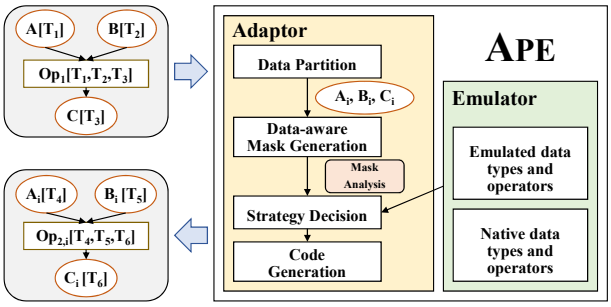

Figure 3: APE overview.

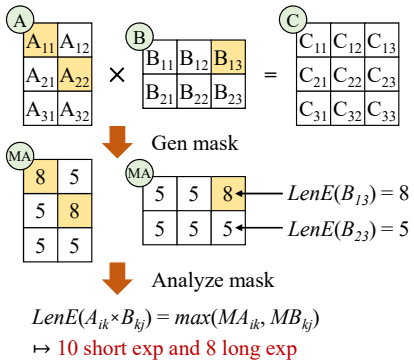

Figure 4: An example of APE. $LenE(X)$ indicates maximum length of exponent bits of all elements for given matrix X.

emulated data types, for a given application, to better utilize the computation capability of a DSA.

For a given computation Op_1, with an input A of data type T_1, an input B of data type T_2, and an output C of data type T_3 (for simplification, we assume any computation has two inputs and one output), APE will first partition input and output data into small blocks, denoted by A_i, B_i, and C_i, respectively. It is guaranteed that the input blocks are the computation dependency of the output block, e.g., A_i Op_1 B_i is part of the computation of C_i. Then APE will generate a mask matrix for each input recording the maximum length of exponent bits for elements in each input block. Then, the mask matrix will be further analyzed to guide the strategy decision to select the most proper data type and the corresponding computation from *Emulator*. Finally, APE generates an optimized code for computations on each block. In this example, APE generates a computation of $Op_{2,i}$ with an input A_i of data type T_4, an input B_i of data type T_5, and an output C_i of data type T_6.

3.2 Example

To give a more concrete picture of how APE works, we take a GEMM computation as an example, as shown in Figure 4. Given a GEMM computation $C = A \times B$, we first partition data into several small blocks. In this case, we suppose that APE partitions A into 6 (3×2) blocks, B into 6 (2×3) blocks, and C into 9 (3×3) blocks correspondingly. APE will analyze the input blocks and indicates that at least one value that must be expressed with 8 exponent bits in A_{11}, A_{22}, and B_{13}, while all the values in other input blocks can be expressed with less than 5 exponent bits. Thus we generate their corresponding mask matrices MA and MB. Each element of MA and MB indicates the minimum length of exponent bits to be used for this block. Considering the computation pattern of GEMM, each block in A, namely A_{ik}, will perform block-level GEMM computations with certain blocks in B, namely B_{kj}. After analyzing the mask matrices, APE decides that 8 out of the total 18 block-level GEMM computations should be computed in a data type with long exponent bits, while the others can be computed with short exponent bits. Finally, APE generates the computation for this case and selects a proper strategy to achieve higher performance.

Table 2: Representation Capability of different emulated data types and their performance on NVIDIA A100 GPU.

	Type	Precision	Exponent Range	Sustained Performance
Native	FP32	23 bits	$[-126, +127]$	17.28 TFLOPS
Previous Works	FP32-M [29]	N/A	$[-2, +15]$	54.10 TFLOPS
	EGEMM-TC [11]	21 bits	$[-2, +15]$	Unsupported
Ours	FP32-F	22 bits	$[-14, +15]$	64.15 TFLOPS
	FP32-T	22 bits	$[-114, +127]$	18.94 TFLOPS
	FP32-B	23 bits	$[-110, +127]$	38.38 TFLOPS

```
pair<FP16, FP16> toFP32_F(FP32 A) {
    FP16 Hi_A = FP16(A);
    FP16 Lo_A = FP16(A - FP32(Hi_A));
    return {Hi_A, Lo_A};
}
```

Listing 1: Represent an FP32 using two FP16s.

4 EMULATING HIGH-BITWIDTH DATA TYPES

In this section, we illustrate our approach to emulating a high-bitwidth data type with different low-bitwidth data types and analyze these emulated data types in detail. For each emulated data type, after analyzing its representation precision, range, and computation accuracy, we present two techniques to optimize the representation method. One is to reduce the exponent gap of two elements by shifting so as to make full use of exponent bits' capability. The other is to omit the computations that do not affect the result's precision. With these two optimizations, we can enlarge the data representation range and achieve much higher performance.

To simply demonstrate our method, we choose three kinds of widely supported half-precision data types, FP16, TF32, and BF16, to emulate the commonly used single-precision FP32 data type. We propose three emulated data types: FP32-F (Section 4.1), FP32-T (Section 4.2), and FP32-B (Section 4.3). Table 2 shows the representation capability and sustained performance of the previous work and ours. The BF16- and TF32-emulated data types are first proposed in this work. Although FP16-emulated data type has been introduced in previous works [11, 29], our emulation algorithm, FP32-F, has a larger representation range and benefits from a 1.33× theoretical speedup and 1.19× sustained speedup.

Moreover, in Section 4.4, we briefly discuss how our emulation algorithm can be applied on emulating integer types.

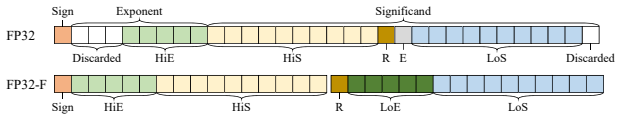

Figure 5: Emulation methods with FP32-F.

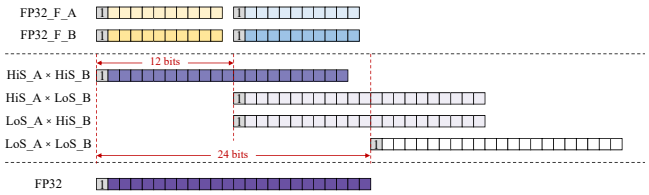

Figure 6: FP32-F multiplication and accumulation.

4.1 Emulating FP32 with FP16 (FP32-F)

The algorithm in Listing 1 describes the process of splitting an FP32 into two FP16s, which are denoted by *Hi* (higher field) and *Lo* (lower field). Since one FP16 only has 5 bits for the exponent, while FP32 includes 8 exponent bits, the highest 3 bits of FP32's exponent are discarded, and the remaining 5 bits are stored in *Hi*'s exponent bits as *HiE*. This process is similar to previous work [29]. Next, by analyzing numerical error, we can optimize the representation and computation to achieve a much larger representation range and higher performance.

Representation Precision Figure 5 shows the mapping of FP32 to FP32-F. As for FP32's 23 significand bits, they are split into several parts as follows.

- The first 10 bits (*HiS*) are stored in *Hi*'s significand bits.
- The 11th bit is called the **rounding bit**, denoted by *R*, and is rounded to the 10th bit using round-to-nearest-even.
- The first bit after the rounding bit that equals 1 is treated as an **encoding bit**, namely *E*, which can be implicitly encoded into the significand bits of *Lo*.
- The next 10 bits after the encoding bit are stored in *Lo*'s significand bits, denoted by *LoS*.

LoS can preserve all the trailing bits after the encoding bit if their length does not exceed 10. Otherwise, i.e., there are 11 trailing bits, the last bit of FP32's significand bits is discarded. For simplicity, we can assume the encoding bit is always the 12th significand bit of FP32 in Figure 5, Figure 6, and for the rest of this section. In this situation, where the 12th bit of an FP32's significand is 1, its last bit is discarded. Therefore, these two FP16s can only represent the first 22 bits out of the 23 significand bits of the FP32 number. As a result, the representation precision of FP32-F is 2^{-22}, one bit less than the precision of standard FP32.

Representation Range Exponent error occurs because FP16 only has 5 bits for the exponent, so the highest 3 exponent bits in FP32 are truncated. Therefore, if an FP32's exponent is larger than 15 or smaller than −14, it cannot be emulated with two FP16s. For simplification, the representation range in this paper refers to *the range of the absolute value* for a given type.

In FP32-F, the lower field *Lo* represents the lower part of the original FP32 number. Thus the exponent bits of *Lo*, denoted by *LoE*, equals *HiE* shifted by the index of encoding bit *E*, i.e., $LoE = HiE - 12$ in the example of Figure 5. With this representation, the rounding bit *R* equals the sign bit of *Lo*. When using FP32-F, an FP32 number *A* is split into two parts, *Hi_A* and *Lo_A*, each having a range of $[6.10 \times 10^{-5}, 6.55 \times 10^{4}]$. If we simply accumulate the results of multiply computations as previous works did, to keep the 23-bit precision, both Hi_A and Lo_A need to be in the range. This limits the range of the original FP32 number to $[0.25, 6.55 \times 10^{4}]$. As a result, the lower bound of the representation range is too large.

To settle the problem, *Lo_A* is scaled up by 4096×, i.e., shifting the exponent by 12 bits. Therefore, *Lo_A* can maintain the precision of *A* and an index equivalent to *Hi_A* at the same time. This patch enlarges the range of FP32-F to $[6.10 \times 10^{-5}, 6.55 \times 10^{4}]$.

```
1  FP32 FMA(FP32 C, FP32 A, FP32 B) {
2    auto [Hi_A, Lo_A] = toFP32_F(A);
3    auto [Hi_B, Lo_B] = toFP32_F(B);
4    C += Hi_A * Hi_B;
5    C += Hi_A * Lo_B + Lo_A * Hi_B;
6    // C += Lo_A * Lo_B; (no effect on C)
7    return C;
8  }
```

Listing 2: Compute FMA with FP32-F.

Computation Accuracy To demonstrate how Ape performs the computation emulation, we take Fused Multiplication Addition (FMA) computation as an example in this section.

Listing 2 shows how to multiply 2 FP32-F with the significand bits correctly processed. We denote the multiplier and multiplicand as *A* and *B*. Their *HiS* bits and *LoS* bits, when emulated by FP32-F, are denoted as *HiS_A*, *LoS_A*, *HiS_B*, and *LoS_B*.

As shown in Figure 6, for each multiplication $A \times B$, four FP32-F multiplications, $HiS\_A \times HiS\_B$, $HiS\_A \times LoS\_B$, $LoS\_A \times HiS\_B$, and $LoS\_A \times LoS\_B$, should be performed. Then, the results of these four FP16 multiplications are added together with shifting indicated by *LoE_A* and *LoE_B*. The result is the significand bits of the original FP32 result. Note that only 23 bits are kept due to the length of FP32's significand bits.

The exponent of the FP32 equals the sum of *Hi_A*'s exponent and *Hi_B*'s exponent, which is similar to the floating point multiplication, while only the lowest 5 bits of *A* and *B*'s exponent bits are kept. Note that as $Lo\_A \times Lo\_B$ is shifted by 12 + 12 bits, it has no effect on the final FP32 result. Therefore this step can be skipped to save computation and bring us 1.33× speedup compared to previous works theoretically.

Since the result of multiplying two FP16s is stored in an FP32, there is no overflow or underflow in the exponent bits of the result. As the accumulation is directly performed on FP32 fragments, no additional error is introduced by the emulated computation. The last significand bit of a standard FP32 result might not be precise in the result of FP32-F multiplication since the last significand bits of the two multipliers may be discarded. As mentioned in Section 4.1, the rest bits are precise, and the computation precision of FP32-F is 2^{-22}.

```
tuple<BF16, BF16, BF16> toFP32_B(FP32 A) {
  BF16 Hi_A = BF16(A);
  BF16 Mi_A = BF16(A - FP32(Hi_A));
  BF16 Lo_A = BF16(A - FP32(Hi_A) - FP32(Mi_A))
  return {Hi_A, Mi_A, Lo_A};
}
```

Listing 3: Represent an FP32 using three BF16s.

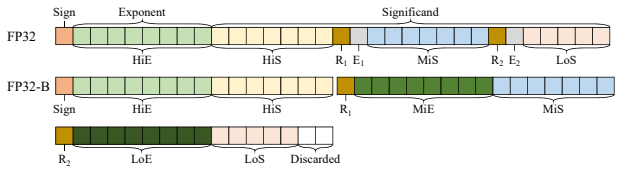

Figure 7: Emulation methods with FP32-B.

4.2 Emulating FP32 with TF32 (FP32-T)

Emulating an FP32 using two TF32s is first proposed in this work. The emulation is similar to that using two FP16s, with the only exception that TF32 has 8 bits for the exponent (same as FP32) and 10 bits for significand (same as FP16). The 10-bit significand makes the significand emulation of FP32-T the same as FP32-F. Similarly, its precision is 2^{-22}. Since TF32's exponent is the same as FP32's, *Lo* does not need to be scaled. Therefore, FP32-T has a range of $[4.81 \times 10^{-35}, 3.40 \times 10^{38}]$.

4.3 Emulating FP32 with BF16 (FP32-B)

We first propose a method of emulating an FP32 using BF16s in this work. Listing 3 shows the algorithm of this emulation. Still, the first step is to split the FP32 into three BF16s, denoted by *Hi*, *Mi*, and *Lo*, respectively.

Similar to FP32-F, the exponent bits of FP32 are stored in *Hi*'s exponent bits. Since BF16 has 8 bits for the exponent, FP32's exponent bits can be fully preserved. For the significand bits, since BF16 only has 7 bits for significand, three BF16s are needed to emulate one FP32.

Representation Precision Figure 7 shows the mapping of FP32 to FP32-B. The basic concept is similar with FP32-F, listed as follows.

- The first 7 bits of FP32's significand bits are stored in *Hi*, denoted by *HiS*.
- The 8th and 9th bits are **rounding bit** and **encoding bit**, denoted by R_1 and E_1, respectively. (Similarly, we assume that the 9th bit is 1.)
- The next 7 bits are stored in *Mi*, denoted by *MiS*.
- The next 2 bits are another rounding bit and another encoding bit respectively, denoted by R_2 and E_2.
- The last 5 bits are stored in *Lo*, denoted by *LoS*.

Since all significand bits of FP32 are completely stored in FP32-B, FP32-B has the same precision as FP32.

Representation Range As *Mi* and *Lo* are not scaled, the representation of *Mi* and *Lo* affects the range of FP32-B. As a result, FP32-B has a representation range of $[7.70 \times 10^{-34}, 3.40 \times 10^{38}]$.

```
FP32 FMA(FP32 C, FP32 A, FP32 B) {
  auto [HiA, MiA, LoA] = toFP32_B(A);
  auto [HiB, MiA, LoB] = toFP32_B(B);
  C += Hi_A * Hi_B;
  C += Hi_A * Mi_B + Mi_A * Hi_B;
  C += Hi_A * Lo_B + Mi_A * Mi_B + Lo_A * Hi_B;
  // C += Mi_A * Lo_B + Lo_A * Mi_B;
  // C += Lo_A * Lo_B;
  return C;
}
```

Listing 4: Compute FMA with FP32-B.

Computation Accuracy The FMA computation of FP32-B is similar to that of FP32-F. Still, we only introduce how Ape emulates the multiplication. Six multiplications, $Hi\_A \times Hi\_B$, $Hi\_A \times Mi\_B$, $Mi\_A \times Hi\_B$, $Hi\_A \times Lo\_B$, $Lo\_A \times Hi\_B$, and $Mi\_A \times Mi\_B$, are needed, as shown in Listing 4. The computation of $Mi\_A \times Lo\_B$, $Lo\_A \times Mi\_B$, and $Lo\_A \times Lo\_B$, are skipped, because they do not affect the result in precision, similar to $Lo\_A \times Lo\_B$ in FP32-F.

4.4 Emulating Integer Data Types

Since current DSAs support low-bitwidth integer types, including INT8, INT4, and INT1 with INT32 accumulators, emulating high-bitwidth integer data types with low-bitwidth data types is possible. For example, we implement emulated INT16 with two INT8 on NVIDIA Tesla A100 and Huawei Ascend 910A. Other integer types are similar.

The emulation method of data representation and computation is similar to floating-point data types. An INT16 is emulated by directly splitting it into two INT8s by byte-order. But there are some unique techniques in integers. To guarantee emulated integer data types can perform exactly the same as native data types, computation of emulated INT16 requires four INT8 computations, which we evaluate in Section 7.

5 ADAPTING APE TO REAL-WORLD HARDWARE

Although Section 4 provides approaches to emulate high-bitwidth data types with low-bitwidth data types, there still remain challenges when applying them on specific hardware.

The first challenge comes from the hardware constraints. The theory in Section 4 guarantees the mathematical errors for different emulation methods for element-wise computation. However, for a real-world DSA, the computation resource is limited and the computation granularity is not always element-wise. For example, on NVIDIA Tensor Core, the input data and output data have different storage bitwidth, and the computation is taken with a fused MMA instead of an FMA. Therefore, we need further analysis when applying the emulation methods to specific hardware, which will be detailed in Section 5.1.

The second challenge is how to achieve higher performance while also guaranteeing computation correctness. As shown in Table 2, different emulation methods have various performance and representation capabilities. Although FP32-F has the best performance, it may introduce overflow or underflow as it has a narrow representation range. Considering the real-world matrix data often

contain over thousands of elements and have disparate data patterns which can only be determined at runtime, choosing the correct yet highly efficient data types while preventing large runtime overhead is challenging. To address this, we propose a fine-granularity block-wise approach that enables hybrid emulation methods for a given computation (Section 5.2) and design a lightweight performance-model-based strategy decider to minimize the runtime overhead (Section 5.3).

5.1 Data Correctness under Hardware Constraints

For a specific operator, the numerical correctness is related to the computation patterns. In a real-world DSA, the computation is a fused computation that contains many steps. Taking GEMM for example, several multiplications are performed, and their results are summed up as the output. In this process, overflow may occur in every step, which affects the correctness. Note that a register, formally the **accumulator**, is used to store the partial sum of the multiplication results, which may be in a different data type from the input. Therefore, in the design of the adapter, the architecture of the target DSA needs to be carefully inspected and considered.

Taking Tensor Core for example, by examining the precision and range of MMA instruction, we find the following useful features.

- When using FP32 as the accumulator in an FP16 or BF16 matrix-multiplication, the result has the same accuracy as FP32 computation. In other words, it is exactly as the results of the original inputs.
- The accumulator of Tensor Core accumulates the result of multiplications and performs rounding in every MMA instruction. The rounding method is round-to-zero.

We can find that assuming that GEMM computation does not overflow or underflow FP32, no overflow or underflow will occur during the emulated computation. The rounding method of MMA instruction is different from FMA instruction on CUDA Core. That means the result on Tensor Core is not exactly the same as on CUDA Core, but both of them are correct numerically. Therefore, we can conclude that the emulation method of Ape can be implemented to such hardware whose accumulator supports at least the same bitwidth with the high-bitwidth data type.

Based on these facts, we can select the appropriate exponent by checking the range of the input matrix for GEMM computation. Therefore, we can potentially mine the dynamicity of the input data. We design an adapter that scans each input matrix for each GEMM computation and selects the proper data types.

5.2 Block-Wise Hybrid Computation

As mentioned in Table 2, there is a huge gap in performance between different types with similar precision. Taking GEMM on Tensor Core on NVIDIA A100 as an example, with matrices sized 1024×1024, FP32-F achieves 27.24 TFLOPS, while FP32-B can only achieve 17.20 TFLOPS, 63.1% the throughput of the former. Intuitively, to achieve higher throughput, FP32-F should be used as much as possible. However, the use of FP32-B is unavoidable when there are computations that require more exponent bits.

Moreover, data patterns typically vary in different areas of data. To mine such patterns, we propose a block-wise algorithm. We firstly partition the data into blocks according to the computation granularity of hardware. Then, by analyzing the data and computation dependency, we can determine the data type with the highest possible performance for each input block. In this way, we use different data types for different blocks within one computation for higher performance.

```
type_t getMask(Partition P) {
  type_t mask = FP32_F;
  for (auto v : P) {
    if (v < MIN_FP32_B || v > MAX_FP32_B)
      return FP32;
    if (v < MIN_FP32_F || v > MAX_FP32_F)
      mask = FP32_B;
  }
  return mask;
}
```

Listing 5: Generate the mask for a partition.

After partitioning, the process of analyzing the input data is shown in Listing 5. By checking each element in a partition, the related computation process is analyzed to find the most suitable type that can meet the needs of exponent bit length in the partition. For example, in our GEMM implementation on Tensor Core, after partitioning, a GPU kernel is used to scan each element in each partition and select the representation method with the best performance and sufficient capability. Afterward, statistics on the entire partitions are summarized to determine the most suitable data type for this partition, which is named as **mask**.

In this example, FP32-F and FP32-B are candidate types. For a given binary operator $C = \mathrm{op}(A, B)$, where A, B, and C are blocks, FP32-F, which has higher performance, can only be used when both A and B can be represented by FP32-F. Otherwise, FP32-B must be used.

Therefore, the computation is split into two parts. The blocks that must use FP32-B and the rest part are separately computed by two different kernels. The decision of the two kernels only involves looking up two labels. Thus, it can be made during runtime before starting the actual computation without introducing significant overhead.

As a result, Ape benefits from simultaneously using different types to balance capability and performance.

5.3 Light-Weight Strategy Decider

As mentioned in Section 5.1, by analyzing the data mask, a list of candidate types for computation is given. The most performant case is that all data can be represented by FP32-F, where FP32-F is directly used for computing. On the contrary, if the input data can only be represented by native FP32, we have to roll back to CUDA Core, leading to poor performance. In more common cases, part of the input data can be represented by FP32-F and the others by FP32-B. The block-wise hybrid computation method, as mentioned in Section 5.2, is utilized. But the hybrid method is not always the optimal choice. To clearly demonstrate this scene, we take a hybrid of FP32-F and FP32-B as an example to show the performance of FP32-F, FP32-B, and the hybrid approach. The result is shown in Figure 8. We can observe that the best data type changes with the growth of the proportion of FP32-B blocks. Thus, an online

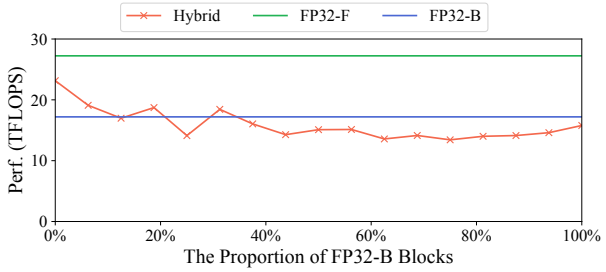

Figure 8: GEMM performance with different proportion of FP32-B blocks.

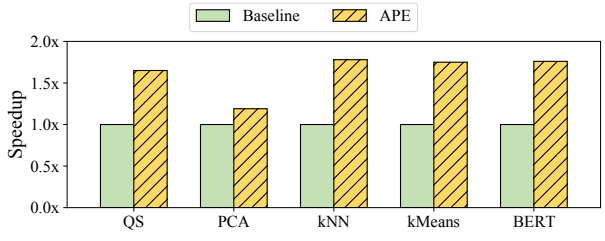

Figure 9: Applications speedup over CUDA Core FP32.

algorithm should be designed to determine the proper data type for each computation.

To lower the latency needed by making the decision online, a light-weight decision algorithm is developed. A performance model is built to predict the running time of GEMM computation using either dedicated FP32-B or block-wise hybrid-type computation method.

The model is based on the data mask. For simplicity, we assume that the scheduling method and load balance during the computation is ideal. For each block $C_{i,j,k}$, the data type used in computation is determined according to $A_{i,k}$ and $B_{k,j}$. When any input block is FP32-B, both computing blocks must use FP32-B. Otherwise, both of the input blocks are FP32-F, and the computing block can use FP32-F. Therefore, the number of blocks corresponding to each type is counted using the following algorithm. First, encode FP32-F and FP32-B in the data mask as 0 and 1, respectively. Then, a small GEMM $mask_C = mask_A \times mask_B$ is performed. Here, each element $mask_C$ represents the number of blocks that can use FP32-F in the output matrix. Finally, all elements in $mask_C$ are summed up to obtain the number of FP32-F computing blocks in the computing process.

By sampling the performance of blocks with each data type, we can calculate the time for a given computation. Therefore, we predict the running time of using dedicated FP32-F, FP32-B, or hybrid-computing and select the best.

6 IMPLEMENTATION

We implement the full-functional Ape on NVIDIA Tensor Core GPUs, including Tesla V100, T4, and A100. Since we do not have open access to program on Huawei Ascend processors, we do not implement the block-wise hybrid computation on Ascend. Instead, we use wrapped kernels like GEMM and Convolution provided by vendors to verify our approach on Ascend. Although existing libraries like cuBLAS [30] provide highly optimized kernels, it is not sufficient enough to support all of Ape's functions. For example, we need to implement kernels for different types with a fine-grained schedule to support our hybrid method. Furthermore, as Ape supports different platforms, we proposed a code generator to generate different kernels with various configurations and choose the best for each platform.

For example, on Tensor Core, the kernel should overlap the computation and memory access to fill the computation throughput. The pipeline should be carefully designed. Moreover, data casting can be put inside or outside a kernel. The better approach depends on the emulated types and the platforms. We implement certain code templates which describe several implementations, including using wrapped kernels or using kernels with different pipeline strategies. The code generator will generate various kernels according to the templates. Ape will choose the most performant kernel for each situation on each platform.

Ape provides users a similar interface to other BLAS [3] libraries and currently supports two linear operations, GEMM and convolution. Users can easily use Ape by replacing the original library calls of GEMM and convolution with Ape's library calls by modifying either the code or the library linking, without modifying the rest of the application. If the users have expert knowledge about the data range and precision, they can also manually choose the kernel of a certain data type for further optimization.

7 EVALUATION

7.1 Experiment Setup

We evaluate Ape on NVIDIA GPUs and Huawei Ascend AI processors, respectively. For NVIDIA GPUs, the evaluation is performed on NVIDIA Tesla V100, T4, and A100. The CUDA version is unified to 11.0.2, and the memory and application clock rates of the GPUs are fixed at maximum to avoid performance variance. For Huawei Ascend processors, the evaluation is on Ascend 910A, which has a peak performance of 256 TFLOPS on FP16 and 512 TOPS on INT8. Unless otherwise stated, all the evaluation includes the overhead of emulation and adaptation.

7.2 End-to-End Performance

We use five applications to evaluate the performance of Ape on NVIDIA GPUs. Baselines of all the applications are those implemented with GEMM in cuBLAS (either initially using cuBLAS or adapted by us) on CUDA Core. As illustrated in Section 6, we only replace GEMM library from cuBLAS to Ape, and the rest of the applications remain unchanged. The results are shown in Figure 9.

Quantum circuit simulation (QS) can be implemented with GEMM and tensor transpose [39]. With the maximum matrix size being 512, the average speedup of Ape is 1.65×.

Principal component analysis (PCA) [19] is used for exploratory data analysis, and the most computationally intensive operations in it are GEMM and EIG (eigenvalues). Our implementation of PCA is based on cuBLAS and cuSolver on GPU. Ape can accelerate PCA

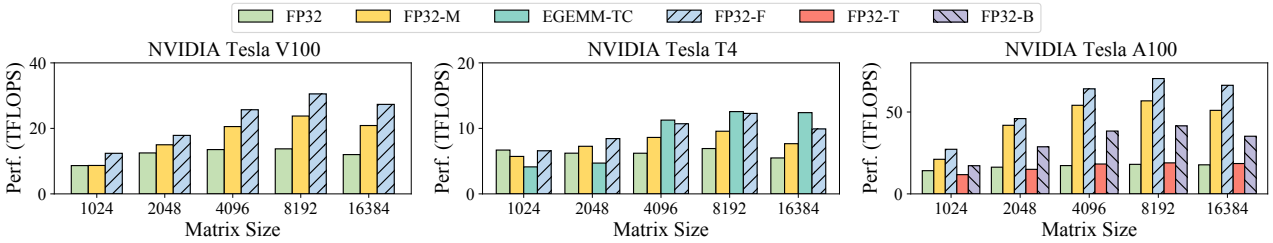

Figure 10: Emulation performance of floating-point data types on NVIDIA GPUs.

by 1.19×, where GEMM takes 39.25% of the total execution time in the original version.

K-Nearest Neighbors (kNN) [2, 13] is a classic algorithm for classification and regression. The inputs are training examples in a feature space. The most computationally intensive operation in kNN is a GEMM. The baseline approach is an open-source kNN implementation on GPU [14]. In the cuBLAS version, GEMM takes 76.9% of the execution time, and Ape can provide a speedup of 1.78× for kNN.

K-means clustering (kMeans) [28] is a method that partition n samples into k clusters, in which each example belongs to the cluster with the nearest mean. We adopt an open-source kMeans implementation on GPU from NVIDIA [31] as the baseline. GEMM accounts for 77.2% of running time in kMeans, and the speedup of Ape is 1.75×.

Bidirectional Encoder Representations from Transformers (BERT) [8] is a Transformer model widely used for natural language processing. GEMM is the most time-consuming computation in BERT. We applied Ape on an open source BERT implementation cuBERT [10]. When evaluating BERT with single precision, FP32-F can achieve up to 1.77× speedup.

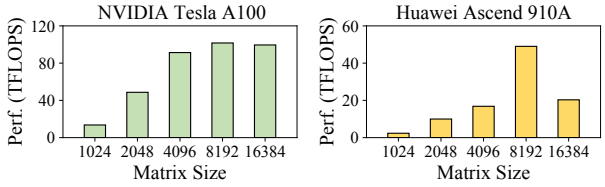

Figure 11: Emulation performance of fixed-point data types on Tesla A100 and Ascend 910A.

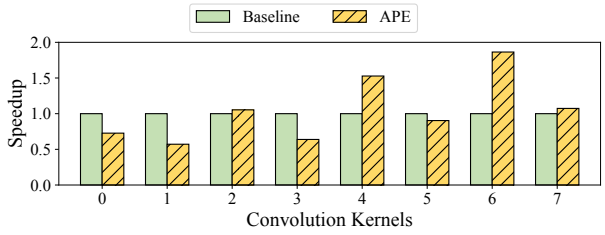

Figure 12: Performance of convolution on Tesla A100.

7.3 GEMM Performance

Floating-Point We evaluate different floating-point emulations on NVIDIA GPUs, including NVIDIA Tesla V100, T4, and A100. The results are shown in Figure 10. The baselines are GEMM on CUDA Core in cuBLAS and previous approaches on Tensor Core, including Markidis et al.'s work [29] denoted as FP32-M, and EGEMM-TC [11]. We reproduce FP32-M following their paper. Since EGEMM-TC is not open-source and uses SASS assembly-style instructions dedicated to Turing architecture, we only compare with the performance metrics reported by EGEMM-TC's paper on Tesla T4. All the evaluation of the performance of Ape on GPU includes the time for pre-processing and post-processing.

On A100, FP32-F and FP32-B have both significantly higher performance than cuBLAS FP32 approach with 3.12× and 1.85× speedups, respectively. Compared to previous approach, FP32-F provides better data ranges, and meanwhile, FP32-F can still outperforms FP32-M by 1.22×.

On V100 and T4, Ape only enables FP32-F since V100 and T4 do not support BF16 or TF32 on Tensor Core. On these platforms, FP32-F can optimize cuBLAS FP32 in most cases by 1.81× and 1.49× respectively. Ape consistently outperforms FP32-M by 1.29× on V100 and 1.23× on T4. Compared to EGEMM-TC on T4, Ape achieves better performance on small matrices but worse on matrices larger than 4096 because T4 provides lower memory bandwidth than V100 and A100, which requires carefully tuning on memory accesses. EGEMM-TC utilizes SASS assembly-style instructions to optimize its performance. However, since SASS is not portable on different microarchitectures, it cannot be easily ported to other GPUs. Interestingly, due to the low memory access bandwidth of T4, the split operations take a large part of the time (e.g., 17.5% when $N = 2048$), so the splitting overhead cannot be ignored and brings challenges to code generation.

Fixed-Point We evaluate fixed-point emulations on different accelerators, including Tensor Core on NVIDIA Tesla A100 GPU and Huawei Ascend 910A processor. We use two INT8 fixed-point numbers to emulate one INT16, which is not supported on either NVIDIA GPU or Ascend processor. The evaluation shows that with Ape's emulation, the accelerators can support computations on non-native data types. The performance of emulated INT16 fixed-point GEMM computation is shown in Figure 11. The running time excludes the data casting time.

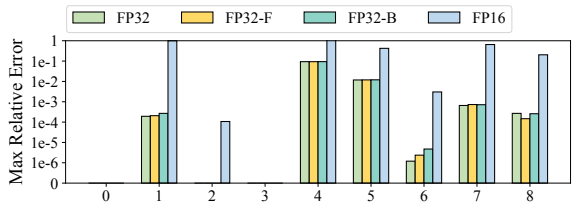

Figure 13: Maximum error of quantum simulation. IDs on x-axis represent different circuits.

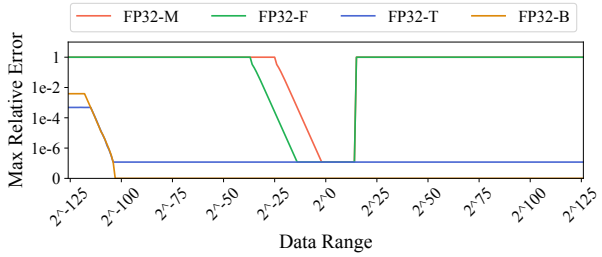

Figure 14: Maximum errors of different emulation types.

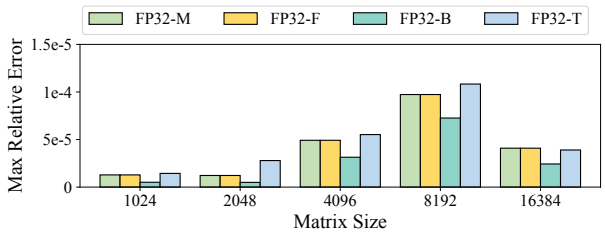

Figure 15: Maximum error of GEMM operator.

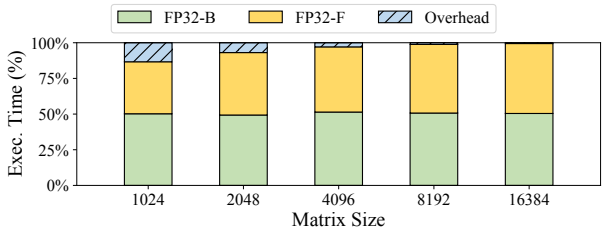

Figure 16: Computation time break-down analysis of the hybrid emulation method.

7.4 Convolution Performance

To verify the correctness of Ape in other linear computations, we evaluate convolution with FP32-F and FP32 on NVIDIA Tesla A100. In this experiment, both approach are based on cuDNN [4]. FP32-F performs with Tensor Core and FP32 performs with CUDA Core. We choose several convolutions in ResNet-18. The result is shown in Figure 12. In some cases, FP32-F can achieve higher performance than FP32, which shows Ape can not only support convolution on Tensor Core but potentially optimize convolution with cuDNN implementation. Although convolution with cuDNN is not a direct hardware implementation, this experiment verifies that Ape's method can be performed on convolution and expanded to convolution DSAs.

7.5 Accuracy

Application Accuracy To evaluate accuracy in real applications when using Ape, we take QS as a case study. QS has a numerical output that can be easily compared with the FP64 result. As a result, QS can intuitively reflect the accuracy loss of Ape under different datatype configurations. As shown in Figure 13, compared with the result of FP64, the errors of FP32-F and FP32-B are equivalent to FP32. However, directly using FP16 will cause obvious errors in most cases. For example, in the circuit of *ISING_25*(#4), the error of FP32-F, FP32-B, and FP32 is all about 9%, while the error of FP16 is 172%. Therefore, the emulation method provided by Ape can meet the requirements of quantum simulation while FP16 cannot.

Emulation Accuracy In order to verify the emulation precision of Ape, we generate multiple sets of data and compare the accuracy loss of emulated types. The result is shown in Figure 14. The point corresponding to the abscissa x represents the error of 16384 numbers generated randomly and uniformly in $(x, 2x)$.

Among them, FP32-F and FP32-M have a stable error in the representation range of about 1.2×10^{-7}, which is equivalent to the previous theoretical analysis. However, compared with FP32-M emulation, FP32-F has a larger data range without much accuracy loss. The accuracy of FP32-M decreases when the data value is less than 0.25, and the accuracy of FP32-F decreases when the data value is below 6.1×10^{-5}.

The FP32-B and FP32-T have a data range close to FP32, while FP32-B has almost no error within its range compared to FP32. FP32-T has a relative error of about 1.2×10^{-7} within its range. This means that FP32-B and FP32-T have similar precision and range compared to FP32.

Computing Accuracy The accuracy of the emulation method in computation is inspected by performing GEMM operator on random matrices of different sizes. The maximum relative error is reported in Figure 15.

An observation is that error increases with matrix size in all emulation methods. This is because the truncation method of Tensor Core is *round-to-zero* rather than *round-to-nearest*. Therefore, truncation errors are accumulated during reductions, leading to that errors would grow as data range increases. However, *round-to-zero* is a hardware design choice of Tensor Core, and it is not brought from emulations.

Another interesting problem is that different kernels have different errors. Ape will choose a proper kernel to achieve better performance on each size. Therefore, the errors caused by the algorithm will vary with the matrix size. The kernel of 16384 is different from the kernel of 8192. So there are obviously different errors between these kernels.

7.6 Breakdown

The performance of the hybrid emulation method is analyzed by breaking down the computation time of performing GEMM on input matrices where 60% of the blocks are emulated by FP32-F and the rest emulated by FP32-B. The result of different input matrix sizes is shown in Figure 16. *FP32-F*, *FP32-B*, and *Overhead* represent their running time. *Overhead* contains the time to inspect the range for each input element and the time to determine the label for each block.

The result shows that making fine-grained decisions between emulation methods for each block costs negligible time compared to the time spent on the actual GEMM computation. Much time is saved by utilizing FP32-F with the decision strategy as we expected. In brief, the adapter introduces little overhead and effectively improves the performance.

8 DISCUSSION

The methodology of Ape can support different linear computation based on Fused Multiplication-Accumu-lation (FMA), such as GEMM and convolution, which play essential roles across different domains, including AI and HPC. In this paper, we mainly evaluate Ape on NVIDIA Tensor Core GPUs and Ascend accelerators. But as mentioned in Section 5, Ape can be implemented on all the hardware that meets certain requirements. Ape is not only limited to the computation and hardware evaluated in this paper, but also can potentially support new hardware designed for other FMA-based linear computations.

Meanwhile, Ape shows that we can use low-bitwidth data types to emulate high-bitwidth data types without losing much accuracy or hurting the correctness, which can shed some light on hardware design. For example, if an FP64 accumulator is provided for an FP16 computing unit, Ape can also be used to emulate FP64 computation. Therefore, we can use the same FP16 computing unit to emulate FP32 and FP64 data types, thereby saving hardware on-chip area, which is a significant bottleneck in chip design.

9 CONCLUSION

In this work, we present Ape, which is the first framework that provides various emulated data types on DSAs and automatically selects proper data types. Ape can accelerate a given computation with fine-granularity to achieve higher performance while maintaining correctness and precision. Ape extends computation scenarios for better utilization of DSAs without any human efforts. We implement Ape on both NVIDIA Tensor Core and Huawei Ascend. The evaluation shows that Ape can boost FP32 General Matrix Multiplication and convolution by up to 3.12× and 1.86× on Tensor Core compared with CUDA Core, and optimize various applications by up to 1.78× (1.61× on average).

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments and suggestions. This work is supported by the National Key R&D Program of China under Grant 2021ZD0110104, the National Natural Science Foundation of China (U20A20226), and the Beijing Natural Science Foundation (4202031). Jidong Zhai is the corresponding author of this paper.

REFERENCES

[1] 2008. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008* (2008), 1–70. https://doi.org/10.1109/IEEESTD.2008.4610935
[2] Naomi S Altman. 1992. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician* 46, 3 (1992), 175–185.
[3] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. 2002. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Software* 28, 2 (2002), 135–151.
[4] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).
[5] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. 2017. Rigorous Floating-Point Mixed-Precision Tuning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 300–315. https://doi.org/10.1145/3009837.3009846
[6] Arnab Das, Ian Briggs, Ganesh Gopalakrishnan, Sriram Krishnamoorthy, and Pavel Panchekha. 2020. Scalable yet Rigorous Floating-Point Error Analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 51, 14 pages.
[7] Theodorus Jozef Dekker. 1971. A floating-point technique for extending the available precision. *Numer. Math.* 18, 3 (1971), 224–242.
[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).
[9] Jack J Dongarra, Hans W Meuer, Erich Strohmaier, et al. 1997. TOP500 supercomputer sites. *Supercomputer* 13 (1997), 89–111.
[10] Liwen Fan, Ruixin Wang, Kuan Fang, and Xian Sun. 2019. cuBERT. https://github.com/zhihu/cuBERT.
[11] Boyuan Feng, Yuke Wang, Guoyang Chen, Weifeng Zhang, Yuan Xie, and Yufei Ding. 2021. EGEMM-TC: accelerating scientific computing on tensor cores with extended precision. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 278–291.
[12] Boyuan Feng, Yuke Wang, Tong Geng, Ang Li, and Yufei Ding. 2021. Apnn-tc: Accelerating arbitrary precision neural networks on ampere gpu tensor cores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.
[13] Evelyn Fix. 1951. *Discriminatory analysis: nonparametric discrimination, consistency properties*. USAF School of Aviation Medicine.
[14] Vincent Garcia, Eric Debreuve, and Michel Barlaud. 2008. Fast k nearest neighbor search using GPU. In *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*. IEEE, 1–6.
[15] Google. 2020. Advanced neural network processing for low-power devices. https://coral.ai/technology
[16] Sridhar Gopinath, Nikhil Ghanathe, Vivek Seshadri, and Rahul Sharma. 2019. Compiling KB-Sized Machine Learning Models to Tiny IoT Devices. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 79–95. https://doi.org/10.1145/3314221.3314597
[17] Huawei. 2022. Ascend to Pervasive Intelligence. https://e.huawei.com/en/products/servers/ascend
[18] Kyuyeon Hwang and Wonyong Sung. 2014. Fixed-point feedforward deep neural network design using weights+ 1, 0, and- 1. In *2014 IEEE Workshop on Signal Processing Systems (SiPS)*. IEEE, 1–6.
[19] J Edward Jackson. 2005. *A user's guide to principal components*. Vol. 587. John Wiley & Sons.
[20] Zhe Jia, Blake Tillman, Marco Maggioni, and Daniele Paolo Scarpazza. 2019. Dissecting the graphcore ipu architecture via microbenchmarking. *arXiv preprint arXiv:1912.03413* (2019).
[21] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 1–12.
[22] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, Raquel Urtasun, and Andreas Moshovos. 2015. Reduced-precision strategies for bounded memory in deep neural nets. *arXiv preprint arXiv:1511.05236* (2015).
[23] Paresh Kharya. 2020. TensorFloat-32 in the A100 GPU Accelerates AI Training, HPC up to 20x. *the NVIDIA Blog* (2020).
[24] Liangzhen Lai, Naveen Suda, and Vikas Chandra. 2017. Deep convolutional neural network inference with floating-point weights and fixed-point activations. *arXiv preprint arXiv:1703.03073* (2017).

[25] Heng Liao, Jiajin Tu, Jing Xia, Hu Liu, Xiping Zhou, Honghui Yuan, and Yuxing Hu. 2021. Ascend: a Scalable and Unified Architecture for Ubiquitous Deep Neural Network Computing: Industry Track Paper. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 789–801.

[26] Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. 2016. Fixed point quantization of deep convolutional networks. In *International conference on machine learning*. PMLR, 2849–2858.

[27] Seppo Linnainmaa. 1981. Software for doubled-precision floating-point computations. *ACM Transactions on Mathematical Software (TOMS)* 7, 3 (1981), 272–283.

[28] James MacQueen et al. 1967. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, Vol. 1. Oakland, CA, USA, 281–297.

[29] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. 2018. Nvidia tensor core programmability, performance & precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 522–531.

[30] NVIDIA. [n.d.]. cuBLAS. https://developer.nvidia.com/cublas.

[31] NVIDIA. 2013. NVIDIA/kmeans. https://github.com/NVIDIA/kmeans.

[32] NVIDIA. 2020. NVIDIA A100 Tensor Core GPU Architecture. UNPRECEDENTED ACCELERATION AT EVERY SCALE. Version v1.0. *NVIDIA* (2020).

[33] Tesla NVIDIA. 2017. V100 GPU architecture. the world's most advanced data center GPU. Version WP-08608-001_v1.1. *NVIDIA. Aug* (2017).

[34] Zhixiang Ren, Yongheng Liu, Tianhui Shi, Lei Xie, Yue Zhou, Jidong Zhai, Youhui Zhang, Yunquan Zhang, and Wenguang Chen. 2021. AIPerf: Automated machine learning as an AI-HPC benchmark. *Big Data Mining and Analytics* 4, 3 (2021), 208–220.

[35] Alexey Solovyev, Marek S Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. 2018. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 41, 1 (2018), 1–39.

[36] Andrew Thall. 2006. Extended-precision floating-point numbers for GPU computation. In *ACM SIGGRAPH 2006 research posters*. 52–es.

[37] Sudharshan S Vazhkudai, Bronis R De Supinski, Arthur S Bland, Al Geist, James Sexton, Jim Kahle, Christopher J Zimmer, Scott Atchley, Sarp Oral, Don E Maxwell, et al. 2018. The design, deployment, and evaluation of the CORAL pre-exascale systems. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 661–672.

[38] Shibo Wang and Pankaj Kanwar. 2019. BFloat16: the secret to high performance on cloud TPUs. *Google Cloud Blog* (2019).

[39] Chen Zhang, Zeyu Song, Haojie Wang, Kaiyuan Rong, and Jidong Zhai. 2021. HyQuas: hybrid partitioner based quantum circuit simulation system on GPU. In *Proceedings of the ACM International Conference on Supercomputing*. 443–454.