Accelerating Applications using Edge Tensor Processing Units

Kuan-Chieh Hsu
University of California, Riverside
Riverside, California, USA

Hung-Wei Tseng
University of California, Riverside
Riverside, California, USA

ABSTRACT

Neural network (NN) accelerators have been integrated into a wide-spectrum of computer systems to accommodate the rapidly growing demands for artificial intelligence (AI) and machine learning (ML) applications. NN accelerators share the idea of providing native hardware support for operations on multidimensional tensor data. Therefore, NN accelerators are theoretically tensor processors that can improve system performance for any problem that uses tensors as inputs/outputs. Unfortunately, commercially available NN accelerators only expose computation capabilities through AI/ML-specific interfaces. Furthermore, NN accelerators reveal very few hardware design details, so applications cannot easily leverage the tensor operations NN accelerators provide.

This paper introduces General-Purpose Computing on Tensor Processing Units (GPTPU), an open-source, open-architecture framework that allows the developer and research communities to discover opportunities that NN accelerators enable for applications. GPTPU includes a powerful programming interface with efficient runtime system-level support—similar to that of CUDA/OpenCL in GPGPU computing—to bridge the gap between application demands and mismatched hardware/software interfaces.

We built GPTPU machine uses Edge Tensor Processing Units (Edge TPUs), which are widely available and representative of many commercial NN accelerators. We identified several novel use cases and revisited the algorithms. By leveraging the underlying Edge TPUs to perform tensor-algorithm-based compute kernels, our results reveal that GPTPU can achieve a 2.46× speedup over high-end CPUs and reduce energy consumption by 40%.

ACM Reference Format:
Kuan-Chieh Hsu and Hung-Wei Tseng. 2021. Accelerating Applications using Edge Tensor Processing Units. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3458817.3476177

1 INTRODUCTION

The demand for artificial intelligence (AI) and machine learning (ML) applications has exploded in recent years, and the increase in AI/ML workloads has led to significant research advances in neural network (NN) accelerators, including Google's Edge Tensor Processing Units (Edge TPUs) [1] and Apple's Neural Engines [2] that are already presented as auxiliary hardware components in commodity systems. These NN accelerators' power/energy efficiency is orders-of-magnitude better than that of conventional vector processors (e.g., Graphics Processing Units [GPUs]) for the same workloads. Despite the differences among microarchitectures, most NN accelerators are essentially matrix processors that take tensors/matrices as inputs, generate tensors/matrices as outputs, and provide operators that facilitate NN computations.

Two decades ago, graphics processing units (GPUs) were just domain-specific accelerators used for shading and rendering. But intensive research into high-performance algorithms, architectures, systems, and compilers [3–12] and the availability of frameworks like CUDA [13] and OpenCL [14], have revolutionized GPUs and transformed them into high-performance, general-purpose vector processors. We expect a similar revolution to take place with NN accelerators—a revolution that will create general-purpose matrix processors for a broader spectrum of applications. However, democratizing these NN accelerators for non-AI/ML workloads will require the system framework and the programmer to tackle the following issues:

(1) The microarchitectures and instructions of NN accelerators are optimized for NN workloads, instead of general matrix/tensor algebra. These auxiliary NN accelerators focus on latency per inference, but not yet on delivering computation throughput comparable to GPUs. Naively mapping conventional matrix/tensor algorithms to AI/ML operations will lead to sub-optimal performance. (2) Because many AI/ML applications are error tolerant, NN accelerators typically trade accuracy for area/energy-efficiency; when such a trade-off produces undesirable results, additional mechanisms are needed to make adjustments. (3) The programming interfaces of existing NN accelerators are specialized for developing AI/ML applications. Existing frameworks expose very few details about the hardware/software interfaces of NN accelerators, so programmers are unable to customize computation and the application can suffer from significant performance overhead due to adjusting the parameters/data bound to the supported ML models. (4) Tensor algorithms are traditionally time-consuming, so programmers have tailored compute kernels in favor of scalar/vector processing. Such tailoring makes applications unable to take advantage of tensor operators without revisiting algorithms.

This paper bridges the gap between general-purpose programmability and domain-specific NN accelerators by presenting a full-stack system architecture that enables General-Purpose Computing on Edge TPUs (GPTPU). GPTPU tackles all the aforementioned challenges through providing a programming interface, a runtime system, compiler and libraries. With the system this paper proposes, programmers will be able to explore the enormous potential of the matrix processing model inherent in Edge TPU, a commercially available accelerator that can be part of a system-on-module (SOM) or be easily attached to various forms of computer systems. A commercialized Edge TPU can inference ML models at 4 TOPS (tera

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SC '21, November 14–19, 2021, St. Louis, MO, USA
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8442-1/21/11.
https://doi.org/10.1145/3458817.3476177

operations per second) with only 2 W of power consumption. The design that GPTPU demonstrates can also work for NN accelerators sharing similar architectures.

GPTPU provides a programming framework, including an Edge TPU-specific C/C++ extension, OpenCtpu and a runtime system. GPTPU offloads programmers from directly interacting with the accelerator's hardware to focus on the design of tensor-based algorithms for applications. OpenCtpu achieves more programming flexibility than existing domain-specific interfaces by exposing high-level tensor/matrix algebra operators (e.g., matrix multiply) and low-level accelerator operators (e.g., convolution and multiplication-accumulation) to the programmer, so programmers can design arbitrary tensor algorithms or customize operators that cannot be easily achieved using domain-specific languages.

The core of the GPTPU runtime system is our proposed *Tensorizer*, a module that dynamically evaluates input data and transforms data into ML models that the underlying Edge TPUs or other NN accelerators can efficiently perform inference operations on. Tensorizer handles quantization and calibration of input datasets and computation results, thereby minimizing the impact of limited precision on NN accelerators. The GPTPU runtime also schedules computation tasks and distributes prepared data to available NN accelerators in a manner that minimizes the data movement overhead.

Despite the Edge TPU's promising energy efficiency and recently open-sourced C++ API, documentation is vague regarding the Edge TPU's hardware/software interface and architecture. This lack of detail complicates the design of systems that fully exploit the Edge TPU's capabilities. To develop GPTPU, we measured the performance of available Edge TPU operators, reverse-engineered the Edge TPU hardware/software interface for data exchanges, and analyzed the Edge TPU architecture. We applied our understanding of the Edge TPU to optimize the backend runtime system for efficient task creation and data transformation. We then built a prototype GPTPU system with 8 Edge TPUs to allow concurrent GPTPU task execution.

We demonstrate the potential of the GPTPU system by modifying several key applications for financial computing, linear algebra, physics simulations and graph analytics. By revisiting the algorithms at the heart of these applications and using OpenCtpu, we show that GPTPU can simplify compute kernels; GPTPU preserves the nature of the application's tensor/matrix inputs and avoids explicit decompositions of datasets and algorithms into vector or scalar data. When used with the GPTPU-integrated applications, our prototype GPTPU system exhibits a 2.46× speedup and 40% reduction in energy consumption relative to modern CPUs.

By introducing the GPTPU system architecture, this paper makes the following contributions: (1) The paper introduces a novel full-stack system architecture to efficiently support general-purpose programming on Edge NN accelerators. (2) The paper characterizes the capabilities and previously unidentified architectural details of an inferencing hardware so that future research may exploit and optimize the GPTPU concept. (3) The paper proposes and implements Tensorizer to demonstrate the mechanism of dynamically and transparently mapping operators to NN models and Edge TPU instructions that lead to efficient use of underlying NN accelerators. (4) The paper demonstrates algorithm design for non-NN applications on NN accelerators by revisiting application algorithms to wisely use available accelerator instructions. (5) The paper provides an open-source framework working on commercially available components that will allow the community to reproduce the proposed system architecture and explore additional applications on the GPTPU platform [1]. (6) The paper shows the performance and energy-consumption benefits of the prototype GPTPU system.

2 BACKGROUND

This section briefly highlights TPU architectures and introduces alternative NN accelerators where GPTPU can potentially work.

2.1 TPU Architecture

As most NN applications take matrix/tensor inputs and iteratively update parameters/weights from previous outcomes, the TPU microarchitecture accelerates NN tasks for modern ML applications by creating a systolic array that performs operations on the units of matrices/tensors. For inferencing tasks, the TPU treats one of the input matrices as the trained model and the other as the samples to predict or classify. Taking matrices/tensors as the default inputs and outputs makes the TPU architecture and its corresponding execution model fundamentally different from conventional CPU/GPU architectures that compute on scalar/vector data pairs. TPUs also incorporate large on-chip memory to hold the intermediate results that later iterations reuse. Unlike conventional processors, TPUs do not contain on-chip instruction caches but simply use a CISC-style instruction-set architecture and rely on the host program to issue instructions through the system interconnect. And whereas conventional processors aim for precise computation results, TPU matrix units only support operations on a limited level of precision that is sufficient to satisfy the demands of modern ML applications while significantly reducing both TPU costs and energy requirements.

2.2 Edge TPU

This paper uses Edge TPUs, the trimmed-down versions of the Google Cloud TPU to build our system. Compared with Cloud versions, Edge TPUs contain smaller data memory (i.e., 8 MB). The documented peak TOPS of Edge TPU is 4 TOPS under a 2 W TDP, while Cloud TPUs can achieve up to 90 TOPS under a 250 W TDP.

Although Google Cloud TPUs offer higher performance, we chose the Edge TPUs for the following reasons: (1) The Edge TPU hardware is publicly available, whereas Cloud TPU hardware is available exclusively through Google services; our use of Edge TPUs will therefore allow the developer and research communities to easily replicate, utilize, and optimize the system that this paper describes. (2) The current version of the Edge TPU software framework has a partially open-sourced C++ backend that enables language front-end and runtime-system customization, whereas the Google Cloud TPU only provides a TensorFlow front-end without the backend source code. (3) Each Edge TPU offers better performance per watt than Cloud TPUs (i.e., 2 TOPS/W v.s. 0.36 TOPS/W) with just 2 W of power consumption and costs as little as USD 29, making a platform like GPTPU applicable to a broader range of

[1]The entire codebase of GPTPU is available in a public repository (https://github.com/escalab/GPTPU).

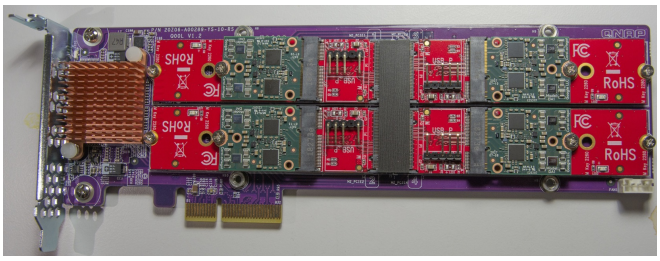

Figure 1: The custom-built quad-EdgeTPU PCIe card

computer systems than would be possible with the Google Cloud TPU alone.

2.3 Alternatives to TPUs

In addition to TPUs, several other hardware-accelerator architectures can improve tasks in AI/ML workloads. HyGCN [15], Caffeine [16], Chain-NN [17], Tensaurus [18], Eyeriss [19], Tangram [20], SNNAP [21], AccPar [22], Wei et al. [23], and Kung et al. [24] all adopt a systolic-array-based design, just as TPUs do.

DianNao [25, 26], MAERI [27], Cambricon [28], FlexFlow [29], ScaleDeep [30], MnnFast [31], TIE [32], UCNN [33], CirCNN [34], HyPar [35], Park et al. [36], Sharma et al. [37], Alwani et al. [38], Song et al. [39], Shortcut Mining [40], VIP [41], and Simba [42] focus on memory bandwidth optimizations, chip layout, data reuses, workload balancing, and reducing inter-chiplet communications in their NN-accelerator architectures.

Recent advances in near-data/in-memory processing now allow data-intensive NN computation to occur without explicitly moving data through bandwidth-limited system interconnects. Neural Cache [43], TensorDIMM [44], Manna [45], DRISA [46], TETRIS [47], NAND-Net [48], SCOPE [49], Wang et al. [50], Liu et al. [51], and Imani et al. [52] place logic in memory controllers for volatile SRAM/DRAM memory modules. FPSA [53], and LerGAN [54], Sparse ReRAM engine [55], PRIME [56], PipeLayer [57], PUMA [58], Bojnordi et al. [59], Zhang et al. [60], and FloatPIM [61] use the physical characteristics of resistive random-access memory (ReRAM) technologies to create NN accelerators.

Regrettably, the aforementioned academic NN accelerators are not currently in production. And commercially available ML accelerators such as Khadas VIM3 [62], Rockchip RK1808 [63], Sophon BM1880 [64], HiKey970 [65], and Jetson Nano [66] lack the performance and energy-efficiency of Edge TPUs.

Though NN accelerators have different microarchitectures, they all use the tensor/matrix as the basic unit of processing, and they all have limited precision, just like Edge TPUs. The architecture, programming interface, methods, and policies embodied in GPTPU can easily be adapted to different NN accelerators as long as the accelerators expose their instructions appropriately.

3 CHARACTERIZING EDGE TPUS

To directly measure the characteristics of Edge TPUs and determine their performance numbers, we built a customized machine with Edge TPUs attached. This section describes the architecture of our GPTPU testbed and reports the key performance characteristics of Edge TPUs that serve as the basis for our GPTPU system and application designs.

3.1 The prototype Edge TPU accelerated machine

The TPU architecture relies heavily on the CPU to distribute instructions, so our custom-built GPTPU hardware prototype aims at minimizing communication latency while efficiently using the limited system-interconnect bandwidth. To achieve this goal, the GPTPU prototype machine uses Edge TPUs in PCIe M.2 form factors; the Edge TPUs are attached directly to the PCIe system interconnect to allow lower latency and better bandwidth compared to other Edge TPU interconnect options, such as USB 3.0.

Each M.2 Edge TPU is designed to occupy only a single PCIe 2.0 lane, whereas most expansion slots that physically connect to the processor use multiple lanes. To efficiently use the precious PCIe lanes from the processor and the limited expansion slots, Figure 1 shows our custom-built quad-EdgeTPU PCIe expansion cards using QNAP QM2-4P-384A [67]. Each quad-EdgeTPU PCIe card contains 4× M.2 Edge TPUs with M.2 slots connected to a PCIe switch. The PCIe switch evenly divides the PCIe lanes (attached to each expansion slot) to four Edge TPUs and makes all Edge TPUs available to the host processor.

The current GPTPU hardware prototype contains an AMD Ryzen 3700X CPU with a Matisse microarchitecture that can reach a max-boost clock speed of 4.4 GHz with 32 MB LLC and 24× PCIe lanes available to all peripherals. Excluding the expansion slots used for essential peripheral devices, our hardware prototype can host 8× M.2 Edge TPUs, and each Edge TPU connects to the processor with just one hop (i.e., the PCIe switch) in the middle. The machine also contains 64 GB DDR4 main memory and an NVMe SSD. In addition to the hardware specifications, the prototype machine runs Ubuntu Linux 16.04 with kernel version 4.15.

3.2 Characterizing Edge TPU instructions

Due to the long interconnect latency and the absence of instruction caches on Edge TPUs, coped with the variable number of cycles and different types of input/output data resulting from the available CISC instructions, the GPTPU library, runtime system, and applications must use Edge TPU instructions wisely to achieve the best performance. The released Edge TPU performance numbers are available only in TOPS (tera operations per second) and IPS (inferences per second). However, neither TOPS nor IPS provides sufficient insight for general-purpose software design because (1) TOPS or IPS is highly task specific, and (2) IPS is only representative for inferencing but not for other workloads [68].

We therefore use the *RPS (results per second)* as an additional metric to assess the benefits of each available Edge TPU operator/instruction. We define RPS as the amount of final result values an Edge TPU can generate within a second. We measure the OPS, RPS, and data-exchange rate of each tensor arithmetic instruction as follows: First, we begin timing the program and send the input datasets with size s to the target Edge TPU. Second, we issue and execute the same operator 10,000 times and measure the end-to-end latency (t_1) as well as the total number of result values (r_1) generated since the timer started. Third, we repeat the first and

Operator	OPS (ops per second)	RPS (results per second)	Description
conv2D	10268.80	168240326.89	2D Convolution on a matrix
FullyConnected	51924.96	6646394.57	Input vector multiplies a weight matrix
sub	6273.28	82871343.60	Pair-wise subtraction on two matrices
add	6203.52	98293633.48	Pair-wise addition on two matrices
mul	14515.84	216469999.54	Pair-wise multiplication on two matrices
crop	4867.96	1562904391.76	Remove all unwanted elements outside of a sub-matrix from a given 2D matrix and return the sub-matrix
ext	1604.78	3637240203.38	Pad a matrix to the target dimensionality and return the padded matrix
mean	408.54	408.54	Count the average value of all elements in the matrix
max	477.08	477.08	Find the maximum value within a matrix
tanh	3232.31	2148232470.28	Perform tanh function on a matrix pair-wisely
ReLu	11194.26	4043196115.38	Leave only non-zero values on a matrix pair-wisely

Table 1: The maximum OPS and RPS for each Edge TPU operator/instruction

second step, but this time we execute the operator 20,000 times with the same input to get the end-to-end latency (t_2) and the total number of generated result values (r_2). Finally, we calculate the OPS of instructions/operators using Equation 1, their RPSs using Equation 2, and the data-exchange rate using Equation 5.

$$OPS_{operation} = \frac{10,000}{t_2 - t_1} \tag{1}$$

$$RPS_{operation} = \frac{r_2 - r_1}{t_2 - t_1} \tag{2}$$

$$Data\text{-}Exchange\ Rate = \frac{s}{t_1 - (t_2 - t_1)} \tag{3}$$

Table 1 lists the RPS and the OPS of each Edge TPU instruction. The results lead to three observations on Edge TPUs. (1) `Conv2D` (convolution) achieves a very high RPS given the high amount of operations required in equivalent CPU/GPU implementations, a hint that Edge TPU optimizes its microarchitecture for this instruction, (2) the OPS and RPS vary significantly for different types of instructions, and (3) the OPS and RPS are not strongly correlated because output varies; for example, instructions like `sub` generate outputs with the same dimensions as their inputs, but instructions like `FullyConnected` only produce vectors.

Our measurements also show that data-exchange performance does not vary among different types of instructions, but simply correlates with data size; transmitting 1 MB of data to an Edge TPU takes around 6 ms, while transmitting 8 MB of data that completely fill the on-chip memory takes 48 ms. The latency of copying data between the host main memory and Edge TPU's on-chip memory is significantly longer than any Edge TPU instruction.

3.3 Characterizing Edge TPU data and model formats

Edge TPU instructions ordinarily take two types of data inputs: (1) a tensor used for input datasets to be inferenced and (2) a model that the TFLite framework must generate and compile. Both types of inputs must be quantized before the host program sends them to the Edge TPU for computation. As GPTPU needs to use both types of inputs to achieve general-purpose computing, the GPTPU runtime library must translate one of the instruction inputs as a model for the Edge TPU.

The current Edge TPU toolchain only allows the user to generate models by invoking the Edge TPU compiler in the Python-based TFLite. With TFLite, translating a 2K × 2K matrix into a model takes 2.7 seconds on our testbed. This latency does not create issues for inferencing tasks in ML applications, as inferencing tasks tend to reuse the same model for continuously changing inputs, and the overhead of creating models is amortized as input datasets scale. However, such amortization does not stand for many applications outside the ML realm. Unfortunately, neither the Edge TPU compiler code nor the Edge TPU model encoding has been released, so we have been unable to optimize the Edge TPU model-creation overhead.

To compensate for this lack of information, we reverse-engineered the Edge TPU model formats by creating models with different inputs, dimensions, and value ranges. We examined the models generated with the different inputs, and we identified the following key characteristics that allowed us to optimize the GPTPU runtime-system Edge TPU model-input instructions: (1) Models embed a 120-byte general header that allows TPUs to recognize the model-format version. The last 4 bytes of the header contain an unsigned integer describing the size of the data section. (2) Following the header, the data section contains binary-encoded 8-bit integers stored in row-major order. If the raw data values exceed the scope of 8-bit integers or are non-integers, the values must be scaled to fit in the 8-bit integer range. (3) A metadata section following the data section describes the data-section dimensions in terms of rows and columns. The metadata section also contains the scaling factor (f), a floating-point number that the compiler uses to rescale raw data into 8-bit integers; that is, an 8-bit integer value in the data section is calculated by multiplying its raw value by f. (4) The model encodes all values using little endian.

In addition to making the above observations, we determined that data dimensions do not necessarily reflect the dimensions of raw data inputs. The Edge TPU compiler adds zero padding to unused elements (depending on the instructions) to reflect the hardware microarchitecture. Taking the most optimized instruction in Edge TPU architecture as an example, the Edge TPU compiler reshapes all input data into 128×128 sub-matrices. This implies that the Edge TPU's matrix unit is designed for computing on 128×128×8-bit matrices, in contrast to the Cloud TPU matrix unit, which is designed for 256×256×8-bit matrices.

4 OVERVIEW OF THE GPTPU SYSTEM

Using insights learned from Section 3, this paper presents the system stack of the GPTPU framework that Figure 2 shows. GPTPU maintains the original heterogeneous-computing system stack and extends the programming-language front end. GPTPU also provides a system library that can trigger the runtime system to (1) transform data, (2) schedule instructions for the underlying TPU

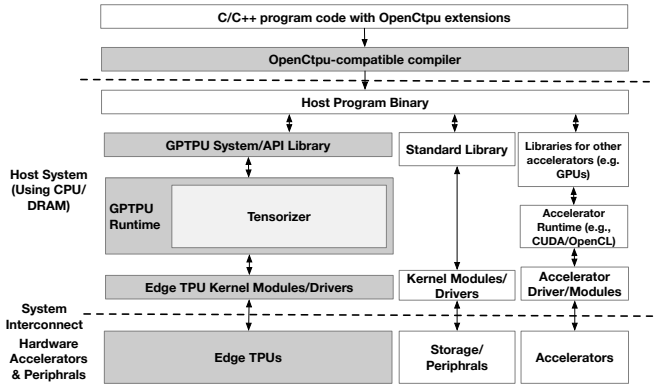

Figure 2: The GPTPU system overview

hardware, (3) communicate with the TPU hardware, and (4) use the TPU hardware to accomplish computation tasks.

OpenCtpu serves as the programming-language front end for GPTPU. A programmer can use OpenCtpu to create a host program that describes TPU tasks and coordinates the use of heterogeneous computing resources and data exchanges in the system. A compiler supporting the OpenCtpu extensions will generate machine binaries compatible with the host CPU architecture and will generate code that transfers control to the GPTPU runtime system.

The GPTPU runtime system coordinates available TPU hardware. The runtime system schedules TPU operations from programmer-defined TPU tasks and prepares the inputs/outputs for TPU operations. Task scheduling and data preparation are left to the runtime system because doing so allows the GPTPU system to (1) adapt to changes in the underlying hardware without the need for reprogramming, (2) flexibly utilize underlying hardware resources, and (3) unburden the programmer of hardware-limitation details (e.g., data precision).

The following sections describe the design of the OpenCtpu programming-language front end (Section 5), the GPTPU runtime system (Section 6), and optimized operators/library function/applications (Section 7).

5 OPENCTPU—THE GPTPU PROGRAMMING INTERFACE

OpenCtpu is a C/C++ extension for general-purpose programming with GPTPU. OpenCtpu shares similarities with popular GPU programming models like CUDA [13] and OpenCL [14] in that OpenCtpu (1) places the control of application flow and device usage on the CPU-based host, (2) leverages virtual memory abstraction so that applications can specify data locations, (3) requires the programmer to explicitly manage data buffers for TPUs, and (4) provides functions that enable programmers to describe computation tasks for computation on TPUs.

A programmer can use OpenCtpu API functions and the C/C++ standard library to compose a TPU-accelerated program (see Table 2 for a list of representative OpenCtpu API functions). To create tasks for TPUs with the OpenCtpu API functions, a program needs to have the following: (1) kernel functions that describe the desired computation for TPUs, (2) input/output data buffers/structures for TPU kernels, and (3) enqueuing kernel functions and their

```c
#include <stdio.h>
#include <stdlib.h>
#include <gptpu.h>

// The TPU kernel
void *kernel(openctpu_buffer *matrix_a,
             openctpu_buffer *matrix_b,
             openctpu_buffer *matrix_c)
{
    // invoke the TPU operator
    openctpu_invoke_operator(conv2D, SCALE, matrix_a, \
                                            matrix_b, matrix_c);
    return 0;
}

int main(int argc, char **argv)
{
    float *a, *b, *c; // pointers for raw data
    openctpu_dimension *matrix_a_d, *matrix_b_d, *matrix_c_d;
    openctpu_buffer * tensor_a, * tensor_a, * tensor_a;
    int size; // size of each dimension

    // skip: data I/O and memory allocation/initialization

    // describe a 2-D tensor (matrix) object for a
    matrix_a_d = openctpu_alloc_dimension(2, size, size);
    // describe a 2-D tensor (matrix) object for b
    matrix_b_d = openctpu_alloc_dimension(2, size, size);
    // describe a 2-D tensor (matrix) object for c
    matrix_c_d = openctpu_alloc_dimension(2, size, size);

    // create/fill the tensor a from the raw data
    tensor_a = openctpu_create_buffer(matrix_a_d, a);
    // create/fill the tensor b from the raw data
    tensor_b = openctpu_create_buffer(matrix_b_d, b);
    // create/fill the tensor c from the raw data
    tensor_c = openctpu_create_buffer(matrix_c_d, c);

    // enqueue the matrix_mul TPU kernel
    openctpu_enqueue(kernel, tensor_a, tensor_b, tensor_c);
    // synchronize/wait for all TPU kernels to complete
    openctpu_sync();

    // skip: the rest of the program
    return 0;
}
```

Figure 3: An OpenCtpu code sample

inputs/outputs as tasks (OpenCtpu is similar to OpenCL in this respect). In the OpenCtpu programming model, all TPU operations within a task (i.e., an instance of a TPU kernel function) will perform in serial, but tasks can perform out of order in parallel. Therefore, the programmer may need to invoke synchronized primitives to ensure execution order and task completion.

To use Edge TPU operators in the kernel function, OpenCtpu provides an API function `openctpu_invoke_operator`. As the runtime system handles the precision, the programmer simply needs to specify the desired quantization method. In addition to `openctpu_invoke_operator` that directly invoke Edge TPU instructions, OpenCtpu also implemented optimized overloaded operators on tensor data (e.g., matrix-add [+], matrix-sub [-], matrix-multiply [*]) to perform pair-wise matrix addition, subtraction and multiplication to further simplify programming.

The current OpenCtpu design brings several benefits to the GPTPU system. First, OpenCtpu gives the runtime system the flexibility to schedule and execute parallel tasks and to control the data movements associated with each task. Second, OpenCtpu avoids hardware complexity related to managing data consistency/coherency; OpenCtpu does this by leaving data management to software, as with GPGPU programming models. Third, OpenCtpu is designed to be complementary to existing heterogeneous computing platforms (we have verified that CUDA/OpenCL are compatible with our

Synopsis	Description
`openctpu_dimension *openctpu_alloc_dimension(int dimensions, ...)`	This function allocates an `openctpu_dimension` data structure that describes the dimensionality of data in an input/output buffer. Depending on the input value of `dimensions`, the function can accept additional parameters that describe the dimensions.
`openctpu_buffer_t *openctpu_create_buffer( openctpu_dimension *dimension, void *data, unsigned flags)`	This function creates an input data buffer for TPU kernels. The pointer `dimension` provides a data structure with information about the number of data elements, the data type, and the dimensionality of the data. The pointer `data` provides the address for the raw data. The `openctpu_buffer_t` function returns a pointer to the created buffer.
`int *openctpu_enqueue(void *(*func)(void *), ...)`	This function enqueues a TPU task described in `func`. In addition to `func`, this function can accept an arbitrary number of arguments as `func` parameters. The function returns a task ID for the enqueued task.
`int *openctpu_invoke_operator(enum tpu_ops op, unsigned flags, ...)`	This function invokes a supported TPU operator (with operator arguments) and returns the operator output. The `flags` consist of parameters like the quantization method.
`int *openctpu_sync()`	This synchronization function requires all TPU tasks to complete before it returns.
`int *openctpu_wait(int task_id)`	This function blocks the calling thread until the specified task returns.

Table 2: Sample functions from the OpenCtpu API

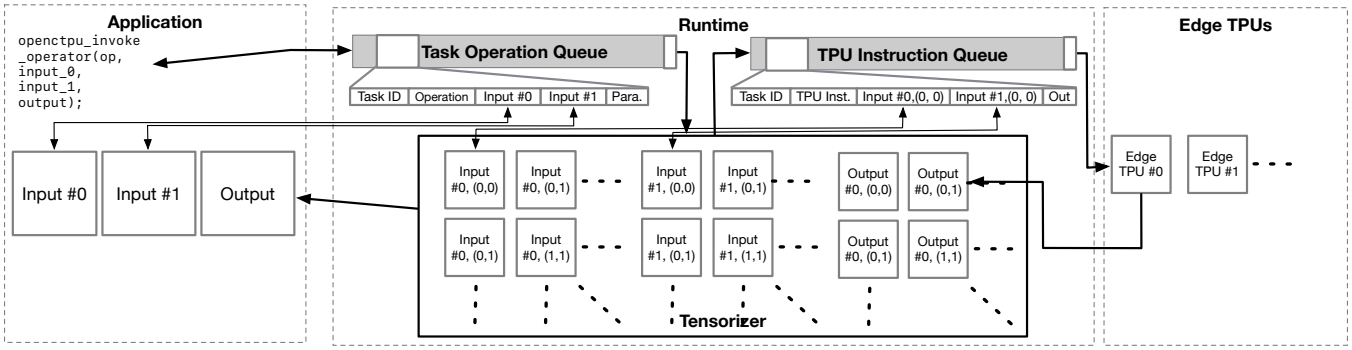

Figure 4: An overview of GPTPU's runtime system.

OpenCtpu extensions when run in the same program). We expect that CUDA/OpenCL can easily integrate our proposed extensions into their programming interface. The purpose of OpenCtpu simply serves as a transition for developers to easily exploit Edge TPU features and rethink/rewrite algorithms for applications, rather than replacing any existing heterogeneous programming standard.

Figure 3 shows an OpenCtpu code sample. The code contains a kernel function that uses the `conv2D` operator. Before creating a task instance from the kernel, the code must prepare two tensors, `a` and `b`, as inputs and another tensor, `c`, as the output. To describe the dimensionalities of these tensors, the program must call `openctpu_alloc_dimension` to create `openctpu_dimension` data structures for each tensor. The program can then make calls to `openctpu_create_buffer`, which contains the `openctpu_dimensions` values created for `a` and `b`, the pointers to the raw data for `a` and `b`, and the reserved data buffer for the product, `c`. To perform the `conv2D` operation, the program calls the `openctpu_invoke_operator` function, specifying `SCALE` as the quantization method for input/output data, `a` and `b` as the inputs, and `c` as the output for the Edge TPU operator (currently a one-to-one mapping to a fixed set of Edge TPU/CPU instructions). The kernel function returns when the operator is complete.

6 THE GPTPU LIBRARY AND RUNTIME SYSTEM

The GPTPU runtime system receives tasks from the OpenCtpu front end, dynamically schedules tasks, and transforms input/output datasets for tasks. This section describes the design of the GPTPU runtime system.

6.1 Task scheduling

The GPTPU runtime task-scheduling policy is a dataflow-based algorithm on a front-end task operation queue (OPQ) and a back-end instruction queue (IQ) as Figure 4 highlights. An OPQ entry contains a task ID, the requested TPU operation, the input and output locations, and parameters like the quantization method.

GPTPU gradually fills the OPQ during the execution of the user application. When the program calls the `openctpu_enqueue` function, the GPTPU runtime system initiates a new task ID for the invoked kernel function. The runtime system then executes the code designated by the function pointer using the set of parameters from the `openctpu_enqueue` call. The above process ends when `openctpu_invoke_operator` is called to request the involvement of a TPU operator/instruction.

The `openctpu_invoke_operator` function triggers the runtime system to create an OPQ entry with the task ID created from the current kernel function. The GPTPU runtime system then fills the rest of the queue entry with information passed to the `openctpu_invoke_operator` function. As the current OpenCtpu design serializes operators from a single kernel-function instance, kernel-function execution will be blocked until the operation finishes and each task has one operator from the `openctpu_invoke_operator` function in the OPQ. Since OpenCtpu allows all tasks to execute in parallel, the GPTPU runtime system can issue entries in the OPQ to Tensorizer without considering their original order.

After Tensorizer optimizes, reshapes and transforms data and operations into instructions, Tensorizer divides a task into instructions in the IQ. The runtime system then schedules to the same Edge TPU if they share the same input, quantization flags, and the same task ID, but have different output locations—a scheduling

approach that reduces movement overhead and the number of data transformations required. For other instructions, the GPTPU runtime system will use a first-come-first-serve policy to assign them to available Edge TPUs.

6.2 Tensorizer

Tensorizer is responsible for dynamic optimizations at the task level. Tensorizer transforms and optimizes programmer-requested operations into instructions, input tensors and models that enable efficient use of Edge TPUs.

Upon receiving a task from OPQ, Tensorizer first partitions the programmer-requested operation into Edge TPU instructions into sub-problems where each instruction works on its optimal data/model shapes using insights from Section 3.2. Tensorizer transforms the input data to minimize loss of accuracy due to the 8-bit precision of TPU matrix units for each Edge TPU operator.

6.2.1 Mapping operators into instructions. As OpenCtpu hides the hardware details from the programmer, programmer's tasks are agnostic to the granularity of inputs that optimize Edge TPU instructions. Tensorizer tackles this performance issue by dynamically partitioning these tasks into Edge TPU instructions working on their optimal data sizes/shapes (e.g., 128×128 matrices in most arithmetic instructions). As Edge TPU supports limited numbers of instructions/operators, we creates a set of rules that guides Tensorizer in rewriting tasks.

For pair-wise operators that calculate on pairs of values from both input matrices, including `add`, `sub` and `mul` or element-wise operators that calculate on each value of an input matrix, including `tanh` and `relu` the rule is straightforward. Tensorizer simply needs to first divide the input data into tensors and models that contain sub-matrices with the optimal shape. Then, Tensorizer rewrites the operator into a set of Edge TPU instructions where each works on a sub-matrix or a pair of sub-matrices locating at the corresponding positions in the original inputs and collects the results in the corresponding memory locations.

For matrix-wise operators, including `mean` and `max`, Tensorizer still divides the input into sub-matrices with optimal shapes (i.e., both instructions favor 64×64 sub-matrices) and uses instructions to work on each sub-matrix. However, Tensorizer will additionally generate CPU code to aggregate the received values from results of instructions to produce the final outcome. An alternative approach is to create another sets of Edge TPU instructions and making the received values an input tensor/model to iteratively use Edge TPU to produce the result. Tensorizer does not take this approach as (1) the first round of executing `mean` or `max` instruction already shrinks the values to aggregate by a factor of 4096, and (2) the latency of moving data in the currently system architecture is significantly longer than aggregating results with CPU code.

For arithmetic operators, including `FullyConnected` and `conv2D`, Tensorizer applies mechanisms similar to the blocking algorithm for matrix multiplications [69] in rewriting tasks. If each input matrix is partitioned into $P \times Q$ sub-matrices, The resulting code will contain Edge TPU instructions that perform $P \times Q$ `FullyConnected` or `conv2D` instructions and CPU code that aggregates results into the final outcome. The CPU code only needs to add received values that requires very short latency to execute on modern processors. In addition, as CPU registers are wider than Edge TPU's data precision, aggregating results on CPU will allow the platform to reduce precision loss in results.

After rewriting operations into actual machine/accelerator code, Tensorizer will obtain the mapping between an input value and its location in the transformed tensor/model.

6.2.2 Data transformation. To minimize the inaccuracy of computation, Tensorizer carefully rescales values into fixed-point numbers and fill numbers into models or inference data arrays that Edge TPUs can accept. GPTPU determines the scaling factor for input datasets using (1) the sequence of operators, (2) the number of operators, and (3) the range of input data. As the data size of each Edge TPU instruction and the sequence of operators are known at runtime, the GPTPU system can estimate the number of logical arithmetic operations (*num_logical_operations*) that the instructions will generate. By discovering the maximum value (*max*) and the minimum (*min*) value of the dataset, the runtime system can estimate the range of output values and derive the model/tensor scaling factor. The general rule of the scaling factor S of an operator is

$$S = \frac{1}{max(|output_{max}|, |output_{min}|)} \tag{4}$$

where $output_{max}$ is the expected maximum output value and $output_{min}$ is the expected minimum output value. For most datasets, sampling is efficient enough in large datasets as previous work indicates that small subset of input data is representative for the whole dataset [70]. As GPTPU calculates S using the maximum absolute value of outputs, GPTPU prevents the case of overflow.

GPTPU applies different formulas for different types of operators. If the input data is a pair of $N \times N$ matrix, GPTPU estimates the scaling factor for each `conv2D` and `FullyConnected`, as:

$$S = \frac{1}{|max - min|^2 \times N} \tag{5}$$

For pair wise `add` and `sub`, GPTPU uses:

$$S = \frac{1}{2 \times |max - min|} \tag{6}$$

as the scaling factor. For pair wise `mul`, GPTPU uses:

$$S = \frac{1}{|max - min|^2} \tag{7}$$

as the scaling factor, and for other operators, GPTPU calculates the scaling factor as:

$$S = \frac{1}{|max - min|} \tag{8}$$

For example, consider a request that performs matrix multiplication and then pairwisely add another matrix on $N \times N$ matrices with data ranging from 0 to $n - 1$. The maximum output value in the resulting matrix will be $2 \times N \times (n-1)^2$. The runtime system can choose $\frac{1}{2 \times N(n-1)^2}$ as the scaling factor.

6.2.3 The overhead of Tensorizer. Using the information we gained from reverse-engineering the Edge TPU model format as described in Section 3.3, we implemented the proposed Tensorizer to dynamically create models from arbitrary input data. The C-based Tensorizer can bring the latency of generating a model from a 2K×2K matrix down to 1.8 ms—a 1500× speedup over the original

Python-based Edge TPU TFLite compiler and shorter than the latency of data transfer. The GPTPU runtime system thus can overlap Edge TPU matrix-input data movements with Tensorizer to reduce the total latency of executing Edge TPU instructions from tasks.

7 OPTIMIZING APPLICATIONS FOR GPTPU

Mapping a problem into a GPTPU application requires inputs/outputs to be transformed into tensors that Edge TPUs can operate on. Although many applications use data in tensor form, the Edge TPU instructions are optimized for NN workloads, meaning that naively applying the default tensor operators may not improve performance. Tensorizer helps to optimize performance in the task level, but using the most efficient operator for a task still requires programmer's optimization. This section describes GPTPU application design and optimization using matrix multiplication as an example.

7.1 General Matrix Multiply (GEMM)

To demonstrate the importance of designing algorithms to wisely use Edge TPU instructions, we explain the design of an efficient GEMM on Edge TPUs, a fundamental linear-algebra tool for matrices. GEMM takes two 2-dimensional tensors (matrices) as inputs and produces a single 2-dimensional tensor as output. We can calculate each element in the result matrix, C, obtained from a set of pairwise multiplications and accumulations from an $M \times N$ matrix, A, and an $N \times K$ matrix, B.

7.1.1 GEMM and the `FullyConnected` *operator.* The Edge TPU `FullyConnected` instruction offers an intuitive way to implement GPTPU GEMM, as the operator essentially produces a matrix-vector product. A program can select either matrix A or matrix B and iterate through a column or row of the other matrix to produce the result, and matrix multiplication will be performed via the M or K `FullyConnected` operators.

7.1.2 The `conv2D` *operator/instruction.* Edge TPU's `conv2D` instruction can also perform multiplications and accumulations but in different orientations to derive the result. In conventional architectures, programmers implement convolutions by performing scalar-scalar or vector-vector multiplications and accumulations for higher efficiency. However, Table 1 shows that the RPS of convolution (i.e., `conv2D`) is 25× the RPS of matrix-vector multplications (i.e. `FullyConnected`) on Edge TPUs. Inspired by this observation, we therefore explore the implementation by changing the layout of input data and using `conv2D` to perform exactly the same number of multiplications and accumulations on the set of input numbers to leverage the high RPS of `conv2D` for a more efficient GEMM implementation.

The `conv2D` instruction takes one of its inputs as the kernel, multiplies each kernel element with an input element mapping to the corresponding location, and accumulates the result as an output element. Each `conv2D` instruction can produce a result matrix that has the same size as the non-kernel input.

For an $M \times N$ input matrix, A, and an $L \times L$ kernel, B', each element in the `conv2D` $M \times N$ output matrix, C, is:

$$C_{i,j} = \sum_{q=0}^{L} \sum_{p=0}^{L} A_{i+p,j+q} \cdot B'_{p,q} (\forall\ 0 \le i < M, 0 \le j < N) \qquad (9)$$

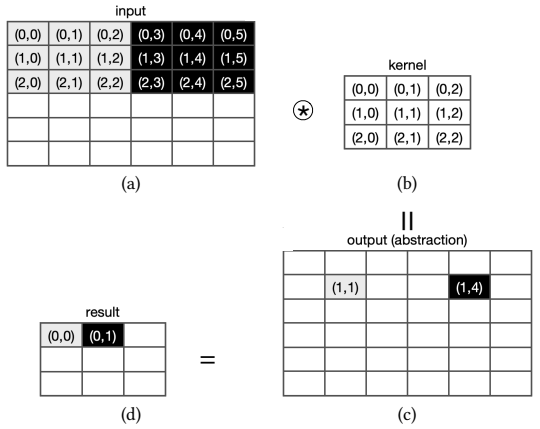

Figure 5: The `conv2D` as implemented with stride

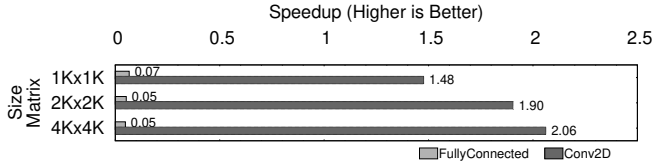

Figure 6: Speedup of GEMM GPTPU implementations using `FullyConnected` and `conv2D`, relative to the baseline CPU OpenBLAS implementations.

Targeting AI/ML workloads that are error tolerant (and so permit approximations), the Edge TPU `conv2D` instruction allows a programmer to assign a *stride* value (s_x, s_y) that treats inputs as groups of $s_x \times s_y$ sub-matrices and produces a corresponding result value for them.

Figure 5 illustrates the concept of `conv2D` with stride. We select (3, 3) as our stride, restricting `conv2D` to 9 numbers in a group; the `conv2D` operator only produces a value for every 3 row/column elements in the abstracted outcome, as in Figure 5(c), from the source matrix, as in Figure 5(a), using the kernel in Figure 5(b). The final output of `conv2D` is a condensed matrix, as in Figure 5(d).

GPTPU uses `conv2D` and its striding feature to implement an efficient GEMM algorithm. The algorithm starts by reshaping both inputs that transform each row in the chosen source matrix into a sub-matrix whose size is determined by the selected stride (s_x, s_y). Ordinarily, both s_x and s_y are the round-up of the square root of the column dimension in the source matrix. The other input matrix serves as a list of kernels, where each kernel of size $s_x \times s_y$ contains a column from that matrix. When creating the kernels, the GPTPU GEMM algorithm fills the kernel elements to match the desired element-wise multiplications for GEMM. In other words, for a matrix with N columns and K rows, the resulting kernel matrix will contain N kernels where each kernel contains $\lceil\sqrt{K}\rceil \times \lceil\sqrt{K}\rceil$ elements. That being said, the resulting kernel matrix still contains exactly the same or similar amount of elements (i.e., $N \times (\lceil\sqrt{K}\rceil)^2$ v.s. $N \times K$) as the original input matrix. After transforming both inputs, `conv2D` iterates through all sub-matrices over each kernel with the selected stride and generates output identical to that of conventional matrix multiplication.

7.1.3 `FullyConnected` *and* `conv2D` *together.* Figure 6 shows the performance of GPTPU GEMM kernel implementations using `FullyConnected` and `conv2D` compared to the CPU baseline using OpenBLAS [71]. The `conv2D` implementation reveals a strong performance gain (a 2.06× speedup in the 4K×4K microbenchmark) over the CPU baseline. In contrast, the GPTPU GEMM implementation cannot beat the CPU baseline without `conv2D` (i.e., when GEMM only uses `FullyConnected`).

Though the GPTPU GEMM algorithm incurs additional data-transformation overhead, GPTPU's `conv2D`-based GEMM significantly outperforms the conventional vector-product-based algorithm by 43× on our GPTPU platform. This is because the Edge TPU architecture highly optimizes `conv2D` and the favorable RPS of `conv2D` compensates for the additional overhead.

Since GEMM is a widely used, fundamental linear-algebra tool for matrices, GPTPU makes the core GEMM algorithm available as an optimized library function, `tpuGemm`, that GPTPU applications can invoke—just as CUDA invokes the cuBLAS GEMM implementation via the `cublasGemm` function [72].

7.2 Other applications

As with GEMM, our goal for all GPTPU applications is to utilize instructions with the highest RPS. We now summarize how we extended the GPTPU GEMM approach to other applications whose workloads we evaluate in the latter part of this paper. This section focuses on the GPTPU instructions that the GPTPU implementations use.

7.2.1 PageRank. The PageRank algorithm [73] is a representative graph application. PageRank takes an adjacency matrix representing a graph as input. Both the baseline and the GPTPU implementations use the classic power method that iteratively performs matrix-vector multiplications. In contrast to CPU/GPU PageRank implementations that perform pairwise or vector-wise multiplications, the GPTPU PageRank implementation simply uses one `FullyConnected` instruction for each adjacency-matrix multiplication with a single vector.

7.2.2 HotSpot3D. HotSpot3D is a thermal-simulation tool for estimating the temperature of a chip made with 3D-stacking. The main algorithm gradually and iteratively updates each point on the chip, which is represented as a matrix with a weighted average of the point's closest neighbors in 8 different directions. The HotSpot3D algorithm can naturally map to `conv2d` with a 3×3 kernel without striding.

7.2.3 LU Decomposition (LUD). LUD factors a matrix into a lower triangular matrix (L) and an upper triangular matrix (U) such that $L \times U$ yields the original matrix. Our GPTPU LUD implementation uses the recursive algorithm [74] via `crop`, `FullyConnected`, and `conv2D` to partition matrices and perform appropriate operations on different combinations of the partitioned matrices.

7.2.4 Gaussian elimination (Gaussian). Like LUD, Gaussian is a method for solving a system of linear equations. Gaussian combines row swaps, the scalar multiplication of rows, and row additions until the lower left-hand triangular matrix contains only zeroes. For Gaussian, GPTPU uses `mul` to perform each row reduction.

Benchmark	Input Matrices	Data Size	Category	Baseline Implementation
Backprop	1×8K×8K	512MB	Pattern Recognition	[76, 77]
BlackScholes	1×256M×9	9GB	Finance	[78]
Gaussian	1×4K×4K	64MB	Linear Algebra	[76, 77]
GEMM	2×16K×16K	1GB	Linear Algebra	[71, 72, 79]
HotSpot3D	8×8K×8K	2GB	Physics Simulation	[76, 77]
LUD	1×4K×4K	64MB	Linear Algebra	[76, 77]
PageRank	1×32K×32K	4GB	Graph	[80]

Table 3: The input dataset sizes for the GPTPU benchmark applications

7.2.5 Backpropagation (Backprop). Backprop is foundational to NN supervised learning. We implemented a plain-vanilla version of Backprop to demonstrate the ML/AI-generalizable nature of GPTPU. For a feedforward NN, the GPTPU Backprop uses (1) multiple layers of `FullyConnected` and sigmoid activation functions in `ReLu`, (2) `add` for the actual backpropagation, and (3) `tpuGEMM` to derive weights for the delta matrix.

7.2.6 Black–Scholes (BlackScholes). BlackScholes is a financial model for estimating the market price of stock options. GPTPU uses a ninth-degree polynomial function [75] with the `FullyConnected` instruction to compute the cumulative normal distribution function.

8 EXPERIMENTAL METHODOLOGY

8.1 The system platform

We use exactly the same prototype machine described in Section 3 for all experiments performed with GPTPU. When performing experiments for baseline applications, we removed the TPUs from the machine.

For each application, we measured the end-to-end latency. We also measure the total system power using a Watts Up meter. When calculating energy consumption, we aggregate the total system power throughout the application execution time. On average, each active Edge TPU adds only 0.9 W to 1.4 W of power consumption, while a loaded AMD Matisse core in the GPTPU hardware prototype consumes from 6.5 W to 12.5 W. As GPTPU still relies on the CPU for the runtime system and data transformation, both CPUs and Edge TPUs can be active when running applications. The idle power of the experimental platform is 40 W, including the southbridge chip on the motherboard, NVMe-based storage devices as well as other peripherals connected to the system.

8.2 The baseline application implementations

For each application described in Sections 7, we compared our GPTPU implementations with (1) optimized CPU/GPU implementations from benchmark suites [76, 78] or (2) widely-used distributions [71, 72, 80]. Table 3 lists the input datasets and the baseline implementations for each application we used in our experiments. We only select a subset of applictions from these benchmark suites because these are all applications that (a) preserve the form of matrix inputs and (b) can map their core algorithms to reasonable matrix operations. We do not expect GPTPU and Edge TPUs to be effective for applications that can only exploit vector arithmetics since Edge TPU's architecture is specialized for matrix operations.

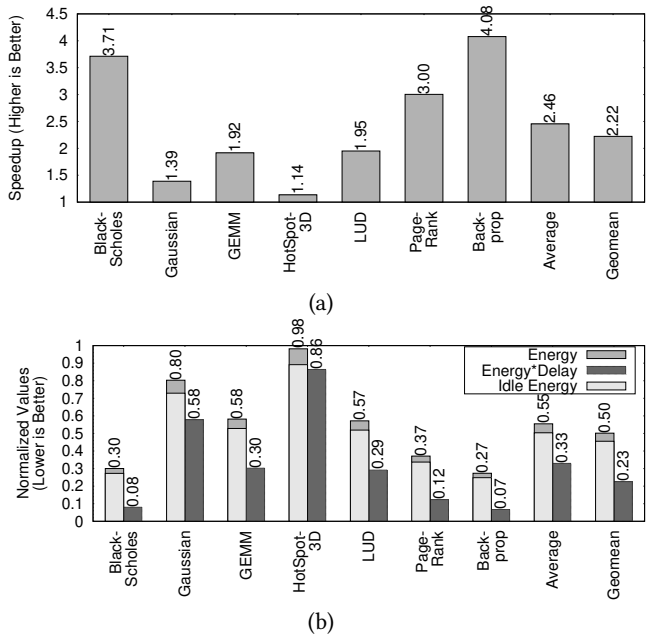

Figure 7: The application (a) speedup, (b) energy consumption, and energy-delay products for a single Edge TPU, relative to the baseline CPU implementations

We also use Facebook's GEMM (FBGEMM) [79] for approximate computing on GEMM.

9 RESULTS

This section describes the speedup, energy consumption, and accuracy observed for GPTPU when running different applications. Compared to modern CPU-based platforms running optimized code, GPTPU exhibits improved performance and significantly reduced energy needs. In addition, the GPTPU GEMM implementation yields more reliable results in approximation than a low-precision matrix-multiplication library run on a CPU.

9.1 Single core performance: GPTPU vs. CPU

Figure 7 summarizes the speedup, energy consumption, and energy-delay of GPTPU-based applications. We used a single Edge TPU and a single CPU core to compare execution of workloads in our baseline tests to compare the per-core raw hardware capabilities.

Figure 7(a) compares end-to-end latency. The GPTPU system is, on average, 2.46× faster than the CPU. For Backprop, the speedup is 4.08× (not surprising given that the Edge TPU was originally designed for applications like Backprop). Excluding Backprop, the average speedup is still 2.19×. HotSpot3D actually experiences the least speedup with GPTPU. This is because GPTPU's HotSpot3D uses very small kernels and large inputs accompany each iteration, the data-movement overhead dominates end-to-end application latency. However, even under this scenario, GPTPU can still speed up the performance of HotSpot3D by 1.14×.

Figure 7(b) shows the relative energy consumption and energy-delay products for GPTPU applications vs. their CPU baseline implementations. GPTPU consumes only 5% of the active energy and only 51% of the idle energy that a CPU consumes (an energy savings of 45%), and even the worst-performing GPTPU benchmark still saves 3% overall system energy. For energy-delay products, which take both latency and energy consumption into consideration, applications run with GPTPU enjoy a 67% reduction over the baseline CPU. Excluding the top-performing Backprop, GPTPU still achieves an 40% energy savings and a 62% energy-delay improvement.

Benchmark	default	$-2^7 \leq x < 2^7$	$-2^{15} \leq x < 2^{15}$	$-2^{31} \leq x < 2^{31}$
Backprop	0.12%	0.17%	0.10%	0.11%
Blackscholes	0.18%	0.18%	0.18%	0.18%
Gaussian	0.00%	0.00%	0.00%	0.00%
GEMM	0.89%	0.90%	0.90%	0.90%
HotSpot	0.50%	0.49%	0.46%	0.46%
LUD	0.00%	0.00%	0.00%	0.00%
PageRank	0.61%	0.73%	0.73%	0.73%
Average	0.33%	0.35%	0.34%	0.34%

(a)

Benchmark	default	$-2^7 \leq x < 2^7$	$-2^{15} \leq x < 2^{15}$	$-2^{31} \leq x < 2^{31}$
Backprop	0.14%	0.17%	0.12%	0.12%
Blackscholes	0.33%	0.33%	0.33%	0.33%
Gaussian	0.00%	0.00%	0.00%	0.00%
GEMM	0.98%	0.91%	0.91%	0.91%
HotSpot	0.64%	0.64%	0.59%	0.59%
LUD	0.00%	0.00%	0.00%	0.00%
PageRank	0.41%	0.91%	0.91%	0.91%
Average	0.41%	0.42%	0.41%	0.41%

(b)

Table 4: The (a) MAPEs and (b) RMSEs for GPTPU applications

GPTPU sacrifices accuracy—but only to a limited degree. Table 4 measured the mean absolute percentage error (MAPE) and the root mean square error (RMSE) between the GPTPU and CPU application implementations using the default dataset from the benchmark and our randomly generated datasets with various ranges of values in their inputs. The MAPE is always less than 1% across all applications, regardless their ranges of input values. The average MAPE is 0.26%–0.33%. The largest RMSE we measured was an acceptable 0.98%. In some cases, the GPTPU results in higher error rates in compute on default datasets than on synthetic inputs with larger data ranges. This is because the input values of synthetic datasets are typically normally distributed but the real, default datasets are not always normally distributed.

9.2 GPTPU-GEMM vs. 8-bit CPU GEMM

GPTPU allows single–Edge TPU performance to surpass single-CPU-core performance. That being said, the Edge TPU uses low-precision data types, whereas the baseline CPU implementations do not. To account for this difference when using approximate computing with the CPU cores, we compared the GPTPU implementation running with the state-of-the-art FBGEMM low-precision CPU matrix-multiplication library that intensively uses the latest AVX instructions to support 8-bit operations [79]. We did not include other workloads in this part as other workloads do not have implementations optimized for 8-bit CPU operations.

Table 5 shows the results for GPTPU's GEMM vs. FBGEMM using 1024×1024 matrices with positive integers and maximum input values ranging from 2 to 128 (we chose this data size only to accommodate FBGEMM's limitations). As Figure **??**(a) shows, GPTPU-GEMM consistently outperforms FBGEMM on high-performance CPU cores with 1.22× to 1.28× across all configurations. However,

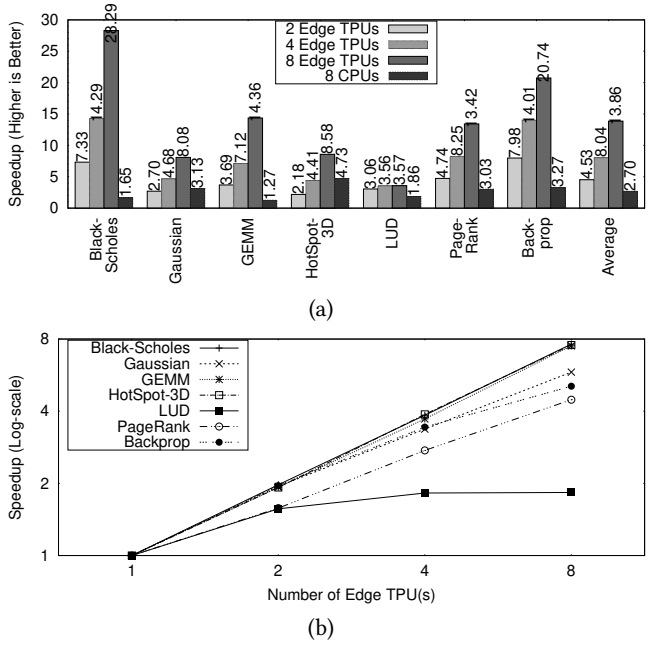

Figure 8: Performance scaling for multiple Edge TPUs

Range of Values		0–2	0–4	0–8	0–16	0–32	0–64	0–128
Speedup over FBGEMM		1.26	1.27	1.28	1.22	1.28	1.27	1.28
RMSE	FBGEMM	0.00	0.00	0.00	0.00	0.47	0.87	0.97
	TPUGEMM	0.00	0.00	0.00	0.00	0.00	0.00	0.01

Table 5: The speedup and RMSE for GPTPU's GEMM library function relative to FBGEMM

when the maximum matrix-entry value exceeds 16, FBGEMM's RMSE is poor as Table 5 presents, reaching 47% when the largest value within the dataset is 32. Furthermore, the FBGEMM RMSE goes as high as 97%, meaning that most result values are not convincing when the largest value is 128. In contrast, GPTPU-GEMM's RMSE is always less than 1% (0.82% when maximum value is 128). This is because FB's GEMM targets at error-tolerant ML applications but does not handle overflow cases. However, the performance evaluation indicates that even if the CPU baseline uses 8-bit operations, GPTPU-GEMM is faster.

9.3 Parallel processing with multiple Edge TPUs

The GPTPU runtime system uses a task queue that allows multiple Edge TPUs to process tasks in parallel. Even without programmer's explicit partitioning of tasks, Tensorizer also automatically generates parallel tasks from the user code. Figure 8(a) shows the speedup of adding more Edge TPUs into our system, without modifying the user code, compared with the single-core CPU baseline. With 8 Edge TPUs that consume similar active power as a single RyZen core, GPTPU achieves an average 13.86× speedup. In constrast, the 8-core, OpenMP-based CPU implementations can only achieve 2.70× speedup over the baseline. Figure 8(b) further shows log-scale performance with up to 8 Edge TPUs running GPTPU tasks, compared with single Edge TPU. The linear plots reveal good performance scaling for 6 out of 7 applications when the GPTPU runtime system executes tasks in parallel. The only exception is LUD, which already

	Cost	Power Con.	Comment
Single Edge TPU	USD 24.99	2 W	
RTX 2080	USD 699.66	215 W	Now USD 1399
Jetson Nano	USD 123.99	10 W	
8× Edge TPU	USD 159.96	16 W	Using 4× dual Edge TPU modules

Table 6: The cost and power consumption of hardware accelerators that we compared in this work

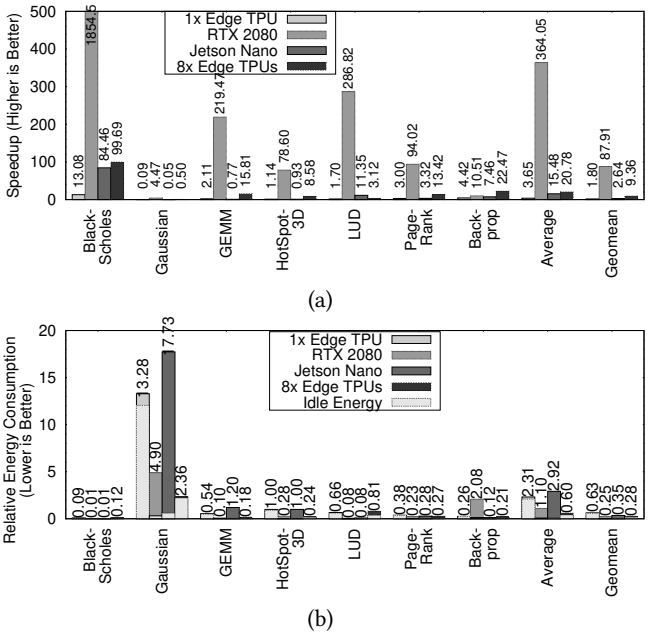

Figure 9: The relative (a) performance and (b) energy for GPTPU with 1× and 8× Edge TPUs vs. the GTX 2080 GPU and Jetson Nano

partitions matrices into four sub-matrices for computation using matrix-wise operators, making it difficult for Tensorizer to scale the performance in only one of the four partitions.

9.4 Comparison with GPUs

Because an increasing number of workloads leverage GPU parallelism, we compared the GPTPU to NVIDIA's high-end Turing-architecture-based GTX 2080 and NVIDIA's embedded Jetson Nano platform. Table 6 lists the cost and power consumption of evaluated GPUs along with Edge TPUs. Due to the limitation of Jetson Nano's available memory capacity, we have to scale down the input datasets of Blackscholes, Gaussian, GEMM, LUD and PageRank by 25% to 50% to not crash the GPU kernel. Figure 9(a) compares the performance for the RTX 2080 and Jetson Nano, using a single Ryzen 3700X CPU core as the baseline, for Rodinia benchmark applications and GEMM using cuBLAS. We enabled RTX-2080's 16-bit ALUs for Gaussian, HotSpot3D, Backprop and Tensor Cores in 8-bit mode for GEMM. The GTX 2080 GPU is 364× faster than a CPU core and 69× faster than the Edge TPU. The embedded GPU on Jetson Nano is still 15× faster than a CPU core and 2.30× faster than an Edge TPU on average. However, with 8× Edge TPUs, the GPTPU can outperform the CPU core by 3.65× and Jetson Nano by 2.48×.

Figure 9(b) compares the energy consumption of evaluated platforms. Including idle energy, the 8×-Edge TPU system is the most energy-efficient as the platform can save energy by 40% from the CPU baseline but achieve reasonable speedup. In constrast, the GTX 2080 platform consumes 9% more energy than the CPU baseline. Even though the idle power of the Jetson nano development kit is simply 0.5 W, Jetson nano is still more energy-consuming than GTX 2080 due to the limited speedup.

If we only consider the active power consumption to exclude the factor of various idle power in different system settings, the GTX 2080 consumes 14× the energy of 1× Edge TPU on average, due to the GPU's 195× average active power consumption compared with the Edge TPU, translating to 4.96× worse energy-delay than the baseline. Jetson Nano consumes 23.55× more energy than 1× Edge TPU, making the energy-delay of nano 15.54× worse than 1× Edge TPU. 8×-Edge TPU system consumes just 75% more active energy than 1× Edge TPU, even though the active power consumption is almost 8× of a single Edge TPU. With 8× Edge TPUs, GPTPU offers even better energy-delay (i.e., 46% lower) than the baseline. This result shows that GPTPU offers better energy-efficiency than the current GPU-based solution on embedded/edge platforms.

10 RELATED WORK

Neural processing units (NPUs) [81, 82] work by using pre-trained models that predicts the outcome of code blocks and map the user program to these models. The GPTPU-based approach is fundamentally different from approaches that rely on the acceleration of approximate programs via NPU in three important ways: (1) GPTPU can accelerate any user-defined algorithm by mapping tensor/matrix operations to supported operators, whereas NPUs can only accelerate a limited set of algorithms that match previously trained NN models. (2) GPTPU can leverage the Edge TPU microarchitecture and NN hardware to implement exact tensor/matrix operations for applications, whereas NPUs use NNs to produce approximate results for applications. (3) GPTPU can achieve the desired level of precision by iteratively computing on different portions of raw input numbers, whereas NPUs are always limited by the approximate outcomes of NN models.

ASICs can be used like TPUs to accelerate NN applications, as can existing fine-tuned architecture components. Industry data centers [83–85] take advantage of heterogeneous hardware components by using different processors and reconfigurable hardware for different ML tasks. EFLOPS [86], Richins et. al. [87], and FlexTensor [88] optimize algorithms and task allocations for network traffic in data-center-scale edge computing or single-server computing to reduce infrastructure costs. Language frameworks like ApproxHPVM [89] and ApproxTuner [90] further helps programmer to estimate and optimize the loss of accruacy in ML workloads. The GPTPU framework is orthogonal to the aforementioned research because GPTPU is compatible with existing heterogeneous computing platforms; Edge TPUs can function as complementary hardware accelerators within the system. Ultimately, emerging tensor-processing hardware will inspire the development of related algorithms and associated software [91–93]. We have seen work extending the application of TPUs to medical image processing [94]. We expect GPTPU can further facilitate this trend. GPTPU can exist in parallel to such future research and potentially extend newly developed algorithms to work in additional application domains.

This paper does not focus on sparse matrices, as many NN accelerators implicitly optimize for sparse matrices. Examples include SCNN [95], SparTen [96], Sparch [97], Scalpel [98], SIGMA [99], Cambricon-X [100], Bit-Tactical [101], Bit-Pragmatic [102], OuterSPACE [103], Laconic [104], Bit Fusion [105], Sparse Tensor Core [106], PermDNN [107], Park et al. [108], Song et al. [109], and Rhu et al. [110].

11 CONCLUSION

This paper presents GPTPU to bridge the gap between NN accelerators and general-purpose programming. By reverse engineering the commercially available, low-profile NN accelerator, the Google Edge TPU, to uncover important architectural characteristics and the data-exchange protocol, we implement an efficient runtime system, including Tensorizer that dynamically optimizes data layout and instructions, as the GPTPU platform's backend. Using the GPTPU platform and the derived performance numbers, we re-designed the algorithms for a set of important, non-AI/ML related applications. The prototype GPTPU system exhibits a 2.46× speedup over modern high-end CPUs with 40% energy reduction. Though single Edge TPU performance is not yet competitive with high-end GPUs, but the strong scalability of multiple Edge TPUs reveals the potential of future extensions of this line of accelerators. As the demand of ML applications keep growing, we expect manufacturers to keep advancing the microarchitecture of ML accelerators for higher performance and energy-efficiency. GPTPU thus represents an important exploration of general-purpose computing on NN accelerators and is complementary to existing work. The insights presented in this paper will also help extend the range of NN accelerator applications as well as guiding the algorithm design and code optimization for future NN accelerators.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their helpful comments. This work was sponsored by an National Science Foundation (NSF) award, 2007124. This work was also supported by new faculty start-up funds from University of California, Riverside. We also owe a debt of gratitude to Christopher Fraser for his excellent copyediting skills.

REFERENCES

[1] Google LLC, "Coral M.2 accelerator datasheet." https://coral.withgoogle.com/static/files/Coral-M2-datasheet.pdf, 2019.
[2] Apple, "Small chip. Giant leap.." https://www.apple.com/mac/m1/.
[3] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pp. 73–82, 2008.
[4] V. Volkov and J. W. Demmel, "Benchmarking gpus to tune dense linear algebra," in *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pp. 1–11, IEEE, 2008.
[5] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel computing experiences with cuda," *IEEE micro*, vol. 28, no. 4, pp. 13–27, 2008.
[6] S. Lee, S.-J. Min, and R. Eigenmann, "Openmp to gpgpu: a compiler framework for automatic translation and optimization," *ACM Sigplan Notices*, vol. 44, no. 4, pp. 101–110, 2009.

[7] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving gpu performance via large warps and two-level warp scheduling," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 308–317, 2011.

[8] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A gpgpu compiler for memory optimization and parallelism management," *ACM Sigplan Notices*, vol. 45, no. 6, pp. 86–97, 2010.

[9] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-m. W. Hwu, "Program optimization space pruning for a multithreaded gpu," in *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pp. 195–204, 2008.

[10] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "A compiler framework for optimization of affine loop nests for gpgpus," in *Proceedings of the 22nd annual international conference on Supercomputing*, pp. 225–234, 2008.

[11] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen, "On-the-fly elimination of dynamic irregularities for gpu computing," *ACM SIGPLAN Notices*, vol. 46, no. 3, pp. 369–380, 2011.

[12] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August, "Automatic cpu-gpu communication management and optimization," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pp. 142–151, 2011.

[13] NVIDIA Corporation, "CUDA C programming guide v6.0." http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, 2014.

[14] Khronos Group, "OpenCL." http://www.khronos.org/opencl/.

[15] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "HyGCN: A GCN accelerator with hybrid architecture," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 15–29, 2020.

[16] C. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, IEEE Press.

[17] S. Wang, D. Zhou, X. Han, and T. Yoshimura, "Chain-NN: An energy-efficient 1D chain architecture for accelerating deep convolutional neural networks," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, DATE, '17, pp. 1032–1037, IEEE Press, 2017.

[18] N. Srivastava, H. Jin, S. Smith, H. Rong, D. Albonesi, and Z. Zhang, "Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '20, pp. 689–702, IEEE Press, 2020.

[19] Y. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 367–379, 2016.

[20] M. Gao, X. Yang, J. Pu, M. Horowitz, and C. Kozyrakis, "Tangram: Optimized coarse-grained dataflow for scalable nn accelerators," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, (New York, NY, USA), p. 807–820, Association for Computing Machinery, 2019.

[21] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin, "SNNAP: Approximate computing on programmable SoCs via neural acceleration," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 603–614, 2015.

[22] L. Song, F. Chen, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "AccPar: Tensor partitioning for heterogeneous deep learning accelerators," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 342–355, 2020.

[23] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, DAC, '17, (New York, NY, USA), IEEE Press, 2017.

[24] H. Kung, B. McDanel, and S. Q. Zhang, "Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, (New York, NY, USA), p. 821–834, Association for Computing Machinery, 2019.

[25] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *SIGARCH Comput. Archit. News*, vol. 42, pp. 269–284, Feb. 2014.

[26] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "DaDianNao: A machine-learning supercomputer," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 609–622, 2014.

[27] H. Kwon, A. Samajdar, and T. Krishna, "MAERI: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, (New York, NY, USA), p. 461–475, Association for Computing Machinery, 2018.

[28] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen, "Cambricon: An instruction set architecture for neural networks," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 393–405, 2016.

[29] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "FlexFlow: A flexible dataflow accelerator architecture for convolutional neural networks," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 553–564, 2017.

[30] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey, and A. Raghunathan, "ScaleDeep: A scalable compute architecture for learning and evaluating deep networks," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 13–26, 2017.

[31] H. Jang, J. Kim, J. Jo, J. Lee, and J. Kim, "MnnFast: A fast and scalable system architecture for memory-augmented neural networks," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pp. 250–263, 2019.

[32] C. Deng, F. Sun, X. Qian, J. Lin, Z. Wang, and B. Yuan, "TIE: Energy-efficient tensor train-based inference engine for deep neural network," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pp. 264–277, 2019.

[33] K. Hegde, J. Yu, R. Agrawal, M. Yan, M. Pellauer, and C. Fletcher, "UCNN: Exploiting computational reuse in deep neural networks via weight repetition," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 674–687, 2018.

[34] C. Ding, S. Liao, Y. Wang, Z. Li, N. Liu, Y. Zhuo, C. Wang, X. Qian, Y. Bai, G. Yuan, X. Ma, Y. Zhang, J. Tang, Q. Qiu, X. Lin, and B. Yuan, "CirCNN: Accelerating and compressing deep neural networks using block-circulant weight matrices," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, (New York, NY, USA), p. 395–408, Association for Computing Machinery, 2017.

[35] L. Song, J. Mao, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "HyPar: Towards hybrid parallelism for deep learning accelerator array," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 56–68, 2019.

[36] J. Park, H. Sharma, D. Mahajan, J. K. Kim, P. Olds, and H. Esmaeilzadeh, "Scale-out acceleration for machine learning," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 367–381, 2017.

[37] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to FPGAs," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, 2016.

[38] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, 2016.

[39] M. Song, J. Zhang, H. Chen, and T. Li, "Towards efficient microarchitectural design for accelerating unsupervised GAN-based deep learning," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 66–77, IEEE, 2018.

[40] A. Azizimazreah and L. Chen, "Shortcut mining: Exploiting cross-layer shortcut reuse in DCNN accelerators," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 94–105, 2019.

[41] S. Hurkat and J. F. Martínez, "VIP: A versatile inference processor," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 345–358, 2019.

[42] Y. S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, S. G. Tell, Y. Zhang, W. J. Dally, J. Emer, C. T. Gray, B. Khailany, and S. W. Keckler, "Simba: Scaling deep-learning inference with multi-chip-module-based architecture," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, (New York, NY, USA), p. 14–27, Association for Computing Machinery, 2019.

[43] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaauw, and R. Das, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA'18, pp. 383–396, IEEE Press, 2018.

[44] Y. Kwon, Y. Lee, and M. Rhu, "TensorDIMM: A practical near-memory processing architecture for embeddings and tensor operations in deep learning," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, (New York, NY, USA), p. 740–753, Association for Computing Machinery, 2019.

[45] J. R. Stevens, A. Ranjan, D. Das, B. Kaul, and A. Raghunathan, "Manna: An accelerator for memory-augmented neural networks," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, (New York, NY, USA), p. 794–806, Association for Computing Machinery, 2019.

[46] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "DRISA: A dram-based reconfigurable in-situ accelerator," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 288–301, 2017.

[47] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "TETRIS: Scalable and efficient neural network acceleration with 3D memory," *SIGPLAN Not.*, vol. 52,

p. 751–764, Apr. 2017.
[48] H. Kim, J. Sim, Y. Choi, and L. Kim, "NAND-Net: Minimizing computational complexity of in-memory processing for binary neural networks," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 661–673, 2019.
[49] S. Li, A. O. Glova, X. Hu, P. Gu, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "SCOPE: A stochastic computing engine for DRAM-based in-situ accelerator," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 696–709, 2018.
[50] X. Wang, J. Yu, C. Augustine, R. Iyer, and R. Das, "Bit prudent in-cache acceleration of deep convolutional neural networks," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 81–93, 2019.
[51] J. Liu, H. Zhao, M. A. Ogleari, D. Li, and J. Zhao, "Processing-in-memory for energy-efficient neural network training: A heterogeneous approach," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 655–668, 2018.
[52] M. Imani, M. Samragh Razlighi, Y. Kim, S. Gupta, F. Koushanfar, and T. Rosing, "Deep learning acceleration with neuron-to-memory transformation," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 1–14, 2020.
[53] Y. Ji, Y. Zhang, X. Xie, S. Li, P. Wang, X. Hu, Y. Zhang, and Y. Xie, "FPSA: A full system stack solution for reconfigurable ReRAM-based NN accelerator architecture," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'19, (New York, NY, USA), pp. 733–747, Association for Computing Machinery, 2019.
[54] H. Mao, M. Song, T. Li, Y. Dai, and J. Shu, "LerGAN: a zero-free, low data movement and PIM-based GAN architecture," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 669–681, 2018.
[55] T. Yang, H. Cheng, C. Yang, I. Tseng, H. Hu, H. Chang, and H. Li, "Sparse ReRAM engine: joint exploration of activation and weight sparsity in compressed neural networks," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pp. 236–249, 2019.
[56] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "PRIME: a novel processing-in-memory architecture for neural network computation in ReRAM-based main memory," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 27–39, 2016.
[57] L. Song, X. Qian, H. Li, and Y. Chen, "PipeLayer: A pipelined ReRAM-based accelerator for deep learning," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 541–552, 2017.
[58] A. Ankit, I. E. Hajj, S. R. Chalamalasetti, G. Ndu, M. Foltin, R. S. Williams, P. Faraboschi, W.-m. W. Hwu, J. P. Strachan, K. Roy, and D. S. Milojicic, "PUMA: A programmable ultra-efficient memristor-based accelerator for machine learning inference," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, (New York, NY, USA), pp. 715—731, Association for Computing Machinery, 2019.
[59] M. N. Bojnordi and E. Ipek, "Memristive boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 1–13, 2016.
[60] X. Zhang, S. L. Song, C. Xie, J. Wang, W. Zhang, and X. Fu, "Enabling highly efficient capsule networks processing through a PIM-based architecture design," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 542–555, 2020.
[61] M. Imani, S. Gupta, Y. Kim, and T. Rosing, "FloatPIM: In-memory acceleration of deep neural network training with high precision," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pp. 802–815, 2019.
[62] Khadas, Shenzhen Wesion Technology Co., Ltd., "VIM3." https://www.khadas.com/vim3l, 2019.
[63] Fuzhou Rockchip Electronics Co., Ltd., "Rockchip RK1808." https://www.rockchips.com/a/en/products/RK18_Series/2019/0529/989.html, 2019.
[64] Sophon Technology (Beijing) Co., Ltd. , "Tensor Computing Processor BM1880." https://www.sophon.ai/product/introduce/bm1880.html, 2018.
[65] Shenzhen LeMaker Technology Co., Ltd, "HiKey 970." http://www.lemaker.org/product-hikey970-specification.html, 2018.
[66] NVIDIA Corporation, "Jetson Nano Developer Kit." https://developer.nvidia.com/embedded/jetson-nano-developer-kit, 2019.
[67] QNAP, "QM2 Expansion Card (Add M.2 SSD Slots)." https://www.qnap.com/en/product/qm2-m.2ssd, 2020.
[68] V. Sze, Y. H. Chen, T. J. Yang, and J. S. Emer, "How to evaluate deep neural network processors: Tops/w (alone) considered harmful," *IEEE Solid-State Circuits Magazine*, vol. 12, no. 3, pp. 28–41, 2020.
[69] J. J. Dongarra and D. C. Sorensen, "Linear algebra on high performance computers," *Applied mathematics and computation*, vol. 20, no. 1-2, pp. 57–88, 1986.
[70] M. A. Laurenzano, P. Hill, M. Samadi, S. Mahlke, J. Mars, and L. Tang, "Input responsiveness: Using canary inputs to dynamically steer approximation," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, (New York, NY, USA), pp. 161–176, ACM, 2016.
[71] Zhang Xianyi and Martin Kroeker, "OpenBLAS: An optimized BLAS library." https://www.openblas.net/, 2021.
[72] NVIDIA, "cuBLAS." https://docs.nvidia.com/cuda/cublas/index.html, 2019.
[73] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web.," Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.
[74] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd ed., 2001.
[75] K. Aludaat and M. Alodat, "A note on approximating the normal distribution function," *Applied Mathematical Sciences (Ruse)*, 01 2008.
[76] M. B. S. Che, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the IEEE International Symposium on Workload Characterization*, IISWC '09, pp. 44–54, Oct 2009.
[77] N.-M. Ho and W.-F. Wong, "Exploiting half precision arithmetic in nvidia gpus," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, 2017.
[78] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran, "AxBench: A Multiplatform Benchmark Suite for Approximate Computing," *IEEE Design Test*, vol. 34, pp. 60–68, April 2017.
[79] Daya S Khudia and Protonu Basu and Summer Deng, "Open-sourcing FBGEMM for state-of-the-art server-side inference." https://engineering.fb.com/ml-applications/fbgemm/, 2018.
[80] Carl Yang and Aydin Buluc and Yangzihao Wang and John D. Owens, "GraphBLAST." https://github.com/gunrock/graphblast, 2019.
[81] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 449–460, 2012.
[82] A. Yazdanbakhsh, J. Park, H. Sharma, P. Lotfi-Kamran, and H. Esmaeilzadeh, "Neural acceleration for GPU throughput processors," in *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, (New York, NY, USA), pp. 482–493, ACM, 2015.
[83] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–13, IEEE, 2016.
[84] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang, "Applied machine learning at Facebook: A datacenter infrastructure perspective," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 620–629, 2018.
[85] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pp. 13–24, June 2014.
[86] J. Dong, Z. Cao, T. Zhang, J. Ye, S. Wang, F. Feng, L. Zhao, X. Liu, L. Song, L. Peng, Y. Guo, X. Jiang, L. Tang, Y. Du, Y. Zhang, P. Pan, and Y. Xie, "EFLOPS: Algorithm and system co-design for a high performance distributed training platform," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 610–622, 2020.
[87] D. Richins, D. Doshi, M. Blackmore, A. Thulaseedharan Nair, N. Pathapati, A. Patel, B. Daguman, D. Dobrijalowski, R. Illikkal, K. Long, D. Zimmerman, and V. Janapa Reddi, "Missing the forest for the trees: End-to-end AI application performance in edge data centers," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 515–528, 2020.
[88] S. Zheng, Y. Liang, S. Wang, R. Chen, and K. Sheng, "Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, (New York, NY, USA), pp. 859–873, Association for Computing Machinery, 2020.
[89] H. Sharif, P. Srivastava, M. Huzaifa, M. Kotsifakou, K. Joshi, Y. Sarita, N. Zhao, V. S. Adve, S. Misailovic, and S. Adve, "Approxhpvm: A portable compiler ir for accuracy-aware optimizations," vol. 3, no. OOPSLA, 2019.
[90] H. Sharif, Y. Zhao, M. Kotsifakou, A. Kothari, B. Schreiber, E. Wang, Y. Sarita, N. Zhao, K. Joshi, V. S. Adve, S. Misailovic, and S. Adve, "Approximating APSP without Scaling: Equivalence of Approximate Min-plus and Exact Min-Max," in *Symposium on Principles and Practice of Parallel Programming*, PPoPP 2021, 2021.
[91] S. Chou, F. Kjolstad, and S. Amarasinghe, "Format abstraction for sparse tensor algebra compilers," *Proc. ACM Program. Lang.*, vol. 2, pp. 123:1–123:30, Oct. 2018.
[92] P. Holanda and H. Mühleisen, "Relational queries with a tensor processing unit," in *Proceedings of the 15th International Workshop on Data Management on*

New Hardware, DaMoN'19, (New York, NY, USA), Association for Computing Machinery, 2019.

[93] A. Dakkak, C. Li, J. Xiong, I. Gelado, and W.-m. Hwu, "Accelerating reduction and scan using tensor core units," in *Proceedings of the ACM International Conference on Supercomputing*, ICS '19, (New York, NY, USA), p. 46?V57, Association for Computing Machinery, 2019.

[94] C. Ma, T. Marin, T. Lu, Y. fan Chen, and Y. Zhuo, "Accelerating mri reconstruction on tpus," 2020.

[95] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, (New York, NY, USA), p. 27–40, Association for Computing Machinery, 2017.

[96] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. N. Vijaykumar, "SparTen: A sparse tensor accelerator for convolutional neural networks," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '19, Association for Computing Machinery, 2019.

[97] Z. Zhang, H. Wang, S. Han, and W. J. Dally, "SpArch: Efficient architecture for sparse matrix multiplication," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 261–274, IEEE, 2020.

[98] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing DNN pruning to the underlying hardware parallelism," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 548–560, 2017.

[99] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "SIGMA: A sparse and irregular GEMM accelerator with flexible interconnects for DNN training," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 58–70, 2020.

[100] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-X: an accelerator for sparse neural networks," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, 2016.

[101] A. Delmas Lascorz, P. Judd, D. M. Stuart, Z. Poulos, M. Mahmoud, S. Sharify, M. Nikolic, K. Siu, and A. Moshovos, "Bit-Tactical: A software/hardware approach to exploiting value and bit sparsity in neural networks," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, (New York, NY, USA), p. 749–763, Association for Computing Machinery, 2019.

[102] J. Albericio, A. Delmás, P. Judd, S. Sharify, G. O'Leary, R. Genov, and A. Moshovos, "Bit-pragmatic deep neural network computing," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, (New York, NY, USA), p. 382–394, Association for Computing Machinery, 2017.

[103] S. Pal, J. Beaumont, D. Park, A. Amarnath, S. Feng, C. Chakrabarti, H. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, "OuterSPACE: An outer product based sparse matrix multiplication accelerator," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 724–736, 2018.

[104] S. Sharify, A. D. Lascorz, M. Mahmoud, M. Nikolic, K. Siu, D. M. Stuart, Z. Poulos, and A. Moshovos, "Laconic deep learning inference acceleration," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pp. 304–317, 2019.

[105] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh, "Bit Fusion: Bit-level dynamically composable architecture for accelerating deep neural networks," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 764–775, 2018.

[106] M. Zhu, T. Zhang, Z. Gu, and Y. Xie, "Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern GPUs," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, (New York, NY, USA), pp. 359–371, Association for Computing Machinery, 2019.

[107] C. Deng, S. Liao, Y. Xie, K. K. Parhi, X. Qian, and B. Yuan, "PermDNN: Efficient compressed DNN architecture with permuted diagonal matrices," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 189–202, 2018.

[108] E. Park, D. Kim, and S. Yoo, "Energy-efficient neural network accelerator based on outlier-aware low-precision computation," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 688–698, 2018.

[109] M. Song, J. Zhao, Y. Hu, J. Zhang, and T. Li, "Prediction based execution on deep neural networks," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 752–763, 2018.

[110] M. Rhu, M. O'Connor, N. Chatterjee, J. Pool, Y. Kwon, and S. W. Keckler, "Compressing DMA engine: Leveraging activation sparsity for training deep neural networks," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 78–91, IEEE, 2018.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We ran the Rodinia_3.1 on a machine with an AMD 8-core processor and 8x M.2. Edge TPUs. The machine hosts a Ubuntu Linux 16.04 system with both TensorFlow 1.13.0 and with libedgetpu frameworks installed. These libraries are essential for driving the Edge TPU hardware to run compatible models with our desired instructions. The baseline GPU applications are mainly from Rodinia_3.1 benchmark suite, except GEMM is directly derived from call NVIDIA's cuBLAS. The GPU hardware used for baseline are GeForce RTX 2080 with CUDA 11.0 and driver 450.51.06, and jetson nano 4G.

Author-Created or Modified Artifacts:

```
Persistent ID: https://anonymous.4open.science/r/cd0」
↪  5e700-cd63-43c7-814f-11fbcc459eb9/
Artifact name: GPTPU open-sourced code

Persistent ID: "ssh sc21@escal.escalab.org -p 425"
↪  and then "ssh gengar" with both pw: 21scea. (make
↪  all to compile, make run to actual run)
Artifact name: GPTPU open-sourced code on an local
↪  machine

Persistent ID: 10.5281/zenodo.5156431
Artifact name: The DOI GPETPU code used for the AE
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: AMD Ryzen 7 3700X 8-Core 113, Google M.2. EdgeTPU

Operating systems and versions: Ubuntu 16.04.6 running Linux kernel 4.15.0

Compilers and versions: g++ 5.4.0

Applications and versions: rodinia_3.1

Libraries and versions: tensorflow 1.13.0 , github-libedgetpu

Key algorithms: General Matrix Multiplication

Input datasets and versions: rodinia_3.1 dataset