VitBit: Enhancing Embedded GPU Performance for AI Workloads through Register Operand Packing

Anonymous Author(s)

ABSTRACT

The rapid advancement of Artificial Intelligence (AI) necessitates significant enhancements in the energy efficiency of Graphics Processing Units (GPUs) for Deep Neural Network (DNN) workloads. Such a challenge is particularly critical for embedded GPUs, which operate within stringent power constraints. Traditional GPU architectures, designed to support a limited set of numeric formats, face challenges in meeting the diverse requirements of modern AI applications. These applications demand support for various numeric formats to optimize computational speed and efficiency. This paper proposes VitBit, a novel software technique designed to overcome these limitations by enabling efficient processing of arbitrary integer format values, especially those 8 bits or fewer, which are increasingly prevalent in AI workloads. VitBit introduces two key innovations: the packing of arbitrary integer formats for parallel computation and the simultaneous execution of Tensor cores, INT and FP (Integer and Floating-Point) CUDA cores. This approach leverages the architectural features of modern GPUs, such as those based on NVIDIA Ampere architecture, which allows concurrent operation of FP32 and INT32 cores at full throughput. Our evaluation of VitBit on NVIDIA Jetson AGX Orin demonstrates substantial improvements in arithmetic density and peak throughput, achieving up to a 22% reduction in execution time for benchmark AI workloads without compromising computational accuracy. VitBit effectively bridges the gap between current hardware capabilities and the computational demands of AI, offering a scalable and cost-effective method for enhancing GPU performance in AI applications.

KEYWORDS

GPU, DNN

ACM Reference Format:

Anonymous Author(s). 2024. VitBit: Enhancing Embedded GPU Performance for AI Workloads through Register Operand Packing. In *ICPP '24: International Conference on Parallel Processing, August 12–14, 2024, Gotland, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Artificial Intelligence (AI) landscape has seen the emergence of various numeric formats designed to boost computational and memory efficiency [1–12]. Low-bitwidth floating point quantization such as FP4 and FP6 efficiently reduces the size of Large Language Models (LLMs) over the traditional floating point formats (e.g., FP32) while maintaining consistent model quality across diverse applications [13, 14]. Diverse integer formats are also employed in emerging AI algorithms to enhance inference speed [6–10].

Improving the energy efficiency of Graphics Processing Units (GPUs) in Deep Neural Network (DNN) workloads becomes a paramount consideration [1–10, 15–21]. It is even more critical for embedded GPUs, which have to perform well under strict power limits. A prominent strategy to enhance GPU energy efficiency involves increasing the arithmetic density (operations per second per mm^2). Traditionally, GPUs were built to handle a limited set of data formats, which was enough in the past [22, 23]. AI applications demand a diverse array of numeric formats to enhance computation speed and efficiency, a requirement that traditional GPUs, with their fixed design, fail to meet. While producing new GPUs capable of handling the latest formats could mitigate this issue, it would entail substantial costs and environmental impacts associated with manufacturing.

Building upon prior work that has leveraged register packing [24] and register coalescing [25] for utilizing low-bitwidth values on GPUs with limited numeric format support, our investigation reveals insights into optimizing GPU performance for deep learning applications. While executing DNN workloads on GPUs, utilizing Tensor cores for the primary computations, such as General Matrix-Matrix Multiplication (GEMM) has been a focal strategy. Although CUDA cores offer comparatively lower performance, harnessing register packing can significantly enhance the actual throughput of CUDA cores. By employing both Tensor and CUDA cores concurrently in the primary operations, performance improvement can be achieved [26, 27]. The register packing technique has shown promise in enhancing register file utilization by condensing low-bitwidth operations. Facilitating the accommodation of more operands within a Streaming Multiprocessor (SM) does not alter register operands. This limitation implies that, despite the efficient use of register files, the peak throughput of the GPU remains unaffected, as the arithmetic operations executed by the arithmetic logic units (ALUs) in GPUs continue to rely on unchanged operands. If GPUs leverage register packing and coalescing, they can compute arbitrary integer format values of 8 bits or fewer while achieving a higher available throughput than a predetermined peak throughput. We ensure that these computations can occur independently by employing a strategy of packing multiple values, adjusted according to their bitwidth and separated by zero-padding.

In this paper, we propose VitBit[1], a software technique that enhances the performance of conventional GPUs. We design VitBit to circumvent these limitations by enabling GPUs to efficiently

ICPP '24, August 12–15, 2024, Gotland, Sweden

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

[1]We call our new method "VitBit," a name that blends "vitesse," the French word for "speed," with "bit."

process arbitrary integer format values, particularly those used in inference, which is increasingly prevalent in AI applications. VitBit consists of dual strategies: packing arbitrary integer formats and facilitating the simultaneous execution of Tensor cores, INT, and FP CUDA cores. By packing two or more integer values into a single register for parallel computation, VitBit effectively utilizes INT CUDA cores, thus reducing GPU execution time. Also, VitBit converts numeric data formats and reconstructs kernels to execute Tensor, INT, and FP CUDA cores simultaneously by fusing GPU kernels, operating each core part at warp granularity. This method not only optimizes the use of available hardware but also aligns with the capabilities of modern GPU architectures, such as those based on the Ampere architecture. Our experiments, conducted on NVIDIA Jetson AGX Orin, reveal that this approach can lead to an improvement in peak throughput and hardware arithmetic density.

In our evaluation, VitBit substantiates its theoretical advantages, showcasing an improvement in GPU performance. We observed an enhancement in arithmetic density, conducted on the NVIDIA Jetson AGX Orin platform, using the VitBit technique, which includes packing integer formats and the simultaneous activation of INT and FP CUDA cores. VitBit reduces the execution time for benchmark AI workloads by 22% while maintaining computational accuracy. Furthermore, the utilization rate of both INT and FP cores increased dramatically, evidencing a more efficient use of the GPU computation resources. These results not only validate the effectiveness of VitBit in bridging the gap between existing hardware limitations and the demands of modern AI applications but also highlight its potential to serve as a scalable solution for enhancing the computational efficiency of GPUs.

In this paper, we make the following contributions.

- We present a comprehensive analysis of the challenges posed by the fixed architecture of conventional GPUs for processing a variety of numeric formats.
- We introduce a software-based solution, VitBit, that utilizes register packing and harnesses Tensor, INT, and FP CUDA cores simultaneously to enable the efficient use of arbitrary numeric formats on existing GPUs.
- Through extensive experimentation on embedded GPUs, we demonstrate VitBit outperforms the baseline and prior work by 22% and 15% in inference time, respectively.

The rest of this paper consists of the following sections. Section 2 explains the key challenge of arbitrary numeric formats and their support in GPUs. Section 3 explains the details of VitBit. Section 4 shows the experimental results. Section 5 explains related work. Section 6 concludes this paper.

2 WHY VITBIT?

In this section, we first explain the challenges of emerging numeric formats and GPUs and then explain our insights to address them.

2.1 Challenges of Emerging Numeric Formats

As the AI industry advances, improving the energy efficiency of Graphics Processing Unit (GPU) in Deep Neural Network (DNN) workloads becomes one of the most important considerations for researchers [28–31]. In particular, unlike discrete GPUs, embedded GPUs that operate within constrained energy resources prioritize energy efficiency as a critical factor. One solution to improve GPU energy efficiency is to increase GPU hardware arithmetic density.

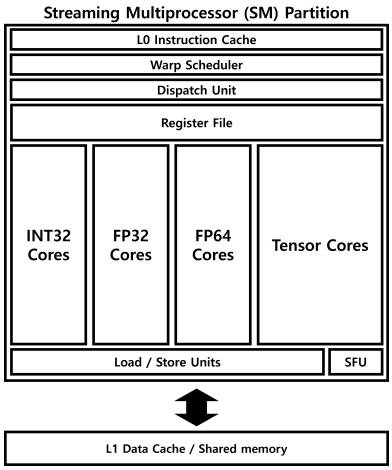


Figure 1: Streaming Multiprocessor (SM) architecture of Jetson Orin GPU

In the recent AI industry, various customized numeric formats have emerged to improve computational and memory efficiency. Six-bit quantization (FP6) reduces the size of Large Language Models (LLMs) effectively while consistently preserving model quality across varied applications [13]. Prior work has introduced 6-bit and 2-bit integer formats for inference as well [6, 8–10]. Also, microscaling data formats (MX) emerge to balance hardware efficiency, model accuracy, and user friction [11]. Hybrid Block Floating Point (HBFP) combines block floating point with fixed point, achieving higher throughput with high floating point accuracy at the superior hardware density of fixed point [12].

Conventional GPUs are limited in their ability to fully support these emerging numeric formats. Figure 1 depicts the structure of a Streaming Multiprocessor (SM) of an embedded GPU. We refer to the specification of Jetson Orin AGX [32]. Conventional GPUs only support limited numeric formats, and utilizing these fixed formats for computations results in significantly low arithmetic density. Leveraging arbitrary numeric formats on existing GPUs without any preprocessing causes two major problems. The first problem is the mismatch between the size of registers and the arbitrary formats. The size of registers currently used within the GPU is fixed at 32 bits, making it impossible to store data of unsupported numeric formats within the GPU. The second problem is that the Arithmetic Logic Units (ALUs) within the GPU do not support arbitrary numeric formats.

Creating a novel GPU hardware architecture suitable for emerging numeric formats not only consumes considerable time but also incurs significant chip manufacturing and environmental effects. As the ALUs within the GPU Streaming Multiprocessor (SM) cores can only perform operations for fixed numeric formats, supporting arbitrary numeric formats on the GPU necessitates a software-based approach. One software method for utilizing arbitrary numeric formats within the registers and ALUs, which only support limited numeric formats, is the zero-masking technique. Zero-masking

Table 1: Peak throughput of NVIDIA Jetson Orin AGX. Performing INT8 or INT4 within CUDA cores is possible with a software technique such as zero-masking. In this case, the peak throughput is equivalent to that of INT32.

Numeric Format	Peak Throughput
FP32 (CUDA Core)	4 TFLOPS
FP16 (CUDA Core)	8 TFLOPS
TF32 (Tensor Core)	32 TFLOPS
FP16 (Tensor Core)	65 TFLOPS
BFloat16 (Tensor Core)	65 TFLOPS
INT32 (CUDA Core)	4 TOPS
INT8 (Tensor Core)	131 TOPS
INT4 (Tensor Core)	262 TOPS

is a simple software technique that enables the use of arbitrary numeric formats on existing GPUs [33–35]. Zero-masking incorporates arbitrary numeric format values into fixed precision by masking all remaining bits with zeros. To evaluate the viability of the zero-masking technique on conventional embedded GPUs, we conduct a simple experiment.

We analyze the arithmetic density of a GPU by varying integer bitwidths. As a GPU is built within a fixed size, we compare the peak throughput of various numeric formats. Table 1 depicts the analysis results. Tensor cores exhibit an increase in peak throughput as a shorter numeric format is employed. Relatively, CUDA cores show a lower throughput than Tensor cores. If CUDA cores support narrow bitwidth numeric formats, they would enhance their peak throughput. For example, CUDA cores do not support INT8 and INT4, so they suffer from throughput saturation at the INT32 throughput. If a GPU in Jetson ideally supports INT8 in CUDA cores and enhances their throughput proportionally, up to 32 TOPS might be achieved. Such a throughput corresponds to 25% of the peak throughput of Tensor cores, which is not negligible. Also, if software could exploit the CUDA cores for floating-point formats as well, available throughput could be improved further. As a result, a novel software-based solution that utilizes CUDA cores to improve their throughput is strongly required to improve the arithmetic density of embedded GPUs.

2.2 Packing Register Operands for Enhanced CUDA Core Utilization

CUDA cores occupy a significant portion of the GPU architecture, highlighting a need to harness their potential more effectively to boost DNN workload performance. The extensive real estate that CUDA cores command on the GPU die presents a unique opportunity for optimization. Moreover, as CUDA cores are not used for performing key computations of DNN tasks, they often become idle. Leveraging this vast array of CUDA cores efficiently can lead to substantial improvements in the processing of DNN tasks. The challenge lies in overcoming the inherent speed limitations of CUDA cores through innovative software and architectural strategies, enabling them to contribute more significantly to the overall computational throughput. By devising methods to enhance the utility of CUDA cores, we can achieve higher level of performance without underutilizing the huge computation units. This approach not only optimizes the existing hardware but also sets a foundation for future GPU designs to achieve a balanced distribution of computational power, ensuring that every component, regardless of its individual speed, contributes effectively to the AI processing capabilities of GPUs.

Prior work has proposed register packing [24] and register coalescing [25] to enable the use of low bitwidth values on hardware supporting limited numeric formats. Register packing involves detecting low bitwidth operations at the write-back stage, then packing them to store in the register file, thereby providing extra space within register file. The additional register file space within the SM enables hosting more thread blocks. While register packing is able to reduce the effective size of the register file, each register read still requires a separate physical register read. Register coalescing aims to read multiple related registers used by the same instruction through a single register read operation in order to utilize the register file bandwidth more efficiently.

Both register packing and register coalescing use the packed values to enable the use of low bitwidth values on hardware supporting limited numeric formats. However, packed values that register packing and register coalescing used only exist on the register file and operand collection pipeline stage. The operand used in the execution stage does not change. Therefore, the GPU peak throughput remains the same because the operand used for the arithmetic unit within the GPU execution pipeline stage is unchanged.

Unlike the register packing and register coalescing techniques, GPUs can compute arbitrary integer format values of 8 bits or fewer in INT cores by packing two or more register operands. Packing register operands by spacing the values according to the bitwidth of the arbitrary numeric format and padding the remaining bits with zeros require a new software implementation for the computations that ensures the accurate results.

2.3 Simultaneous Execution of Tensor Cores and CUDA Cores

In response to the demand for high throughput computations for DNN workloads, the industry releases Tensor cores [36]. Tensor cores are designed specifically for deep learning that accelerates matrix multiplication, which is the core operation in DNNs. Tensor cores exhibit a higher throughput for GEMM than CUDA cores. For this reason, numerous AI frameworks employ Tensor cores for processing matrix multiplication, such as General Matrix Matrix Multiplication (GEMM), and other operations for CUDA cores [37].

However, this exclusiveness of kernel execution on heterogeneous cores results in low arithmetic density as CUDA cores and Tensor cores occupy a vast area of the limited GPU hardware. To improve arithmetic density and accelerate DNN workloads, the simultaneous execution of both Tensor and CUDA cores during kernel execution shows promise [26, 27]. Tacker fuses two different kernels to enable concurrent execution of Tensor cores and CUDA cores [26]. Based on the fact that multiple warps of a single thread block are active at the same time, Tacker enables Tensor cores and CUDA cores concurrently by assigning different warps to utilize the two cores within the same thread block. Also, prior work has proposed a technique that offloads a GEMM operation to Tensor cores and CUDA cores simultaneously [27].

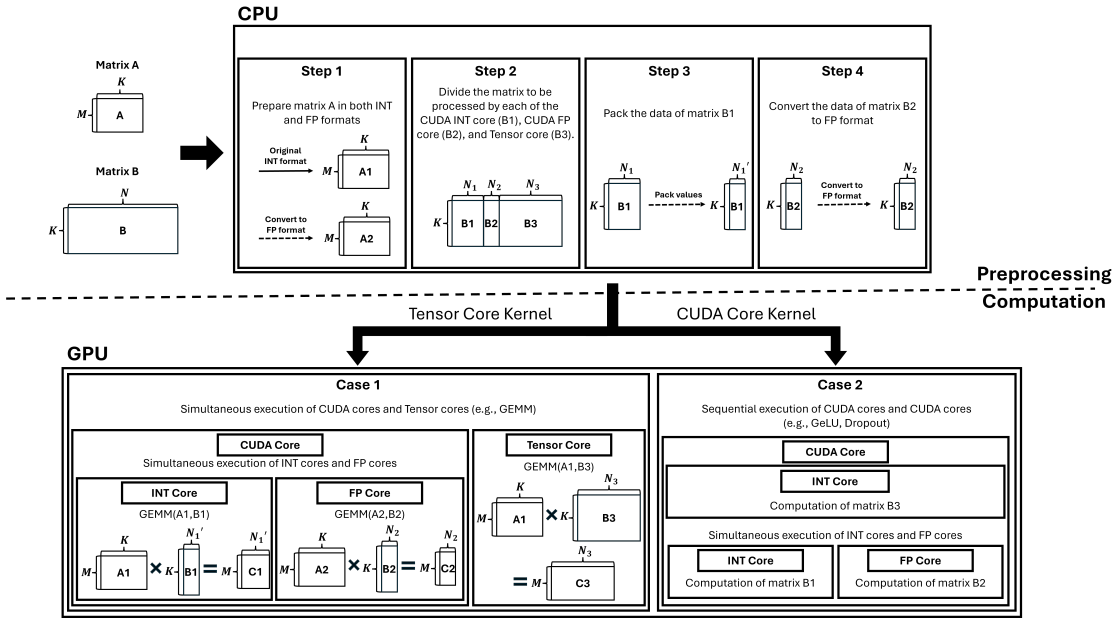


Figure 2: Conceptual overview of VitBit. The upper portion represents the preprocessing stage, which is executed on the CPU. The lower portion involves computing on the GPU using the data obtained from the preprocessing stage.

Unfortunately, the aforementioned approaches are limited to achieving the maximum arithmetic density. As shown in Figure 1, each Streaming Multiprocessor (SM) consists of heterogeneous arithmetic units that process predefined numeric formats. Considering most of the DNN applications utilize fixed numeric formats (e.g., INT32, FP32, etc.), the prior work exploits either INT cores or FP cores while concurrently executing Tensor cores.

Leveraging both CUDA and Tensor cores simultaneously in DNN workloads holds the key to unlocking significant performance gains. To fully harness the potential of simultaneous execution of CUDA and Tensor cores, novel algorithms and programming models are required to orchestrate computational tasks across these heterogeneous cores.

3 VITBIT

To address the challenges outlined in the previous section, we propose VitBit, a software technique designed to enhance peak throughput and arithmetic density on GPU hardware while utilizing low-bitwidth arbitrary integer format values on conventional GPUs. VitBit introduces two key innovations: the packing of arbitrary integer formats and the simultaneous execution of Tensor Cores, INT, and FP CUDA cores. By packing two or more integer values into a single register and performing parallel computations, VitBit enables the simultaneous operation of packed values within a single instruction, significantly reducing GPU execution time. Furthermore, VitBit executes INT CUDA cores, FP CUDA cores, and Tensor cores simultaneously via kernel fusion.

3.1 VitBit Overview

Figure 2 illustrates a conceptual overview of VitBit. We focus on the fact that typical DNN inference utilizes fixed-point formats. Accordingly, we design VitBit to pack multiple integer operands into single registers. VitBit consists of two primary components: data preprocessing and kernel reconstruction. Data preprocessing involves data type conversion and packing to enable simultaneous execution occurring on the CPU. Kernel reconstruction modifies DNN kernels to allow simultaneous execution with preprocessed data on the GPU. The preprocessing phase comprises four steps as shown in the top of Figure 2.

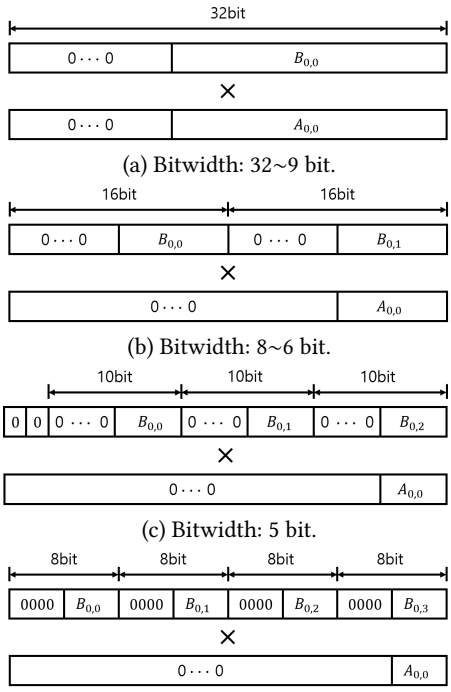
Step 1: VitBit converts the INT filter matrix A to a matrix that can be computed by FP CUDA cores but still contains the parameters with the fixed-point format. This type conversion is only required once during the initial setup for the device.

Step 2: VitBit divides an input matrix (B in Figure 2) for simultaneous execution: matrix B1 for INT CUDA cores, matrix B2 for FP CUDA cores, matrix B3 for Tensor cores. VitBit determines the ratio for the division according to the ratio of execution time for GEMM operations on each core.

Step 3: VitBit packs the parameters in matrix B1 following the packing policy predetermined based on the size of an actual operand (more details in Section 3.2).

Step 4: VitBit converts the data from matrix B2 to be compatible with the floating point format of matrix A.

After preprocessing, VitBit reconstructs new GPU kernels considering two cases. The first case is executing kernels utilizing Tensor cores on the GPU (eg., GEMM, etc.). For the first case, VitBit reconstructs the kernel to execute INT, FP CUDA cores and Tensor cores simultaneously. VitBit divides threads that use INT, FP CUDA cores and Tensor cores in a unit of warp size (i.e., 32), thus



(a) Bitwidth: 32~9 bit.

(b) Bitwidth: 8~6 bit.

(c) Bitwidth: 5 bit.

(d) Bit width: 4 bit.

Figure 3: VitBit packing policy. VitBit applies packing policy to integer data executed on the INT cores. Integer values with a bitwidth of 9 or more (a) utilize simple zero-masking. Integer values with a bitwidth between 6 and 8 (b), those with a bitwidth of 5 (c), and those with a bitwidth of 4 (d) can be packed up to 2, 3, and 4 values, respectively.

automatically synchronizing the process of all active threads via hardware-level warp scheduling. INT, FP, and Tensor core operations within a single SM, enabling them to run in parallel. For the second case, which executes CUDA cores only (e.g., GeLU, Dropout, etc.), VitBit reconstructs the kernels to process the matrix B3 on INT CUDA cores. VitBit applies the same policy for the B1 and B2 used in the first case for computation.

3.2 VitBit Data Preprocessing

VitBit duplicates matrix A in both INT and FP formats. Then, VitBit divides the data corresponding to Matrix B into portions for processing by CUDA cores and portions for processing by Tensor cores. To determine the processing ratio, we measure the execution time of GEMM operation for the following five cases: only using Tensor cores (*TC*), only using INT cores (*IC*) or FP cores within the CUDA cores (*FC*), using both INT and FP cores concurrently within the CUDA cores (*IC+FC*), using both INT and FP cores concurrently within the CUDA cores with packing (*IC+FC+P*). In our initial study, we observe that the execution time for *IC* of *FC* increased by approximately 7.5 times compared to *TC*. Also, the increase is 6.5 times for *IC+FC* and 4 times for *IC+FC+P*. Based on this analysis, VitBit determines the assignment ratio of matrices for Tensor cores as 4 and CUDA cores as 1. The balanced assignment prevents either Tensor cores or CUDA cores from becoming bottlenecks in kernel execution and achieves high throughput.

After determining the ratio of matrices to be processed by CUDA cores and Tensor cores, VitBit determines the number of operands to pack into a single register. To establish the packing ratio, VitBit divides the data corresponding to the portion of Matrix B processed by CUDA cores into 'packing' and 'converting' categories based on the specified ratio.

$$Number_{\text{data for packing}} : Number_{\text{data for converting}} = n : 1 \quad (1)$$

Since all packed integer values allow parallel computation during a single operation, if VitBit packs n integer values, the number of integer operations is reduced by a factor of n. Thus, if VitBit divides data according to the ratio in Equation 1, the number of INT operations and FP operations becomes equal. The ratio in Equation 1 reflects the characteristic that the number of available INT cores and FP cores per SM is the same during simultaneous execution.

Figure 3 illustrates the VitBit packing policy. Packing enables integer values to fit into a single register and ensures simultaneous computation for multiple integer values while guaranteeing accurate outputs.

Integer values with a bitwidth of 9 or more utilize simple zero-masking for computation as shown in 3(a). If the bitwidth is between 6 and 8, the output varies between 12 bits and 16 bits as described in 3(b). In this case, a 32-bit register can accommodate up to 2 integer values for computation. As demonstrated in Figure 3(c), packing 3 integers is feasible for integer values with a bitwidth of 5 as the computation output occupies up to 10 bits. Figure 3(d) depicts that it is possible to pack up to 4 integer values with a bitwidth of lower than 4. The packing technique improves the bit-level utilization of registers with low bitwidth integer values, thereby enhancing arithmetic density.

The proposed packing technique allows CUDA cores to perform GEMM operations with packed values without requiring additional arithmetic operations. A single multiplication automatically completes the multiplications with packed values. There is no overhead for handling overflows, as the proposed technique already reserves space for resulting values that exceed the number of digits in the input operands. Also, VitBit does not require the restoration of packed values during the inference process, as intermediate results from one layer are directly used as packed inputs for the next layer.

Algorithm 1 describes the pseudocode of VitBit input preprocessing. The input preprocessing function receives the input matrix B, which is $N \times K$. N demonstrates the width and K means the height of the matrix B. Also, the function requires two ratios for dividing the input matrix for Tensor and CUDA cores (m), INT and FP CUDA cores (n). The preprocessing needs the bitwidth of the arbitrary integer values. Our initial study determines the ratio m, described at the beginning of Section 3.3, and the equation 1 defines the ratio n. The function returns three output matrices: matrix $B1$ with packed integers, matrix $B2$ with floating points data, and matrix $B3$ with masked integers for Tensor cores. After dividing a matrix for each core, VitBit packs the integer values of the matrix $B1_{before\_packing}$ using bit shifting. VitBit utilizes the bitset library to store multiple integer values in a single register through bit shifting. By using the bitset library, VitBit manipulates the values bitwise. VitBit packs

Algorithm 1 Implementation of input preprocessing

Input: Matrix B, which is $N \times K$,
Tensor/CUDA core ratio m, INT/FP core ratio n, $bit\,width$
Output: Matrix with packed integers $B1$, Matrix with floating points $B2$
Matrix with masked-integers for Tensor cores, $B3$

function INPUT_PREPROCESSING(B, N, K, m, n, $bit\,width$)
  /* Set the width of each matrix. */
  $N_3 = N \times m/(1+m)$
  $N_1 = (N - N_3) \times n/(1+n)$
  $N_1{}' = N_1/n$
  $N_2 = (N - N_3) - N_1$

  /* Divide the matrix to be processed by each cores. */
  for $j \leftarrow 0$ to $(K-1)$ **do**
    for $i \leftarrow 0$ to $(N_1 - 1)$ **do**
      $B1_{before\_packing}[i][j] = B[i][j]$
    end for
    for $i \leftarrow N_1$ to $(N_1 + N_2 - 1)$ **do**
      $B2_{before\_converting}[i - N_1][j] = B[i][j]$
    end for
    for $i \leftarrow (N_1 + N_2)$ to $(N-1)$ **do**
      $B3[i - (N_1 + N_2)][j] = B[i][j]$
    end for
  end for

  for $j \leftarrow 0$ to $(K-1)$ **do**
    /* Pack integer values using bit shifting */
    for $i \leftarrow 0$ to $N_1{}'$ **do**
      for $p \leftarrow 0$ to $n-1$ **do**
        /* To store multiple integer values in a single register through
        bit shifting, VitBit utilizes $bitset$ to manipulate values bitwidth. */
        $bitset < bit\,width > element(B1_{before\_packing}[i*n+p])$
        for $l \leftarrow 0$ to $bit\,width$ **do**
          $B1[i][j] \mid= (element[l] << (l + bit\,width * (n - (p+1))))$
        end for
      end for
    end for
    /* Convert integer values to floating point values */
    for $i \leftarrow 0$ to N_2 **do**
      $B2[i][j] = static\_cast < float > (B2_{before\_converting}[i][j])$
    end for
  end for
  return $B1$, $B2$, $B3$
end function

the integer values stored in matrix $B1_{before\_packing}$ into groups of n elements per register to obtain matrix $B3$. Also, VitBit converts all values of matrix $B2_{before\_converting}$ to floating point format through a simple type conversion using static_cast(). Unlike the input matrix B, the preprocessing of the weight matrix A is simple. VitBit creates matrix $A2$ by duplicating the entire matrix A and converting it to a floating point format.

Overhead analysis VitBit preprocesses weights and inputs to avoid additional data type conversion during kernel execution by retaining the same output data format with the preprocessed inputs. VitBit also avoids significant kernel modification by using the converted inputs. We analyze the overhead from employing VitBit.

First, VitBit converts INT weights to FP format to execute INT and FP cores simultaneously. However, the conversion is only required at the initial setup for the embedded GPUs for DNN inference. Thus, the conversion overhead is negligible.

Second, VitBit converts INT inputs to packed INT and FP at the beginning of the inference and turns the data back to INT at the end of the inference. Nevertheless, the size of inputs is less than 1% of the weights in the case of our experiments. Also, we observe that the conversion time is less than 1% of the inference time. Thus, VitBit input conversion overhead is also minimal.

Lastly, VitBit reconstructs DNN kernels into VitBit kernels, enabling simultaneous execution of inference and preprocessing. This kernel alternation occurs only once at the beginning of the inference. Furthermore, since VitBit kernels return the same output format as the preprocessed input format, the sequential execution of DNN inference kernels incurs no data type conversion during inference. Therefore, VitBit introduces trivial software overhead for simultaneous execution, indicating that VitBit is a simple yet effective method for accelerating DNN kernels.

Algorithm 2 VitBit GEMM kernel example.

```
1: function VitBit_GEMM(int *A1, float *A2, int *B1, float *B2, int *B3, ...)
2:     // tid ← Current thread index
3:     if (tid < TC_thread_num) then
4:         // Using Tensor core Core
5:         // GEMM operation of A1 and B3
6:         TC_GEMM(A1, B3)
7:     else
8:         if ((tid/warpSize)%2 == 0) then
9:             // Using INT Core
10:            // GEMM operation of A1 and B1
11:            INT_GEMM(A1, B1)
12:        else
13:            // Using FP Core
14:            // GEMM operation of A2 and B2
15:            FP_GEMM(A2, B2)
16:        end if
17:    end if
18: end function
```

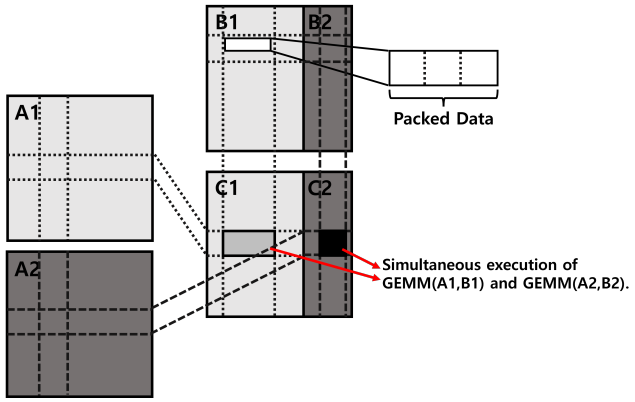


Figure 4: GEMM operation with simultaneous execution of FP cores and INT cores. Matrix C1 is the result of the GEMM operation using matrices A1 and B1. Matrix C2 is the result of the GEMM operation using matrices A2 and B2.

3.3 VitBit Computation

Fusing Tensor Core Kernel: VitBit allows the simultaneous execution of Tensor cores and CUDA cores by fusing kernels. In particular, VitBit targets GEMM operation for simultaneous execution as Tensor cores are optimized for GEMM operation.

Algorithm 2 illustrates the way that VitBit constructs the GEMM kernel. VitBit_GEMM function receives preprocessed data as inputs; filter matrix in INT format (A1), filter matrix in FP format (A2), packed part of input matrix in INT format (B1), converted part of input matrix in FP format (B2), and the part to be processed by the Tensor core of input matrix in INT format (B3). We design the simultaneous execution of Tensor cores and CUDA cores by referring to the kernel fusion technique proposed by prior work [38]. We implement the VitBit kernels by aggregating the threads for Tensor cores and CUDA cores into a single thread block. This strategy relies on the fact that Tensor cores and CUDA cores run in

Table 2: Evaluation configurations and the target DNN model

Platform	NVIDIA Jetson Orin AGX
GPU architecture	Ampere architecture
GPU cores	1792 CUDA cores, 56 Tensor cores
CPU	8-core Arm Cortex-A78AE v8.2
Memory	32GB LPDDR5, Bandwidth: 204.8GB/s
Storage	64GB eMMC 5.1
DNN model	Vision Transformer Base (ViT-Base)

parallel if different warps in a single thread block of a kernel utilize the two units simultaneously [26]. As described in Algorithm 2, VitBit fuses the GEMM operation kernels for B3 and A1, processed on tensor cores, with the GEMM operation kernels for B1 and A1, executed on INT CUDA cores, while GEMM operation kernels for B2 and A2 process FP CUDA cores.

To implement the algorithm 2, we profile the cublas library used in real DNN models to investigate the thread block sizes utilized by Tensor cores [39]. We find that Tensor cores utilize a smaller portion of thread block sizes compared to the total available thread block size. Based on this finding, we divide the entire thread block into two portions; one for the warps to be executed on Tensor cores and the other for the warps to be executed on CUDA cores. Due to the significant proportion of warps occupied by INT cores and FP cores, we schedule them alternately at the warp level to prevent task concentration on one core during warp scheduling.

CUDA Core Kernel: VitBit reconstructs kernels to process matrix B3 using the INT cores within CUDA cores for CUDA core kernels. Then, VitBit uniformly applies packing to matrix B1 for both types of kernels, enabling simultaneous execution of operations on matrix B1 and matrix B2. Unlike in Tensor core kernels, VitBit enables processing matrix B3 differently, in CUDA core kernels, VitBit enables executing the operations for matrix B3 and simultaneous operations for matrix B1 and matrix B2 sequentially. Employing both INT and FP CUDA cores for processing provides a certain degree of performance improvement compared to simply using INT cores for all matrices.

Figure 4 illustrates GEMM operation performed by VitBit in CUDA cores. The figure demonstrates that the use of packed data allows for parallel computation. INT core processes Matrix A1, in integer format, and the packed data matrix B1, while FP core processes matrix A2, in floating point format, and the converted data matrix B2. VitBit divides the width of the packed portion and the converted portion of Matrix B according to the ratio defined in Equation 1. This computation takes advantage of the parallelism afforded by packing, thereby balancing the computational load with that of the converted data. Consequently, VitBit maintains an equitable workload distribution across both the INT and FP cores during operations.

4 EVALUATION

4.1 Methodology

Table 2 describes the system specification of the NVIDIA Jetson Orin platform for evaluation and the target DNN workload, Vision Transformer Base model (i.e., VIT-Base). We obtain the ViT-Base model, pretrained with ImageNet, from Hugging Face [40, 41]. As described in Section 2, we focus on integer-only quantized models, which are composed of integer weights and receive integer inputs for inference. We implement the Vision Transformer Base model (ViT-Base) and customize it to measure the performance improvement of VitBit [42]. We reference the open-source code for implementation of kernels similar to those used in the actual ViT-Base model [43]. Additionally, to ensure accuracy while using the integer-based computations, we apply the computational process provided in the I-ViT paper [44].

Table 3: Comparison group for evaluation. Methodologies labeled with "T" are evaluated for Tensor core kernels (e.g., GEMM) while those labeled with "C" are evaluated for CUDA core kernels (e.g., GeLU, Dropout, etc.). A method with "T,C" is evaluated for both Tensor core kernels and CUDA core kernels, indicating that VitBit serves as a universal solution for both types of kernels.

Methods		Description
TC (baseline)	T	Execution of Tensor cores only (baseline for Tensor core kernels)
IC (baseline)	C	Execution of INT cores only (baseline for CUDA core kernels)
FC	C	Execution of FP cores only by converting INT inputs to FP using type casting
IC+FC	C	Simultaneous execution of INT and FP CUDA cores
Tacker	T	Simultaneous execution of Tensor cores and INT CUDA cores
TC+IC+FC	T	Simultaneous execution of Tensor cores, INT and FP CUDA cores
VitBit (Ours)	T,C	INT packing with simultaneous execution of Tensor cores, INT and FP CUDA cores

We conduct experiments targeting INT8, one of the numeric formats commonly used in DNN model inference [8, 44]. VitBit packs two INT8 input values. It is worth noting that although VitBit utilizes INT8 in this paper, VitBit is applicable to the lower bitwidth integers, allowing for packing of up to 4 values as illustrated in Figure 3. Further analysis and research on these numeric formats will be conducted as part of future work.

Table 3 describes several evaluated methods that utilize CUDA cores and Tensor cores. *TC* represents the baseline methodology, which only executes Tensor cores for Tensor core kernels. *IC* executes INT CUDA cores only, serving as a baseline for CUDA core kernels. *FC* utilizes FP CUDA cores only by converting INT inputs to floating point data format. *IC+FC* enables simultaneous execution for INT and FP CUDA cores. *Tacker* fuses kernels from multiple workloads to utilize Tensor cores and CUDA cores in parallel [26]. As *Tacker* requires distinct kernels for fusion, we enable simultaneous execution of Tensor cores and CUDA cores during a single kernel execution for fair comparison. *TC+IC+FC* leverages INT, FP CUDA cores, and Tensor cores simultaneously. Our design, VitBit, combines *TC+IC+FC* technique with operand packing to further enhance arithmetic density. For CUDA core kernels, VitBit preprocesses inputs, and then executes INT and FP CUDA cores simultaneously.

4.2 Experimental Results

Performance: We evaluate performance using normalized inference time as the key metric. We conduct the experiments five times and calculate the arithmetic average of speedup over the baseline, *TC*. Figure 5 presents the experimental results of simultaneous execution solutions, including *TC*, *Tacker*, *TC+IC+FC*, and VitBit,

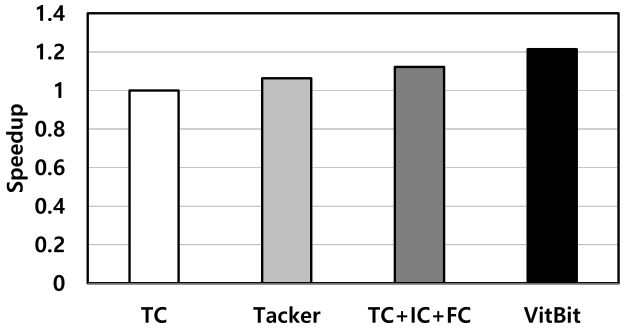


Figure 5: Inference time comparison according to the simultaneous execution techniques. All results are normalized to the baseline, *TC*. VitBit achieves a 1.22× speedup, while *Tacker* and *TC+IC+FC* show 1.06×, and 1.11× speedup, respectively

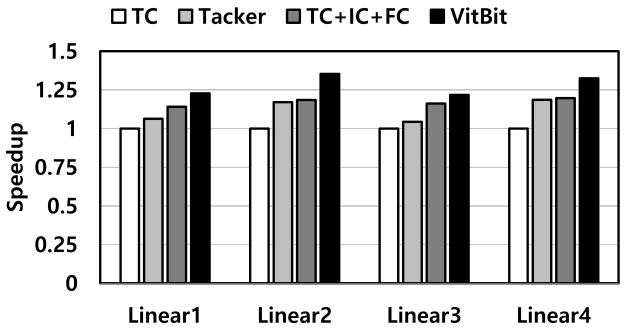


Figure 6: Execution time comparison of Linear kernels of ViT-Base model. All results are normalized to the baseline, *TC*. VitBit achieves the average speedup by 1.28× and the maximum speedup of 1.35× compared to the baseline with Linear kernels.

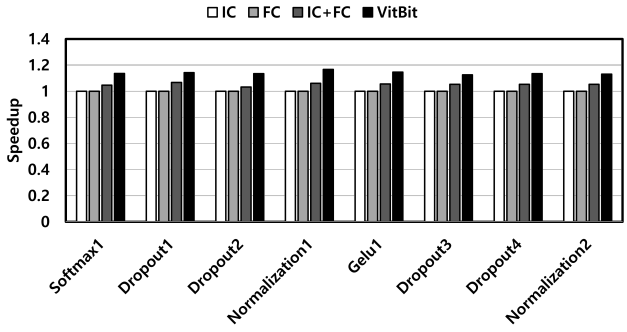


Figure 7: Execution time comparison of kernels excluding the linear kernels of the ViT-Base model. All results are normalized to the baseline, *IC*. VitBit achieves the average speedup by 1.14× compared to the *IC*, while *IC+FC* shows an average of 1.05× speedup.

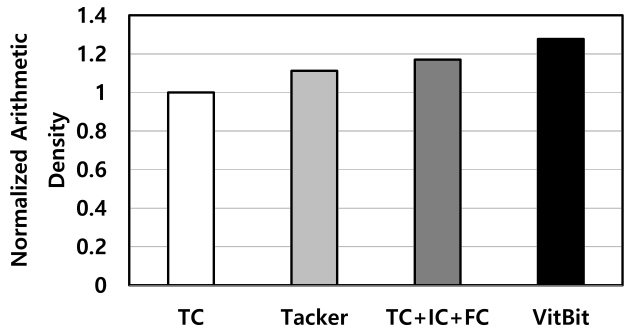


Figure 8: Arithmetic density while inference the ViT-Base model using VitBit. VitBit enhances arithmetic density by 1.28×, while *Tacker* and *TC+IC+FC* improve arithmetic density by 1.11× and 1.17×, respectively.

compared to the baseline *TC*. VitBit achieves a 1.22× speedup, while *Tacker* and *TC+IC+FC* show 1.06×, and 1.11× speedup, respectively. The simultaneous execution of CUDA cores and Tensor cores significantly enhances arithmetic density, resulting in the speedup. Particularly, VitBit outperforms *TC+IC+FC* by 1.1×, highlighting the effectiveness of the proposed operand packing policy.

ViT comprises repetitive attention blocks, each consisting of Linear, Softmax, Dropout, Normalization, and GeLU activation kernels [42]. Figure 6 demonstrates the speedup of Tensor Core kernels, which are Linear, composed of GEMM operations during ViT-Base inference. VitBit achieves the average speedup by 1.28× and the maximum speedup of 1.35× compared to the baseline with Linear kernels. As compute-intensive GEMM operations primarily constitute the Linear kernels, the simultaneous execution of Tensor cores and CUDA cores effectively accelerate Linear kernels.

For CUDA core kernels described in Section 3.3, VitBit executes INT and FP CUDA cores simultaneously with packed integers. Figure 7 depicts the speedup of CUDA core kernels for attention block. VitBit achieves the average speedup by 1.14× compared to the *IC and FC* with CUDA core kernels, while *IC+FC* shows an average of 1.05× speedup. Particularly, VitBit achieves 1.18× speedup by maximum, compared to *IC*, indicating that the operand packing is effective for CUDA core kernels.

Figure 8 exhibits the normalized arithmetic density over the baseline, *TC*. The simultaneous execution techniques consistently show improvement in arithmetic density compared to the baseline. VitBit enhances throughput during Vit-Base model, leading to improved arithmetic density. While VitBit enhances arithmetic density by 1.28×, *Tacker* and *TC+IC+FC* improve arithmetic density by 1.11× and 1.17×, respectively.

Instruction Count Comparison: Figure 9 illustrates the normalized Instruction Count compared to *IC+FC*. VitBit reduces the total instruction count for kernel execution by up to 1.5× compared to *IC+FC*. By packing multiple integer values and processing them simultaneously, VitBit successfully decreases the total instruction count for kernel execution. Reduction in instruction count contributes to accelerating kernel execution. Given that embedded GPUs typically suffer from limited core performance, VitBit is an outstanding solution for embedded GPUs.

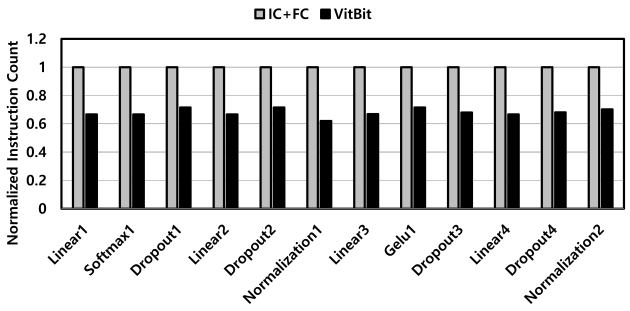


Figure 9: The number of instructions per layer within each layer while inference the ViT-Base model using ViTBit. VitBit reduces the total instruction count for kernel execution by up to 1.5× compared to *IC+FC*.

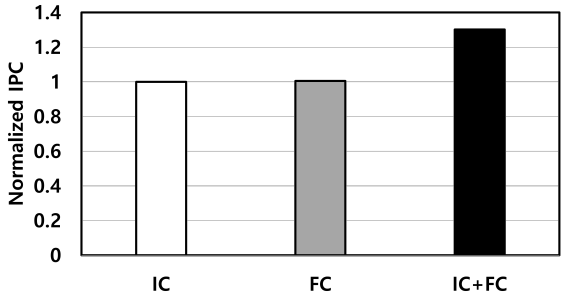


Figure 10: Average IPC of ViT-Base layers while inference the ViT-Base model using ViTBit. Utilizing both INT and FP CUDA cores results in a 1.3 × higher IPC than using either INT cores or FP cores solely.

Instructions Per Cycle: As VitBit efficiently utilizes all the arithmetic units in a GPU and reduces the instruction count, it eventually executes more instructions within a fixed time window compared to the baseline. To show our argument, we measure the average Instructions Per Cycle (IPC) while processing an inference. Figure 10 shows the experimental results. Utilizing both INT and FP CUDA cores with VitBit achieves an 1.3 × higher IPC than using either INT cores or FP cores. Leveraging both cores simultaneously enables processing more instructions within the same time, consequently leading to an overall speedup in inference time.

5 RELATED WORK

Various numeric formats for DNN workloads: Numerous prior work has utilized various bitwidths to run DNN workloads energy-efficiently [2–6, 12–14]. Some prior work has derived accuracy close to using FP32 by changing the precision within the floating point, such as FP8 or FP6 [2, 3, 13]. For example, N. Wang et al., has proposed chunk-based accumulation and floating point stochastic rounding techniques to mitigate accuracy challenges associated with using the FP8 format in back propagation [2]. The approach has enabled significant reductions in computation while maintaining the accuracy of DNN workloads. Similarly, various prior work has obtained performance close to using INT8 with formats such as INT2, INT3, and INT4 in the inference process or has obtained performance close to using FP32 with INT8 in training [4, 6]. HBFP has been proposed a new approach that uses both BFP and FP formats [12]. Using BFP for all operations leads to several challenges for running DNN workloads. Since DNN operations often result in tensors with wide value distributions, that can be too wide for BFP, leading to an accuracy loss. Additionally, BFP may increase costs due to numerous mantissa realignments and exponent computations depending on the characteristics of operations. Therefore, HBFP has used BFP for all dot product operations and FP for other operations.

Implementation of various numeric formats on GPU: There has been several prior work to use low bitwidth values in GPUs [24, 25]. X. Wang et al., has introduced a GPU register packing scheme that dynamically exploits narrow-width operands to pack multiple operands into a single full-width register [24]. The prior work has utilized the key insight that most computed results actually have fewer significant bits compared to the full width of a 32-bit register for many applications. The scheme enables high throughput as well as memory latency hiding in GPUs by exploiting massive thread-level parallelism. It is because the register packing technique utilizes register file space more efficiently, enabling the GPU to assign additional thread blocks on SMs. CORF has proposed a novel register file architecture that performs register coalescing by combining reads to multiple registers required by a single instruction into a single physical read [25]. The prior work has utilized the register packing technique to enable register coalescing. To increase the coalescing opportunities, the prior work has revised the physical register file to allow coalescing reads across different physical registers that reside in mutually exclusive sub-banks.

Simultaneous execution of CUDA cores and Tensor cores: H. Zhao st al., has introduced Tacker, which enables the simultaneous execution of CUDA cores and Tensor cores by fusing the kernels that use CUDA cores and Tensor cores, respectively [26]. Tacker has used both cores if different warps in a thread block of a kernel need to use the cores concurrently. To ensure QoS of applications while using the technique, the prior work has proposed accurate prediction modeling and has applied it to scheduling. K. Ho et al., has proposed a technique that offloads part of the GEMM operation from the Tensor core to the CUDA core to fully utilize GPU resources [27]. The technique has novelty in that it is purely hardware-based and does not require additional compiler or software support. Additionally, it also avoids the resource contention issue since only one kernel is running in one SM.

6 CONCLUSION

Traditional GPU architectures, designed to support a limited set of numeric formats, face challenges in meeting the diverse requirements of modern AI applications, which demand support for various numeric formats to optimize computational speed and efficiency. To address this challenge, we propose a novel software technique called VitBit, enabling efficient processing of arbitrary integer format values. The two key ideas are the packing of arbitrary integer formats for parallel computation and the simultaneous execution of Tensor cores, FP32 CUDA cores, and INT32 CUDA cores. VitBit achieves 1.35 *times* speedup for Tensor core kernels (e.g., GEMM)

compared to the baseline, which only utilizes Tensor cores. Furthermore, VitBit accelerates CUDA core kernels up to 1.18 *times* compared to the kernels leveraging INT cores only. Finally, VitBit achieves the overall 1.22× speedup and 1.28 × improvements in arithmetic density than the vanilla kernels of the quantized ViT-Base model.

REFERENCES

[1] J. Chee, Y. Cai, V. Kuleshov, and C. M. De Sa, "Quip: 2-bit quantization of large language models with guarantees," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[2] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan, "Training deep neural networks with 8-bit floating point numbers," *Advances in neural information processing systems*, vol. 31, 2018.

[3] J. Zhang, J. Yang, and H. Yuen, "Training with low-precision embedding tables," in *Systems for Machine Learning Workshop at NeurIPS*, vol. 2018, 2018.

[4] F. Zhu, R. Gong, F. Yu, X. Liu, Y. Wang, Z. Li, X. Yang, and J. Yan, "Towards unified int8 training for convolutional neural network," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 1969–1979.

[5] X. Sun, J. Choi, C.-Y. Chen, N. Wang, S. Venkataramani, V. V. Srinivasan, X. Cui, W. Zhang, and K. Gopalakrishnan, "Hybrid 8-bit floating point (hfp8) training and inference for deep neural networks," *Advances in neural information processing systems*, vol. 32, 2019.

[6] Y. Li, S. Xu, B. Zhang, X. Cao, P. Gao, and G. Guo, "Q-vit: Accurate and fully quantized low-bit vision transformer," *Advances in neural information processing systems*, vol. 35, pp. 34 451–34 463, 2022.

[7] Z. Li, X. Liu, J. Zhang, and Q. Gu, "Repquant: Towards accurate post-training quantization of large transformer models via scale reparameterization," *arXiv preprint arXiv:2402.05628*, 2024.

[8] Y. Lin, T. Zhang, P. Sun, Z. Li, and S. Zhou, "Fq-vit: Post-training quantization for fully quantized vision transformer," *arXiv preprint arXiv:2111.13824*, 2021.

[9] N. Ranjan and A. Savakis, "Lrp-qvit: Mixed-precision vision transformer quantization via layer-wise relevance propagation," *arXiv preprint arXiv:2401.11243*, 2024.

[10] Z. Yuan, C. Xue, Y. Chen, Q. Wu, and G. Sun, "Ptq4vit: Post-training quantization for vision transformers with twin uniform quantization," in *European conference on computer vision*. Springer, 2022, pp. 191–207.

[11] B. D. Rouhani, R. Zhao, A. More, M. Hall, A. Khodamoradi, S. Deng, D. Choudhary, M. Cornea, E. Dellinger, K. Denolf *et al.*, "Microscaling data formats for deep learning," *arXiv preprint arXiv:2310.10537*, 2023.

[12] M. Drumond, T. Lin, M. Jaggi, and B. Falsafi, "Training dnns with hybrid block floating point," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[13] H. Xia, Z. Zheng, X. Wu, S. Chen, Z. Yao, S. Youn, A. Bakhtiari, M. Wyatt, D. Zhuang, Z. Zhou *et al.*, "Fp6-llm: Efficiently serving large language models through fp6-centric algorithm-system co-design," *arXiv:2401.14112*, 2024.

[14] S.-y. Liu, Z. Liu, X. Huang, P. Dong, and K.-T. Cheng, "Llm-fp4: 4-bit floating-point quantized transformers," *arXiv preprint arXiv:2310.16836*, 2023.

[15] Y. Sekikawa and S. Yashima, "Bit-pruning: A sparse multiplication-less dot-product," in *The Eleventh International Conference on Learning Representations*, 2022.

[16] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks," *arXiv preprint arXiv:1803.03635*, 2018.

[17] T. Chen, X. Chen, X. Ma, Y. Wang, and Z. Wang, "Coarsening the granularity: Towards structurally sparse lottery tickets," in *International conference on machine learning*. PMLR, 2022, pp. 3025–3039.

[18] H. Yang, W. Wen, and H. Li, "Deephoyer: Learning sparser neural network with differentiable scale-invariant sparsity measures," *arXiv preprint arXiv:1908.09979*, 2019.

[19] C. Blakeney, X. Li, Y. Yan, and Z. Zong, "Parallel blockwise knowledge distillation for deep neural network compression," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 7, pp. 1765–1776, 2020.

[20] J. Johnson, "Rethinking floating point for deep learning," *arXiv preprint arXiv:1811.01721*, 2018.

[21] Z. Carmichael, H. F. Langroudi, C. Khazanov, J. Lillie, J. L. Gustafson, and D. Kudithipudi, "Performance-efficiency trade-off of low-precision numerical formats in deep neural networks," in *Proceedings of the conference for next generation arithmetic 2019*, 2019, pp. 1–9.

[22] V. Radu, K. Kaszyk, Y. Wen, J. Turner, J. Cano, E. J. Crowley, B. Franke, A. Storkey, and M. O'Boyle, "Performance aware convolutional neural network channel pruning for embedded gpus," in *2019 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2019, pp. 24–34.

[23] E. Jeong, J. Kim, and S. Ha, "Tensorrt-based framework and optimization methodology for deep learning inference on jetson boards," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 21, no. 5, pp. 1–26, 2022.

[24] X. Wang and W. Zhang, "Gpu register packing: Dynamically exploiting narrow-width operands to improve performance," in *2017 IEEE Trustcom/BigDataSE/ICESS*. IEEE, 2017, pp. 745–752.

[25] H. Asghari Esfeden, F. Khorasani, H. Jeon, D. Wong, and N. Abu-Ghazaleh, "Corf: Coalescing operand register file for gpus," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 701–714.

[26] H. Zhao, W. Cui, Q. Chen, Y. Zhang, Y. Lu, C. Li, J. Leng, and M. Guo, "Tacker: Tensor-cuda core kernel fusion for improving the gpu utilization while ensuring qos," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 800–813.

[27] K. Ho, H. Zhao, A. Jog, and S. Mohanty, "Improving gpu throughput through parallel execution using tensor cores and cuda cores," in *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2022, pp. 223–228.

[28] J. You, J.-W. Chung, and M. Chowdhury, "Zeus: Understanding and optimizing {GPU} energy consumption of {DNN} training," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 119–139.

[29] L. Cai, A.-M. Barneche, A. Herbout, C. S. Foo, J. Lin, V. R. Chandrasekhar, and M. M. S. Aly, "Tea-dnn: the quest for time-energy-accuracy co-optimized deep neural networks," in *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, 2019, pp. 1–6.

[30] X. Wang and W. Zhang, "Energy-efficient dnn computing on gpus through register file management," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–7.

[31] S. Choi, I. Koo, J. Ahn, M. Jeon, and Y. Kwon, "{EnvPipe}: Performance-preserving {DNN} training framework for saving energy," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023, pp. 851–864.

[32] L. S. Karumbunathan, *NVIDIA Jetson AGX Orin Series: A Giant Leap Forward for Robotics and Edge AI Applications*, NVIDIA Corporation, July 2022.

[33] S. Burel, A. Evans, and L. Anghel, "Mozart: Masking outputs with zeros for architectural robustness and testing of dnn accelerators," in *2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS)*. IEEE, 2021, pp. 1–6.

[34] ——, "Mozart+: Masking outputs with zeros for improved architectural robustness and testing of dnn accelerators," *IEEE Transactions on Device and Materials Reliability*, vol. 22, no. 2, pp. 120–128, 2022.

[35] A. M. Radaideh and P. V. Gratz, "Exploiting zero data to reduce register file and execution unit dynamic power consumption in gpgpus," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[36] J. Choquette, W. Gandhi, O. Giroux, N. Stam, and R. Krashinsky, "Nvidia a100 tensor core gpu: Performance and innovation," *IEEE Micro*, vol. 41, no. 2, pp. 29–35, 2021.

[37] D. Ha, Y. Oh, and W. W. Ro, "R2d2: Removing redundancy utilizing linearity of address generation in gpus," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23, 2023.

[38] G. Wang, Y. Lin, and W. Yi, "Kernel fusion: An effective method for better power efficiency on multithreaded gpu," in *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*. IEEE, 2010, pp. 344–350.

[39] NVIDIA. (2024) Cublas library. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUBLAS_Library.pdf

[40] Huggingface. (2024) Transformers. [Online]. Available: https://github.com/huggingface/transformers/tree/main/examples/pytorch

[41] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[42] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.

[43] H. Zhang, Y. Zhou, Y. Xue, Y. Liu, and J. Huang, "G10: Enabling an efficient unified gpu memory and storage architecture with smart tensor migrations," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 395–410.

[44] Z. Li and Q. Gu, "I-vit: integer-only quantization for efficient vision transformer inference," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2023, pp. 17 065–17 075.