Comparing GPU and TPU in an Iterative Scenario: A Study on Neural Network-based Image Generation

Roman Lehmann[1], Paul Schaarschmidt[1], and Wolfgang Karl[1]

Abstract: This paper explores the utilization of TPUs (Tensor Processing Units) and GPUs (Graphics Processing Units) in iterative applications involving neural networks. We employ a Pix2Pix approach for computing sequential flows, evaluating the effectiveness in scenarios where NNs are only a component of the system. While TPUs demonstrate performance improvements during training with large batch sizes, we observe no significant acceleration during inference compared to GPUs. The study highlights the need to carefully consider workload and system architecture when incorporating TPUs, emphasizing that their advantages are more prominent in training tasks.

Keywords: TPU, GPU, Pix2Pix, XLA

1 Motivation

Neural networks (NN) have become indispensable in numerous industrial and scientific fields. In practice, these networks are often utilized as a component of larger applications, such as in iterative processes, where the NN plays a role in computing specific computational steps. For this research, we focus on the simulation of the Kármán vortex street (see Fig. 1). For this, we don't use a numerical method. Instead, we adopt a Pix2Pix [Is16] approach to an initial flow image and calculate subsequent images (next time steps) by a NN.

Fig. 1: Representation of the Kármán vortex street: The fluid flows from the left side into a channel and flows around an object on the left side to the right side of the channel. Typical, counter-rotating vortices are created behind the object.

The core computations within the neural network predominantly involve matrix-vector multiplications and additions. Today, these tasks are primarily executed on a combination of a CPU and a GPU, where the GPU is used as an accelerator. Originally designed for graphical applications, GPUs perform well on these tasks [SBS05]. Similar to GPUs for graphical applications, in recent years there have been developments for domain-specific

1 Karlsruhe Institute of Technology (KIT), CAPP, Haid-und-Neu-Str. 7, 76131 Karlsruhe, Germany, roman.lehmann@kit.edu; roman.lehmann@kit.edu; wolfgang.karl@kit.edu

processors (DSP) in the field of neural networks. For example, Google's Tensor Processing Units (TPUs) were developed to handle machine-learning tasks more efficiently.

This paper analyzes GPU and TPU performance in an application, where only a portion of the application's calculations involve a neural network. We provide an illustrative examination of an iterative process in which a neural network is employed to simulate flow images of a Kármán vortex street.

Section 2 provides a comprehensive review of the related literature. In the following Section 3, we outline our approach to the image generation method. We then move on to the evaluation, starting with an introduction to our hardware setup, including a brief overview of TPUs. Lastly, the paper concludes with a comprehensive summary.

2 Related Work

About DSP, especially TPUs, there are already some studies related to NNs [Ju22, SGG21, Ni22, KPK21, Ra22, WWB19]. [Ni22] deals exclusively with the hardware itself and compares CPUs, GPUs, and TPUs on a technical level. Other works dealt with different models of NNs but focused exclusively on the training of the networks since this takes less time compared to inference [WWB19, SGG21]. Advanced training methods, such as distributed learning, have also been evaluated on TPUs [KPK21]. On the other hand, [Ju22] looks at a Graph Neural Network (GNN) applied to a dataset with a sparse representation (unlike the above). For the GNN, they use an encoder-decoder structure. Despite their differences, all of these works tend to agree on one point: TPUs offer substantial power, but their performance depends on the application. In training, the TPU is convincing in applications with very deep NNs or applications where training can be performed with high batch sizes. On sparse data, TPU is similarly efficient in training compared to the GPU.

In this paper, we do not focus on training a NN but instead concentrate on inference. Moreover, our scope goes beyond examining just the model itself. We integrate the model into an application that generates flow images of a Kármán vortex street. Similar to [Ju22], we also employ an encoder-decoder structure. In our case, it is a U-Net model combined with a Pix2Pix approach [Is16, RFB15]. This structure is iteratively used to generate flow images for different time steps. Through this approach, we contribute additional knowledge to the existing literature by applying TPUs in a real-world application.

3 Method of Image Generation

Our goal is to simulate a Kármán vortex street. But, we do not use numerical methods. We use an image-to-image (Pix2Pix) approach based on a NN instead. The model comprises a U-Net (see Fig. 2) trained using a cGAN approach [Is16, RFB15]. The U-Net takes a 2-channel image as input, with dimensions of 1024 by 256 pixels. Each channel represents the

fluid velocity in the x- and y-direction, respectively, and is mapped to the interval $[0, 255]$ to achieve a grayscale image.

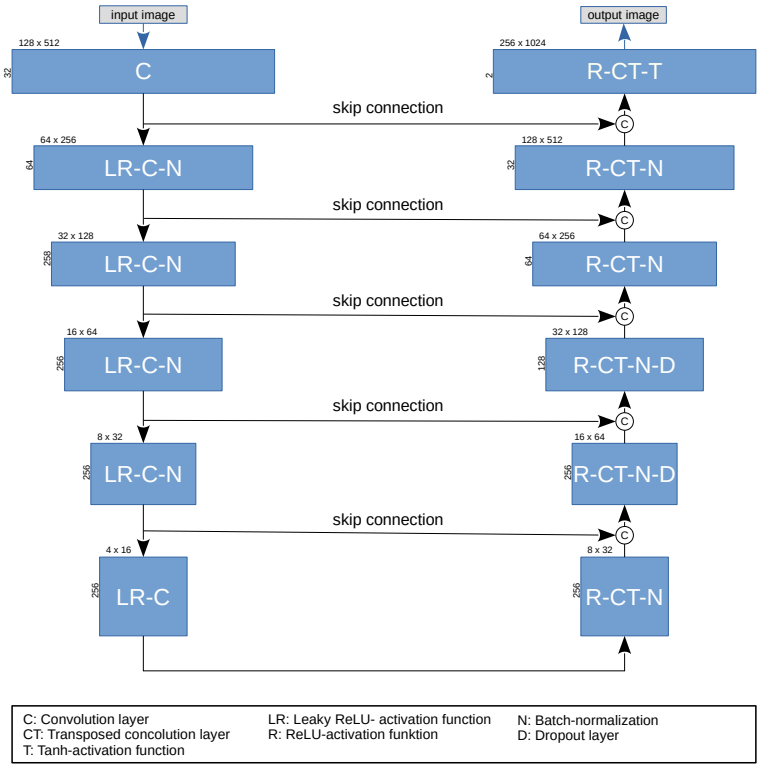

Fig. 2: Structure of the used U-Net with 6 layers: The input image is encoded on the left side (encoder) and the result is decoded on the right side (decoder).

This model is built into our method. This method (illustrated in Fig. 3) consists of a total of 6 stages, which are then iterated through to generate the flow images. The individual stages have the following tasks:
The *image_preparer* stage loads the i-th image and performs preprocessing steps. For example, normalization is done to map the values of the input to $[0, 1]$.
At the *executor* stage, the aforementioned neural network is used.
Image_postprocessor stage: To maintain the object's integrity within the flow, a binary mask is applied to the output image [Le20]. This step ensures that the object boundaries are preserved.
The *metadata_handler* stage calculates and saves metrics about the result like the mean-squared-error, peak-signal-noise-ratio, correlation and structural similarity.

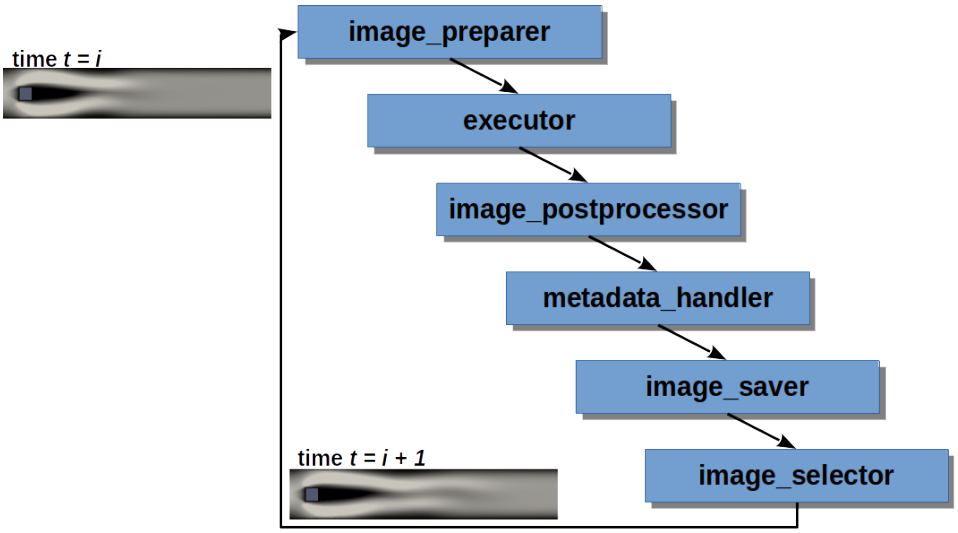

Fig. 3: Method for image generation: An image of the flow at time $t = i$ is input to our method and passes through the 6 stages, generating the flow image at time $t = i + 1$. The generated flow image serves as the input for the next iteration. .

Subsequently, the *image_saver* stage denormalizes the newly generated image $i + 1$ and writes it for future reference or analysis to hard disk. This is implemented in parallel with a producer-consumer pattern, so the next steps don't have to wait for saving.
Lastly, the *image_selector* stage determines the next image in the sequence, governing the iterative process.
After executing the steps, the scheme starts again from the beginning, with the just created image as input and so successive images of a flow are generated for the different time steps.

Furthermore, in addition to the previously mentioned *executor*, an alternative approach for the *executor* has been developed. This alternative approach aims to enhance the simulation's quality. For training, the large flow images (1024 by 256 pixels) are subdivided into smaller, square images (128 by 128 pixels). This strategy extends the dataset and improves the training of the NN. Then, the *executor* is transformed into a scatter-gather pattern. This entails initially dividing the incoming image into sub-images. Subsequently, these sub-images are processed individually by identically trained networks. The resultant sub-images are then combined to create a single, larger image. This method unlocks additional possibilities for parallelization. For instance, individual NNs can be mapped to different computing units, and the results can be reduced into a unified outcome. We, therefore, call this *map-reduce*. In the context of TPUs, this also implies a more straightforward exploitation of multiple TPU cores, which we also put to the test.

4 Evaluation

4.1 Hardware Setup

In this study, we utilized *PyTorch* version 1.8.1 as the programming environment for our testing and training processes [Pa19].

For conducting the tests and training, we employed a local workstation comprising two AMD EPYC 7F32 CPUs, one Nvidia RTX A6000 GPU, and 128 GiB of RAM. The Nvidia RTX A6000 GPU is capable of delivering 38.7 TFLOPS for single-precision computations, without utilizing the sparsity feature [Nv20].

To compare the performance, we also leveraged Google's TPU Research Cloud [Go18], which grants access to multiple TPUs. Specifically, we employed both TPUv2 and TPUv3 versions. TPUs consist of several so-called TensorCores computing units. TensorCores have two basic components: a scalar vector unit (SVU) and a matrix multiplication unit (MXU). The basic architecture of this concept can be found in Fig. 4.

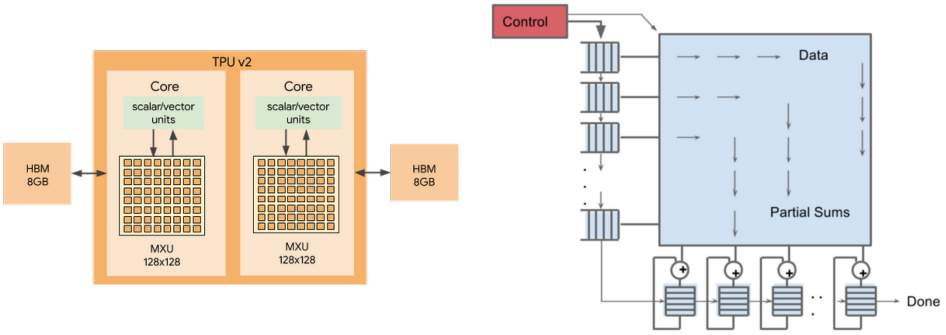

Fig. 4: Each core of a tensor unit consists of an MXU and an SVU (left). The MXU is realized according to the systolic array (right) [SY17, Gö21].

The MXU is specifically designed for efficient and extensive matrix multiplications. These operations are widely used in machine learning. The MXU is built upon systolic arrays, a homogeneous network of tightly coupled cells optimized for linear algebraic operations on densely populated matrices [SY17]. Physically, an MXU is implemented as a hard-wired hardware matrix, eliminating the need for read or write accesses to memory or registers, thereby ensuring high performance and energy efficiency [Gö21]. This contrasts with the conventional Von Neumann architecture, where frequent data exchange between the processor and memory via bus usually creates a bottleneck [Ba78].

The distinction between these two versions v2 and v3 lies in the increased internal memory of TPUv3, offering 32 GiB of High-Bandwidth Memory (HBM) as opposed to TPUv2, which provides 16 GiB of HBM. Additionally, the TPUv3 TesorCores consist of 2 MXU and SVU each and thus twice as many as in v2. Further, the TPUv3 demonstrates superior

integration density, yielding a total of 123 TFLOPS per core, translating to 0.56 watts per TFLOPS. On the other hand, the TPUv2 achieves 45 TFLOPS per core, equivalent to 0.16 watts per TFLOPS [Go19].

Different hardware, especially different hardware architectures, are always difficult to compare. However, the TPUs should be superior to the A6000 in terms of performance data alone.

4.2 Results

For evaluation, we conducted time measurements for each of the six stages mentioned in Section 3. Fig. 5 shows the times of the 6 stages. It can be seen that the use of TPUv3 (green) and TPUv2 (red) does not lead to an acceleration to any stage compared to the A6000 (black). While *image_selector* and *image_preparer* have about the same times, *metadata_handler* is many times slower on the system with the TPU. It should also be noted that the generation jump between v2 and v3 does not affect the performance in our scenario.

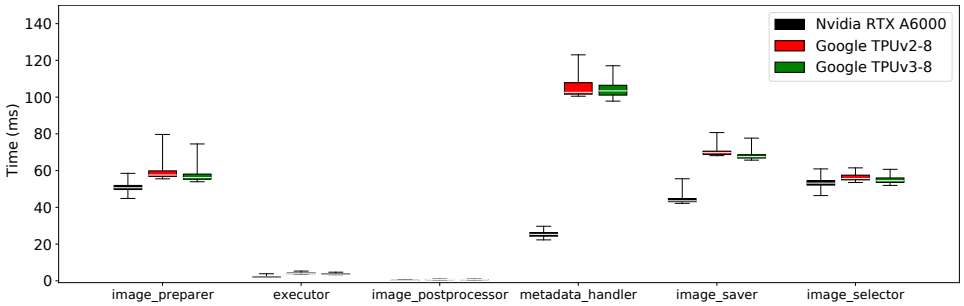

Fig. 5: Time measurements of the six stages. Google's TPUv2 and TPUv3 utilize 8 cores, respectively.

The results reveal a clear bottleneck in the application, which does not lie in the execution of the neural network. Instead, the other stages of the method carry much more weight and present a significant I/O bound. Unfortunately, these stages do not benefit from the superior hardware of the TPU, resulting in no significant acceleration compared to other hardware. Even when focusing solely on the *executor* stage, we also notice this stage has not experienced any acceleration either. The *map-reduce* approach also means an increased effort for the executor. However, we expect that the TPUs can be better used by the simplified splitting, and thus a speedup is achieved.

But, the *map-reduce* approach shows similar results. In Fig.6, the execution time of the *executor* stage is visualized. The measured time of the A6000 is presented in black. We tested different numbers of TensorCores on the TPUs. In red (orange) the time of TPUv2 with 8 (32) TensorCores is presented and in green (blue) the time of TPUv3 with 8 (32) TensorCores is shown. It is evident, that using the TPU does not lead to acceleration compared to using the GPU, even with an approach that plays into the TPU's cards (*map_reduce*). Increasing the

number of TensorCores does result in reduced processing time, but we have not achieved the anticipated linear acceleration, as proposed by Google [Go23].

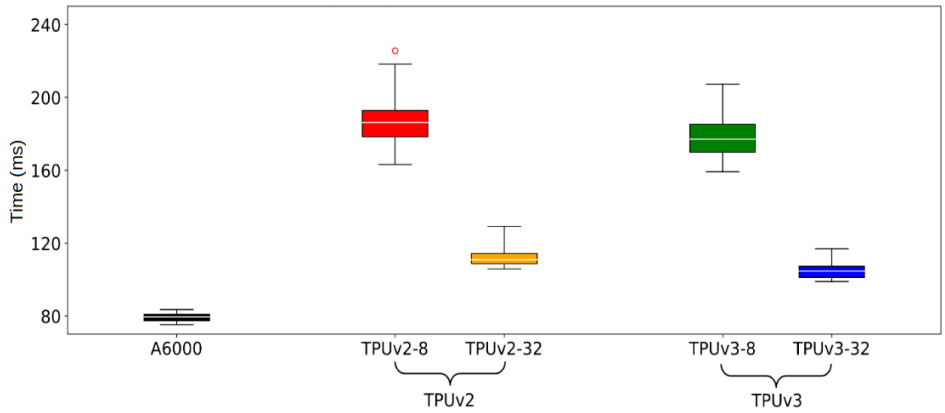

Fig. 6: Time measurements of the *executor* stage in *map-reduce* mode. Google's TPUv2 and TPUv3 utilize 8 or 32 cores, respectively.

The question now is why the TPU is below expectations. On the spec sheet, the TPU should lead to acceleration. Why this is not the case is due to the execution process on a TPU. A classic CPU is built according to the von Neumann principle and is thus very flexible. However, this flexibility comes with a crucial disadvantage: the data must be written back to memory after each operation. This is also called the von Neumann bottleneck [Ba78]. GPUs are already a further development here. They have thousands of individual ALUs and can thus execute thousands of operations. However, GPUs were initially designed for graphical purposes and, as a result, they must continue to support a wide range of diverse applications. For every calculation in the thousands of ALUs, a GPU must access registers or shared memory to read operands and store the intermediate calculation results.
In contrast, the TPU is specifically designed for neural networks. However, the accelerator cannot be used directly. The code to be executed must be compiled by an XLA (Accelerator Linear Algebra) compiler. This compiler is a just-in-time compiler that takes the graph output from an ML framework application and compiles the linear algebra, loss, and gradient components of the graph into TPU machine code. And this is exactly the problem with our application. This step must be applied to every call. There is no possibility to keep the binary and thus save the compiler steps. This is not a disadvantage if you want to speed up training tasks with large batch sizes. Due to the high workload caused by batch sizes and training, the additional effort is negligible, and the advantages outweigh the disadvantages. Namely, it is very easy for the user to use a TPU. The TPU is connected via the compiler and almost no program changes are necessary compared to programming with a GPU under PyTorch. We can also observe this behavior in our iterative application. In training, we achieve a speedup of 1.4 compared to our GPU. And this with significantly lower power consumption.

5 Conclusion

In this study, we have investigated the impact of utilizing TPUs (Tensor Processing Units) on iterative applications, which are predominantly I/O bound. Our research aimed to evaluate the effectiveness of TPUs in such scenarios. Our findings indicate that using TPUs might not be worthwhile for certain tasks, particularly during the execution of inference, as we observed no significant advantages with this new technology. However, it is important to highlight that in the training phase, TPUs did demonstrate performance improvements.

A scatter-gather pattern (*map-reduce*) makes it easier to claim more TrensorCores. Thus, the overhead caused by the just-in-time compiler decreased. Consequently, it could be considered to exploit this behavior. For example, one could build a neural net structure into the method that uses ensemble inference [Fu15, RV20].

In conclusion, our study emphasizes the need to carefully consider the specific nature of iterative applications when incorporating TPUs. While their advantages are evident in training, they might not offer substantial benefits during inference. Nevertheless, the *map-reduce* strategy and utilizing ensemble inference techniques could pave the way for harnessing TPUs more effectively in certain scenarios.

Acknowledgments

We thank the [Google TPU Research Cloud] (`https://sites.research.google/trc/about/`) program for providing part of the computation resources.

Bibliography

[Ba78] Backus, John: Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. Commun. ACM, 21(8):613–641, 1978.

[Fu15] Fukui, Hiroshi; Yamashita, Takayoshi; Yamauchi, Yuji; Fujiyoshi, Hironobu; Murase, Hiroshi: Pedestrian detection based on deep convolutional neural network with ensemble inference network. In: 2015 IEEE Intelligent Vehicles Symposium (IV). pp. 223–228, 2015.

[Go18] Google: TPU Research Cloud. Accessed: Aug. 01, 2023. [Online], 2018.

[Go19] Google: System Architecture. Accessed: Aug. 01, 2023. [Online], 2019.

[Gö21] Görner, Martin: Keras and modern convnets, on TPUs. Accessed: Aug. 01, 2023. [Online], 2021.

[Go23] Google: Google's Cloud TPU v4 provides exaFLOPS-scale ML with industry-leading efficiency. Accessed: Aug. 01, 2023. [Online], 2023.

[Is16] Isola, Phillip; Zhu, Jun-Yan; Zhou, Tinghui; Efros, Alexei A.: Image-to-Image Translation with Conditional Adversarial Networks. CoRR, abs/1611.07004, 2016.

[Ju22] Ju, Xiangyang; Wang, Yunsong; Murnane, Daniel; Choma, Nicholas; Farrell, Steven; Calafiura, Paolo: Benchmarking GPU and TPU Performance with Graph Neural Networks, 2022.

[KPK21] Kimm, Haklin; Paik, Incheon; Kimm, Hanke: Performance Comparision of TPU, GPU, CPU on Google Colaboratory Over Distributed Deep Learning. In: 2021 IEEE 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC). pp. 312–319, 2021.

[Le20] Lehmann, Roman; Arnaudov, Stanislav; Hoffmann, Markus; Karl, Wolfgang: Binary Maps for Image Separation in Iterative Neuronal Network Applications. In: Forum Bildverarbeitung 2020. Ed.: T. Längle ; M. Heizmann. KIT Scientific Publishing, pp. 363–375, 2020.

[Ni22] Nikolić, Goran S.; Dimitrijević, Bojan R.; Nikolić, Tatjana R.; Stojcev, Mile K.: A Survey of Three Types of Processing Units: CPU, GPU and TPU. In: 2022 57th International Scientific Conference on Information, Communication and Energy Systems and Technologies (ICEST). pp. 1–6, 2022.

[Nv20] Nvidia: Nvidia RTX A6000. Accessed: Aug. 01, 2023. [Online], 2020.

[Pa19] Paszke, Adam; Gross, Sam; Massa, Francisco; Lerer, Adam; Bradbury, James; Chanan, Gregory; Killeen, Trevor; Lin, Zeming; Gimelshein, Natalia; Antiga, Luca; Desmaison, Alban; Köpf, Andreas; Yang, Edward Z.; DeVito, Zach; Raison, Martin; Tejani, Alykhan; Chilamkurthy, Sasank; Steiner, Benoit; Fang, Lu; Bai, Junjie; Chintala, Soumith: PyTorch: An Imperative Style, High-Performance Deep Learning Library. CoRR, abs/1912.01703, 2019.

[Ra22] Ravikumar, Aswathy; Sriraman, Harini; Sai Saketh, P. Maruthi; Lokesh, Saddikuti; Karanam, Abhiram: Effect of neural network structure in accelerating performance and accuracy of a convolutional neural network with GPU/TPU for image analytics. PeerJ Computer Science, 8:e909, 3 2022.

[RFB15] Ronneberger, Olaf; Fischer, Philipp; Brox, Thomas: U-Net: Convolutional Networks for Biomedical Image Segmentation. In (Navab, Nassir; Hornegger, Joachim; Wells, William M.; Frangi, Alejandro F., eds): Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015. Springer International Publishing, Cham, pp. 234–241, 2015.

[RV20] Ruiz, Adria; Verbeek, Jakob: Anytime Inference with Distilles Hierarchical Neural Ensemble. CoRR, abs/2003.01474, 2020.

[SBS05] Steinkraus, Dave; Buck, Ian; Simard, Patrice Y: Using GPUs for machine learning algorithms. In: Eighth International Conference on Document Analysis and Recognition (ICDAR'05). IEEE, pp. 1115–1120, 2005.

[SGG21] Sharma, Vijeta; Gupta, Gaurav Kumar; Gupta, Manjari: Performance Benchmarking of GPU and TPU on Google Colaboratory for Convolutional Neural Network. In: Applications of Artificial Intelligence in Engineering. Springer Singapore, Singapore, pp. 639–646, 2021.

[SY17] Sato, Kaz; Young, Cliff: An in-depth look at Google's first Tensor Processing Unit (TPU). Accessed: Aug. 01, 2023. [Online], 2017. https://cloud.google.com/blog/products/ai-machine-learning/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu?hl=en.

[WWB19] Wang, Yu; Wei, Gu-Yeon; Brooks, David: Benchmarking TPU, GPU, and CPU Platforms for Deep Learning. CoRR, abs/1907.10701, 2019.