Faculty of Engineering Sciences

University of Heidelberg

Master thesis

in Computer Engineering

submitted by

Christian Alles

born in Heidelberg

2023

On the Performance of Butterfly Approximations on the Graphcore IPU

This Master thesis has been carried out by Christian Alles

at the

Institute of Computer Engineering

under the supervision of

Prof. Dr. Holger Fröning

Abstract

Over the past decade, the most commonly used hardware for accelerated computing has been the GPU, as it can achieve higher throughput than a CPU for a similar power consumption. In recent years, due to advances in machine learning, a number of custom parallel processing units have been released, out of which, the Intelligence Processing Unit (IPU) is based on the world's first graph toolchain designed for machine intelligence. This thesis investigates whether the IPU can act as a replacement for a GPU with similar transistor size, power consumption and release date for certain workloads. To achieve this, a performance baseline is first established with various benchmarks and characterized using a range of profiling tools. The results and the target group of the IPU lead us to investigate machine learning workloads with a focus on Butterfly Approximations for sparsification. It is found that the IPU can outperform a comparable GPU by up to a factor of 4.5, with the main bottleneck being limited memory.

Zusammenfassung

Im letzten Jahrzehnt war die am häufigsten verwendete Hardware für beschleunigtes Rechnen die GPU, da sie bei ähnlicher Leistungsaufnahme einen höheren Durchsatz als eine CPU erzielen kann. In den letzten Jahren wurden aufgrund von Fortschritten im Bereich des maschinellen Lernens eine Reihe spezieller paralleler Prozessoren auf den Markt gebracht, von denen die Intelligence Processing Unit (IPU) auf der weltweit ersten Graph-Toolchain basiert, die für maschinelle Intelligenz entwickelt wurde. In dieser Arbeit wird untersucht, ob die IPU als Ersatz für eine GPU mit ähnlicher Transistorgröße, Leistungsaufnahme, vergleichbaren Veröffentlichkeitsdatum und bei bestimmten Anwendungen dienen kann. Zu diesem Zweck wird zunächst mit Hilfe verschiedener Benchmarks eine Performance-Baseline geschaffen und mit einer Reihe von Profiling-Tools charakterisiert. Die Ergebnisse und die Zielgruppe der IPU veranlassen uns dazu Anwendungen des maschinellen Lernens zu untersuchen, wobei der Fokus auf Butterfly-Approximationen zur Sparsifizierung liegt. Es zeigt sich, dass die IPU eine vergleichbare GPU um bis zu einem Faktor 4.5 übertreffen kann, wobei der hauptsächlich limitierende Faktor der begrenzte Speicher ist.

Contents

List of Acronyms

AI	Artificial Intelligence
AMP	Accumulating Matrix Product Unit
ASIC	Application-Specific Integrated Circuit
BFS	Breadth-First Search
BSP	Bulk Synchronous Parallel
COO	Coordinate List
CSC	Compressed Sparse Column
CSR	Compressed Sparse Row
CPU	Central Processing Unit
DCT	Discrete Cosine Transform
DDR	Double Data Rate
DFT	Discrete Fourier Transform
DRAM	Dynamic Random Access Memory
FLOPs	Floating Point Operations per Second
GPU	Graphics Processing Unit
HPC	High Performance Comupting
IPU	Intelligence Processing Unit
MIMD	Multiple Instructions Multiple Data
ML	Machine Learning
MM	Matrix Matrix Multiplication
NIC	Network Interface Card
RDMA	Remote Direct Memory Access
SM	Streaming Multiprocessor
SIMT	Single Instruction Multiple Threads
SRAM	Static Random Access Memory
TDP	Thermal Design Power
TPU	Tensor Processing Unit

1 Introduction

Parallel computing has always been considered as both, a scientific field of research and a tool to enhance different applications such as AI, astronomy, medicine, data mining, etc. During the last decades, GPUs were widely used to accelerate CPU computation due to their compute power, programmability, support by the manufacturer, and their community. Recent progress in ML, with its ever increasing model sizes [38], has caused an exponential growth in demand for computational resources, with a corresponding increase in energy use. This need, coupled with the decline of Moore's Law and the fade-out of Dennard Scaling, has led to the advent of custom parallel hardware dedicated for these ML workloads, such as the Tensor Processing Unit (TPU) and the Intelligence Processing Unit (IPU), but also to GPUs specialized for ML workloads like the H100 by NVIDIA. An example of an AI accelerator that supports floating point computation, half and single precision, is the IPU. As the day of writing, Graphcore has released its third generation chip called Bow, relying on a Wafer-on-Wafer 3D stacking technology as the main difference to the previous generation [32].
As an AI accelerator, the IPU includes dedicated hardware for most common operations such as MM, and, in contrast to a GPU, a high amount of on-chip SRAM. Apart from the higher level frameworks used for ML, POPLAR, a low-level C++ framework, is also supported. Both frameworks are used with regard to their commonly used context, POPLAR for HPC and PopTorch for ML applications.
From the properties described before arises the motivation for this thesis: To what extend and in which applications can the IPU outperform the GPU? Can we apply the same methods from the GPU to the IPU or do we have to implement everything from scratch? How well does the IPU handle sparse data as it is a major performance caveat for the GPU? How does the architecture differ to the GPU and what implications does this have?
To answer these questions, this work makes three main contributions:

- **Performance Evaluation of the IPU**
  We evaluate the second generation GC200 IPU in terms of performance, programmability and power consumption and compare against a GPU with a similar power budget, transistor size and year of release.
  We analyze the interconnect via point-to-point data transfers with different data sizes in an congestion-free network to verify the results of [39] for the GC200 IPU

and to review if sparse data is suitable for the IPU.
To evaluate the performance of the IPU, we choose MM as this operation appears in different workloads such as ML [14], image and signal processing [63]. First, we run MM with dense squared matrices to verify the results of [39] for the GC200 IPU and to get a performance baseline against the GPU. To add more insights to the findings of [39], we analyze the underlying behavior on both platforms with the respective profiling tools.
We extend the squared dense with skewed dense and squared sparse input matrices and apply the same methodology.
Additionally, we compare and analyze the lower-level frameworks (POPLAR and CUDA) with the ML frameworks (PopTorch and PyTorch).

- **Characterization of Butterfly Approximations**
  The second contribution of this thesis is the application of various structured sparse matrices. We focus on `Butterfly` [10] and `Pixelated Butterfly` [6] as these show improvements over linear layers and other structured approaches in terms of performance, applicability and model compression on the GPU. For both, we evaluate if the claims hold true for a newer generation of GPU with dedicated MM execution units and how the performance translates to the IPU. To do so, we run the experiment from [10], section 4.3 with both approaches. In addition to [10], we perform it with squared matrices to allow a comparison to the performance evaluation of the IPU. To examine how both approximations perform against various structured sparse layers, we run the single-hidden layer benchmark from [10] with the MNIST and CIFAR10 datasets.

- **Optimizations of Butterfly Approximations for the IPU**
  The third contribution of this thesis focuses on optimizations of the Butterfly Approximations for the IPU. We investigate on the implications on execution time, test accuracy and parameter count resulting from the block size, low rank size and butterfly size for `Pixelated Butterfly` on the single-hidden layer benchmark with the CIFAR10 dataset. Additionally, we provide a deeper analysis of `Butterfly` on the IPU for the same application and evaluate possible optimizations with focus on load balancing and the injection of POPLAR code to accelerate the computation.

This thesis is structured into seven chapters. The following third chapter, introduces the IPU and gives an overview of the hardware properties and the Programming Model. Butterfly Approximations, their usage and background are also explained in this chapter. Chapter

four contains the first contribution of this work in which we evaluate the IPU performance. In this chapter, we investigate on the properties of the memory system with point-to-point data transfers. Matrix multiply performance on the IPU is considered as well with different input matrix dimensions and different numbers of nonzero elements. In addition, we analyze and compare against GPU performance numbers. After that, chapter five examines Butterfly Approximations for the IPU and GPU, compares and analyzes them. To do so, we compare the approximations against a linear layer in terms of performance and then evaluate against other structured sparse approaches in the single-hidden layer benchmark with two different datasets. Chapter six extends the previous with optimizations for Butterfly Approximations for the IPU. Finishing this thesis is the conclusion that discusses the findings and an outlook to future work.

2 Background

2.1 Intelligence Processing Unit (IPU)

Since the GPU only serves as a performance baseline, we decide to omit a thorough explanation. Fig.2.1 shows the general differences between CPU, GPU and IPU. To allow a fair comparison, we opted for similar release dates, transistor sizes and power envelopes for the GPU and IPU.
The GC200 IPU was released in July 2020 with a TDP of 150 W [4] and a transistor size of 7 nm [17]. The chosen A30 GPU was released in April 2021 with a TDP of 165 W and a transistor size of 7 nm [3]. These properties allow a fair comparison between the chosen GPU and the IPU.

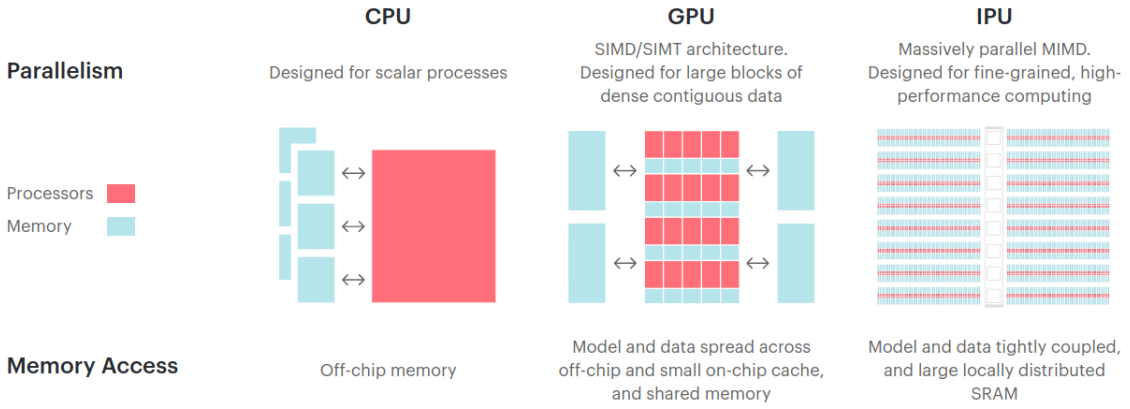


Figure 2.1: Processor Comparison [30]

The IPU, developed by Graphcore, is a processor designed for highly parallel workloads, making it particularly well suited in a wide range of applications including ML and graph processing algorithms. As the IPU is far less common than the GPU and as both are accelerators, we explain most properties of the IPU by comparing them to the GPU.
Both processing units rely on the contributions from [70] with the main aspects being parallel slackness and the BSP execution in supersteps. Parallel slackness refers to the ability to schedule and pipeline computation and communication efficiently, by having much more virtual than physical processors. The BSP execution model is a widely adapted model for parallel architectures, including supersteps that consist of a Compute, Communication

and Synchronization phase.
The IPU incorporates these aspects by (1) being able to schedule six threads [32] in a round-robin fashion [25] for each IPU-Core and (2) by following exactly the BSP execution scheme in supersteps. On the IPU, this translates to three phases: Compute, Sync and Exchange [25]. **Compute Phase:** Each IPU-Tile of an IPU executes independently. **Sync Phase:** Every IPU-Tile and thus every IPU-Core is synchronized via the all-to-all interconnect. **Exchange Phase:** Data is transferred between the different IPU-Tiles or Streaming Memory [25]. Fig.A.2 in the Appendix shows the three phases of the BSP execution on the IPU.
For the GPU, the aspects from [70] are achieved by (1) launching more thread blocks than SMs available. Also, the superstep execution is included on the GPU as barrier synchronization always takes place after communication and computation within a thread block, leading to the next superstep.
The GPU and the IPU do not differ much in terms of their execution scheme, however, the realization of this scheme and the properties of the systems diverge.
Considering Flynn's taxonomy, the two processing units already differ in their classification. In contrast to a GPU, which is categorized as SIMD/SIMT, the IPU is regarded as MIMD, due to the IPU-Tiles being able to execute multiple-, independent threads on unrelated data located on the corresponding In-Processor Memory [4].
These differences also come apparent by considering the different hardware properties. Although both accelerators have roughly the same power budget, the general structure differs greatly.
One key difference between the GPU and the IPU is the complexity of the architecture. The IPU consists of two building blocks: IPU-Tiles and the IPU-Exchange. IPU-Tiles are used for computation while the IPU-Exchange allows communication among them. The latter will be explained in subsection 4.1.3. Each IPU-Tile consists of an IPU-Core and In-Processor Memory.
The GPU, in comparison, has a much more complex architecture, resulting in both, a thread hierarchy and a memory hierarchy. A thread grid consists of one or more thread blocks which, in turn, also consist of one or more thread warps that contain a fixed number of threads, usually 32 [59]. For the IPU, there is no thread hierarchy [25].
The same applies to the memory architecture and the memory accessibility on both systems. The GPU features a more steep memory architecture than the IPU. Every thread from a thread block can access the Global Memory which translates to some GBytes of DRAM memory. This memory is also the one that is exchanged with the host. The smaller on-

chip memory, called Shared Memory, with lower latencies can only be accessed within a thread block, allowing no exchange of data between different thread blocks. Apart from the Global and Shared Memory, there are also registers and, depending on the generation, caches. For the IPU, there is only a single type of memory accessible by the IPU-Cores: In-Processor Memory, which is 900 MB of SRAM, distributed equally over the IPU-Tiles [32]. The In-Processor Memory per IPU-Tile is only accessible by its local IPU-Core, resulting in a lack of global memory accesses and a higher aggregate memory bandwidth, due to the concurrent data access. Data not local to its respective IPU-Core has to be transferred via the non-blocking interconnect called IPU-Exchange. Similar to a GPU with its off-chip global memory, the IPU is also able to access data which is not residing on the processing unit via a type of memory called Exchange Memory. In contrast to a GPU, which includes a single chip, the IPU-Machine contains at least four GC200 IPUs. To make the comparison as fair as possible, we normalize the available memory to a single IPU.
The IPU-Machine has up to 260 GB of Exchange Memory accessible to the IPUs in the system. The Exchange Memory is comprised of up to 256 GB Streaming Memory and 3.6 GB In-Processor Memory [33]. Each IPU is able to access $25\,\%$ of the total Exchange Memory. That translates to the previously mentioned 900 MB of In-Processor Memory and up to 64 GB of Streaming Memory.

Communication

Ethernet connects the IPU-Machine to a host compute. The data is transferred between the host and the IPU-Machine via the RDMA NICs on both Nodes. The NIC on the IPU-Machine loads and stores data to Streaming Memory memory. The IPU will then read and write to Streaming Memory during its Exchange phase. In this phase, data can either be transferred within the IPU-Tiles or between Streaming Memory and the IPU [25]. This communication architecture is depicted in Fig.2.2. Communication on the IPU-Machine is divided into intra-IPU and inter-IPU. From a hardware perspective, the IPU-Exchange is responsible for the all-to-all intra-IPU transfers with $8\,\mathrm{TB\,s^{-1}}$ peak bandwidth. For transfers between separate IPUs, the IPU-Links are used, allowing allows $320\,\mathrm{GB\,s^{-1}}$ chip to chip bandwidth while supporting up to ten devices at once [32]. The Programming Model does not differentiate between intra-IPU and inter-IPU communication. This simplifies porting a given code from a single IPU to multi-IPU configurations, allowing scalability [39].

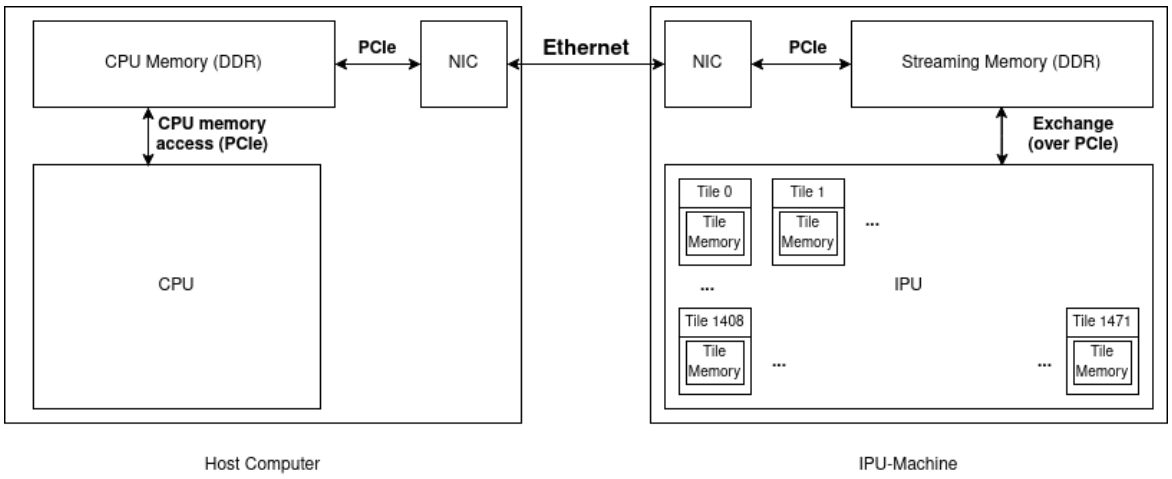


Figure 2.2: Host to IPU Communication

Programming Model of the IPU

Although both being massively parallel processors, the programming models differ substantially between GPU and IPU. This is mainly due to the underlying memory architecture and the execution model, requiring distinct compilers for each platform (nvcc for GPU, popc for IPU).
IPU-Programs are represented by a computational dataflow graph as depicted in Fig.2.3. In this graph, computation is represented as nodes (Vertices), data as Tensors and the flow of these as Edges. Those Vertices consist of Vertex code and connected data as Tensors. Each Vertex can be mapped to IPU-Tiles and execute independently.

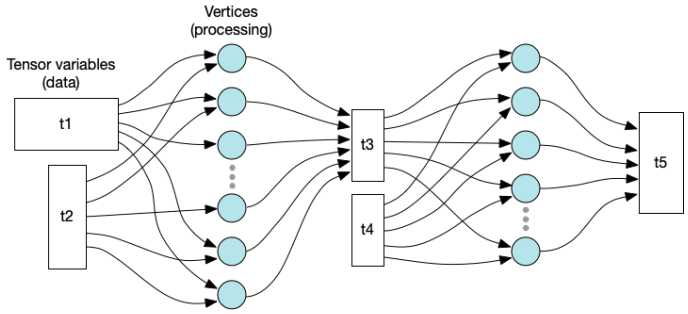


Figure 2.3: POPLAR Computational Dataflow Graph [25]

IPU Software Stack

The IPU can only be programmed via its dedicated software stack. Adapting code for the IPU depends highly on the abstraction layer. While it is possible to run AI applications on the IPU by changing only a couple lines of code [27], doing so for the lower-level frameworks is not possible. This is due to the underlying hardware differences. Fig.2.4 shows the software hierarchy of the IPU, with two frameworks, namely POPLAR and PopTorch, highlighted as these are used in this work.

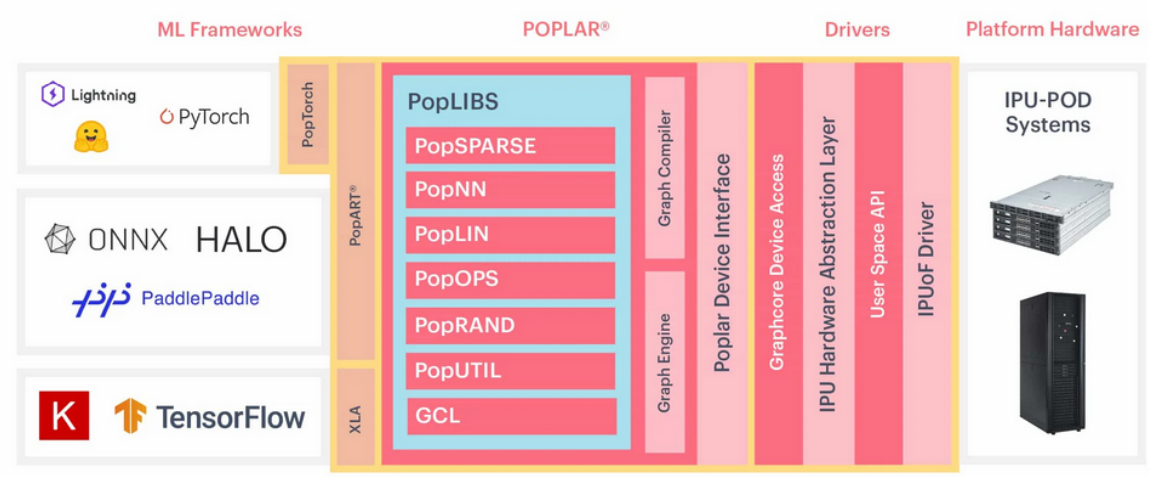


Figure 2.4: IPU Software Hierarchy [34]

Apart from the common ML frameworks such as TensorFlow and PyTorch, Graphcore is also providing low-level programming features with its open-source POPLAR framework. As Fig.2.4 shows, the frameworks include PopLIBS, which is a set of libraries that offer commonly needed functionality like random number operations (PopRAND), neural network functions (PopNN) and linear algebra functions for dense (PopLIN) and sparse (PopSPARSE) Tensors [18].
In POPLAR, the computational graph shown in Fig.2.3 can either be created via POPLAR or PopLIBS. In POPLAR, the Vertex code and the mapping of Tensors and Vertices to IPU-Tiles is defined manually. With PopLIBS, both tasks can be handled by the library functions.
PopTorch builds on top of PyTorch. Its purpose is to run PyTorch models on the IPU with as few changes as possible. Running native PyTorch is not possible on the IPU due to different requirements to the model and a lack of some PyTorch functionality. In contrast to PyTorch, PopTorch creates a static compiled graph, see Fig.2.3. PopTorch compiles

its models into POPLAR executables which also explains the Programming Model for the low-level framework. Both, training and inference are supported [27].

2.2 Butterfly Approximations

In the field of ML, fast linear transform functions such as the DFT or DCT are used to speed up training and inference. The underlying challenge is to find an appropriate transform with an efficient implementation for a given model. As every structured linear transform can be described with dense matrix-vector multiplication, the authors of [10] present Butterfly matrices, replacing specific transformations by universal building blocks called Butterfly Factors. These $O(logN)$ Butterfly Factors, each consisting of $O(N)$ nonzero entries are multiplied, resulting in an $O(NlogN)$ algorithm as a replacement for the $O(N^2)$ algorithm of dense MM [69]. In doing so, both the execution time, by having a more efficient algorithm, and the memory footprint, by applying sparsification, are reduced. By making the nonzero entries of the Butterfly Factors learnable, a variety of discrete transforms can be represented by the Butterfly Matrix as depicted in Fig.2.5.

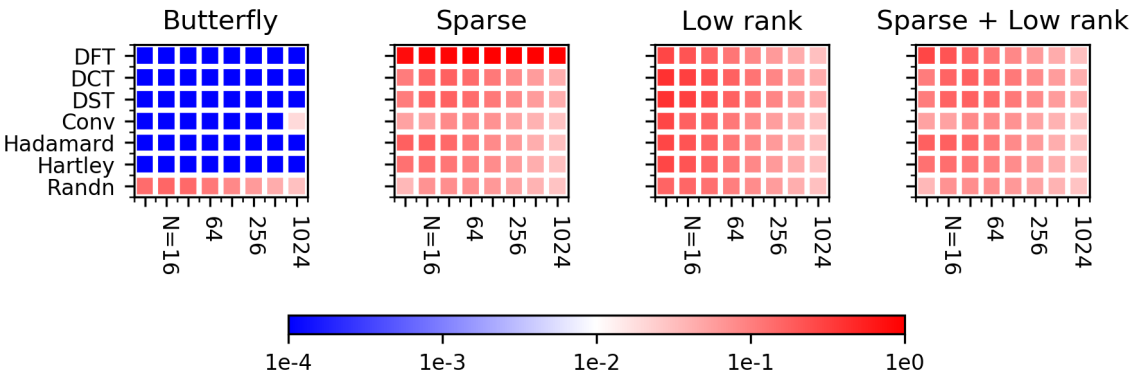


Figure 2.5: RMSE of learning fast algorithms for common transforms, with early stopping when RMSE is below 1e-4. (Blue is better and red is worse.) [10].

Compared to the other structured approaches presented in Fig.2.5, `Butterfly` is able to recover a variety of fast algorithms while consisting of universal building blocks. Fig.2.6 shows an example Butterfly Factorization for the FFT. Generally, this factorization is inspired by the Cooley-Tukey FFT algorithm. Both approaches apply the same divide-and-conquer strategy to generate the Butterfly Factors. Equation 2.1 shows the Cooley-Tukey

FFT algorithm from Fig.2.6 with matrix representation. Equation 2.2 shows the Butterfly Factorization with matrix representation.

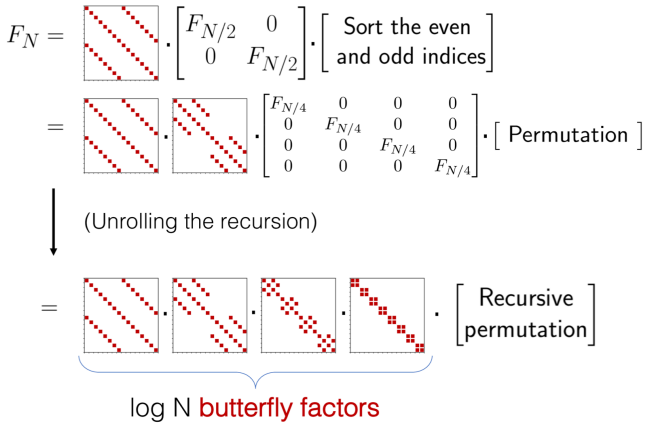


Figure 2.6: Expressing an input matrix as the product of $\log N$ Butterfly Factors. This example is for an FFT but also applies to the compression of structured matrices in neural networks [69].

$$F_N = \begin{bmatrix} I_{N/2} & \Omega_{N/2} \\ I_{N/2} & -\Omega_{N/2} \end{bmatrix} \begin{bmatrix} F_{N/2} & 0 \\ 0 & F_{N/2} \end{bmatrix} \begin{bmatrix} \text{Sort the even} \\ \text{and odd indices} \end{bmatrix} \tag{2.1}$$

$$T_N = \begin{bmatrix} D_1 & D_2 \\ D_3 & D_4 \end{bmatrix} \begin{bmatrix} T_{N/2} & 0 \\ 0 & T_{N/2} \end{bmatrix} \begin{bmatrix} \text{Separate into two halves} \\ \text{by some permutation} \end{bmatrix} \tag{2.2}$$

Equation 2.1 is a special case of 2.2 where $D_1 = D_3 = I_{N/2}$, $D_2 = \Omega_{N/2}$, $D_4 = -\Omega_{N/2}$, $F_{N/2} = T_{N/2}$ and the permutation is the separation of even and odd indices. This translates to every structured matrix being able to be decomposed into $logN$ Butterfly Factors in which the sparsity comes from the FFT factorization as shown in Fig.2.6.
Unrolling 2.2 results in the Butterfly Factorization:

$$T_N = B^{(N)} P^{(N)} \tag{2.3}$$

with $B^{(N)}$ being a Butterfly Matrix and $P^{(N)}$ being a permutation.
The authors from [10] also provide an implementation written in PyTorch with a CUDA interface based on equation 2.3. This code can be used as a drop-in replacement for linear layers.

Pixelated Butterfly

`Pixelated Butterfly`, presented in [6], is based on the Butterfly Matrices with additional efficiency improvements, namely Flat Block Butterfly, and additional low-rank terms. Flat Block Butterfly consists of two additions to the Butterfly Factorization: Flat Butterfly and Block Butterfly. Flat Butterfly approximates the products of Butterfly Factors by a sum with residual connections, enabling easier parallelization. Block Butterfly takes into account the block data access of a GPU by aligning the Butterfly Factors, thereby reducing memory accesses. Fig.2.7 represents Flat Block Butterfly with its components visually. To allow capturing local and global information, low-rank terms are added to the Flat Block Butterfly.

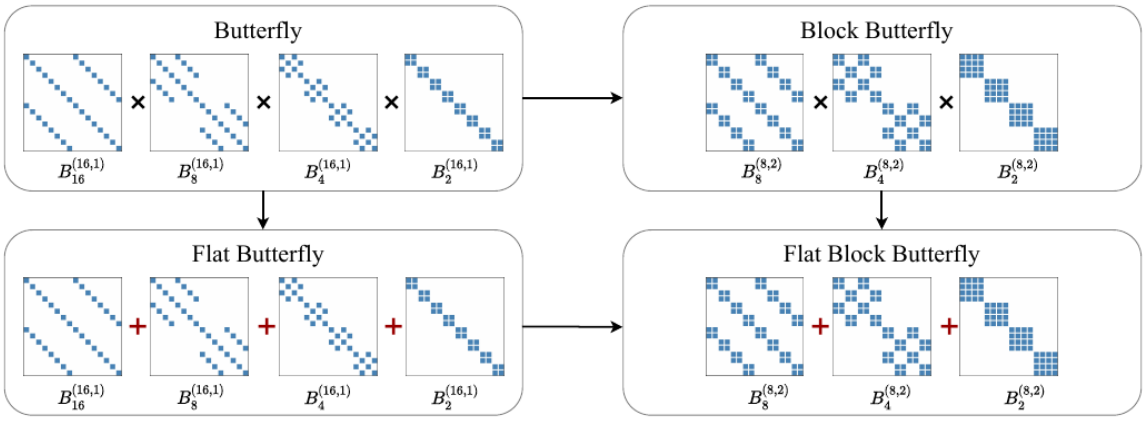


Figure 2.7: Flat Block Butterfly Creation by block-aligning the Butterfly Factors (Block Butterfly) and approximating the products of Butterfly Factors by sums (Flat Butterfly) [6].

In contrast to the `Butterfly` implementation, `Pixelated Butterfly` [6] relies on either Triton or HuggingFace as a backend. We restrict ourselves to plain PyTorch due to performance implications and the absence of some frameworks for the IPU. To overcome this issue, we revert to the implementation from [2]. In contrast to `Butterfly` which, like the linear layer, needs only the dimensions as input parameters, `Pixelated Butterfly` adds additional parameters for configuration. These are the size for the low-rank decomposition, the block size and the butterfly size, leading to different memory footprints and execution times.

3 Related Work

Earlier work in the context of the IPU mostly deals with accelerating various AI/ML workloads. Vaswani et al. [71] present Trusted Extensions, a set of experimental hardware extensions to allow running AI workloads with strong confidentiality and integrity guarantees. Kacher et al. [41] evaluate the performance of the first generation GC2 IPU for deep neural networks for inference. Sumeet et al. [65] analyze the performance of a M2000 IPU-Machine consisting of four second generation GC200 IPUs for Text Detection Applications.

Luow and McIntosh-Smith [49] use the first generation GC2 IPU for stencil computations on structured grids and characterize the performance with STREAM memory benchmark results and a roofline model. They inspect POPLAR and reported sufficient programmability to implement these HPC problems, and achieve performance compared to that of modern GPUs.

Jia et al. [39] dissect the IPU hardware with different microbenchmarks. They examine a IPU-Pod16 system which consists of 16 first generation GC2 IPUs in terms of computational performance, peak bandwidth (inter and intra IPU) and other metrics. They also reported that, in single precision, a single IPU processor outperforms a V100 by factor two (31.1 TFLOPs for the IPU and 15.7 TFLOPs for the GPU).
We extend their findings by dissecting the second generation GC200 IPU with the help of the microbenchmarks from [39]. We run the dense squared MM from section 5.1, evaluate the inter-Tile bandwidth from section 4.1.5 and compare against an A30 GPU. In addition to the dense squared MM, we add results from both dense skewed MM and sparse MM and analyze them. We include results with activated and deactivated second generation Tensor Cores of the A30 GPU to cover the equivalent for the AMP unit of the GC200 IPU. To evaluate programmability and performance, we add hand-written code to both processing units, compare in terms of performance and analyze the results.

Due to the limited memory capacity on the IPU and the different memory architecture, reducing the memory requirements for a given workload is desirable. Sparsification is a common approach to achieve this and not included in any of the previously mentioned works.

Dao et al. [10] present `Butterfly`, a parametrization of divide-and-conquer methods, capable of representing various structured transforms by learning the weights of Butterfly Factors. They claim that `Butterfly` can be easily incorporated as a efficient and compressible replacement for `torch.nn.Linear` in PyTorch.
Chen et al. [6] present `Pixelated Butterfly`, which, on top of `Butterfly`, incorporates properties to adapt to the GPU architecture. They show that the performance of `Butterfly` can be increased with Flat Block Butterfly and low-rank terms.
We evaluate their findings by conducting experiments of [10] on the second generation GC200 IPU and the A30 GPU, with the Tensor Cores turned off and on. We add additional scenarios, layers (including `Pixelated Butterfly`) and performance metrics to their experiments.

Dao et al. [8] point out the problems of `Butterfly` and `Pixelated Butterfly` being hardware-inefficient and less expressive. To address these issues, they propose Monarch Matrices, which incorporate the properties of `Butterfly` by relying on Butterfly Factorization and yield up to $2\times$ speedup compared to dense matrix multiply due to an optimized batch MM.

Lin et al. [48] introduces Deformable Butterfly, an extension of the previously presented `Butterfly`, overcoming the restriction of powers-of-two construction. This makes it possible to represent any input-output dimension.

Dao et al. [11] present `Kaleidoscope` Matrices which are capable to capture any structured matrix with near-optimal space and time complexity. They define the `Kaleidoscope` hierarchy using the Butterfly matrices from [10]. This is realized by multiplying a Butterfly Matrix with the (conjugate) transpose of another Butterfly Matrix.

The findings of the works presented in this chapter demonstrate that the IPU can compete with a comparable GPU in terms of performance and that `Butterfly` and its variants perform favorably compared to other structured sparse approaches. `Butterfly` has not been studied on sparse accelerators such as IPUs. Also, there is currently no work dissecting the second generation GC200 IPU. Therefore, this work focuses on these two aspects.

4 Performance Evaluation of the IPU

4.1 Methodology

This section describes the experimental setup of this work, which consists of an IPU-Machine for performance benchmarking and a GPU for comparison.
Fig.A.1 in the Appendix shows the M2000 IPU-Machine used in this work. It consists of four, second generation, GC200 IPUs, improving overall memory capacity and the number of cores compared to the first generation GC2 IPU [32]. As this type of hardware is quite novel with the GC2 being released in 2018 [68] and to allow a fair comparison to the GPU, we restrict ourselves to single processing units. Future work will include multiple IPUs as scaling has been proven viable by works such as [4]. Chapter 2.1 explained the background why IPUs can be scaled easily.
Fig.4.1 shows the GC200 IPU with its main hardware properties and Table 4.1 shows how they compare against the most recent IPU called Bow, the PCIe variant of the GC200 IPU called C600 IPU and the A30 GPU.

Chip	A30	GC200	Bow	C600
Number of cores	3584 [3]	1472 [32]	1472 [30]	1472 [31]
Number of threads	114 688 [56]	8832 [32]	8832 [30]	8832 [31]
Total SRAM	34.75 MB [3]	900 MB [32]	900 MB [30]	900 MB [31]
Total DRAM	24 GB [57]	64 GB [32]	64 GB [30]	0 GB
Memory Bandwidth	933 GB s^{-1} [57]	47.5 TB s^{-1} [32]	65.4 TB s^{-1} [30]	53 TB s^{-1}
Inter-Chip Bandwidth	200 GB s^{-1} [57]	320 GB s^{-1} [32]	320 GB s^{-1} [30]	128 GB s^{-1} [31]
FP32 peak compute	10.3 TFLOPs [57]	62.5 TFLOPs [39]	87.25 TFLOPs [30]	70 TFLOPs [31]
TF32 peak compute	82 TFLOPs [57]	-	-	-
Clock frequency	1.44 GHz [3]	1.33 GHz [4]	1.85 GHz [53]	1.5 GHz [31]
Power Consumption	165 W [57]	150 W [4]	126 W [53]	185 W [31]

Table 4.1: Comparison of processors

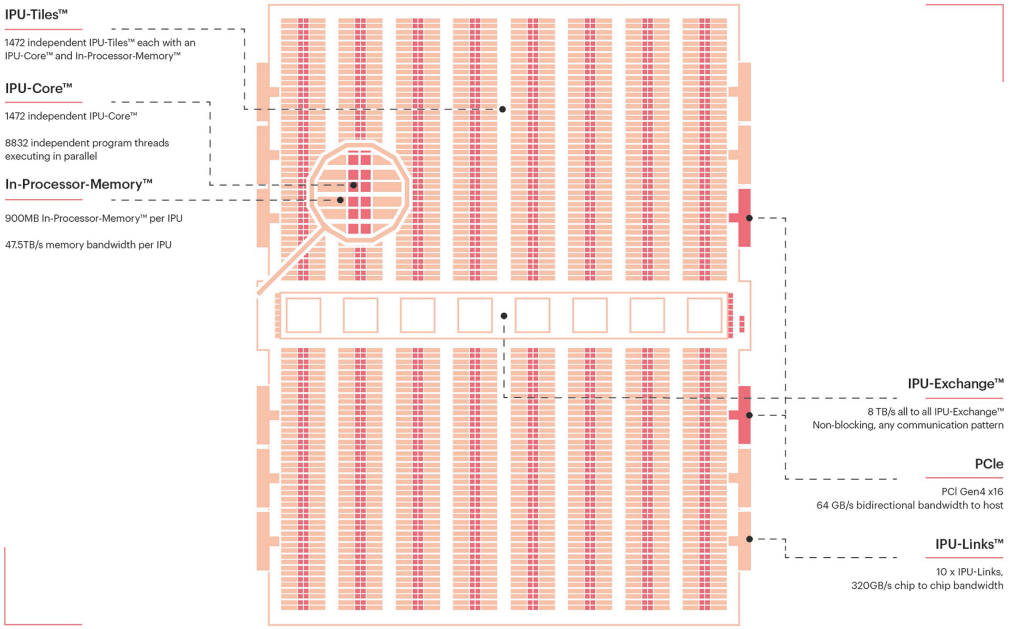


Figure 4.1: GC200 IPU Architecture [32]

4.2 Bandwidth Analysis IPU

Due to the different memory architectures, the memory capacities available for a thread are different between the IPU and the GPU. While a thread on the GPU can access GB of data, a thread on the IPU can only access `memory_capacity` divided by `number_of_cores` which translates to $611\,\text{kB}$ on the GC200 IPU [25]. Therefore, data exchanges between IPU-Tiles are much more apparent than within the GPU.
To examine how data movement affects the performance of the IPU, we analyze the memory bandwidth and memory access costs. This chapter is based on the work of [39], chapter 4.1.5 where the authors present microbenchmarks for the first generation GC2 IPU. We choose the same IPU-Tile pair as in [39] to verify the results for the GC200 IPU. The results are depicted in Fig.4.2.

These results reveal that the findings of [39] with regard to the inter-Tile bandwidth are applicable to the GC200 IPU. In case the data does not fit in the In-Processor Memory of a given IPU-Tile, it is irrelevant where the data is stored as long as it fits on the entire processing unit. This was to be expected because the IPU relies on an all-to-all, non-

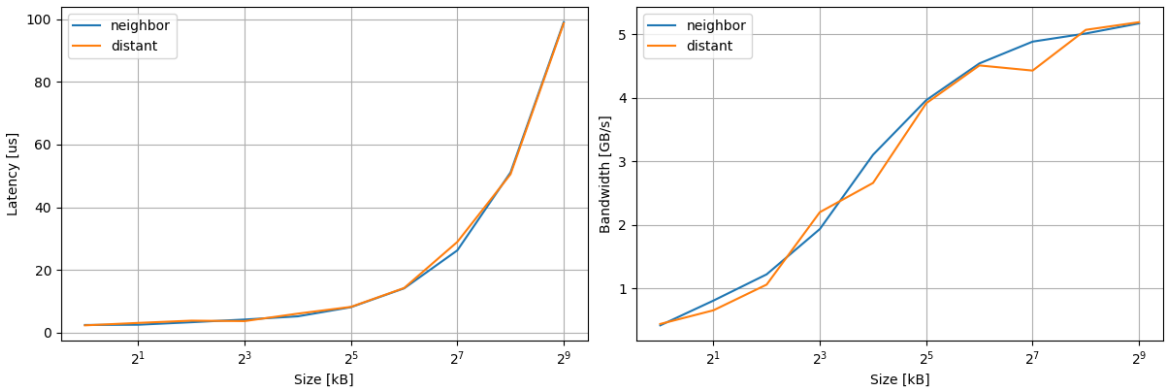


Figure 4.2: Latency and Bandwidth within a GC200 IPU with different physical proximity. The pair of neighboring IPU-Tiles is (0,1) and the chosen pair of distant IPU-Tile is (0,644)

blocking interconnect that exchanges data in every BSP superstep, see Fig.A.2 and [26]. For larger data sizes, the bandwidth increases until it reaches a plateau. This is similar to other processing units because the communication setup overhead is almost independent to the data size, meaning that greater data sizes yield a smaller relative overhead [39]. The experiments from Fig.4.2 are performed on a congestion-free network, meaning that no other IPU-Tile wants to access data simultaneously. For the highly congested scenario, we do not conduct any experiments. As the results in Fig.4.2 verify chapter 4.1.5 of [39], we take the insights from [39], chapter 4.1.6, as applicable for the GC200 IPU. There, congestion only degrades the bandwidth by $1.02\times$.

The results from Fig.4.2 allow to determine that sparsification is suitable for the IPU. This is in particular interesting for evaluating `Pixelated Butterfly` for the IPU as one of its components (Block Butterfly) relies on block data access known from GPUs. In contrast to a GPU, where the performance decreases for sparse matrices as some of the threads in a thread warp idle while the rest executes, the IPU-Tiles can execute independently. Only the data exchange costs have to be included as an IPU-Core can only access $611\,\text{kB}$ of In-Processor Memory. As the exchange phase is always included in a BSP superstep on the IPU, the performance penalty is small compared to idling threads on the GPU.

4.3 Experimental Overview: Matrix Matrix Multiplication (MM)

A prime example for dissecting the performance of a given hardware is the MM as these calculations are often used in various scientific fields. In [39], MM problems on the IPU were only analyzed with squared dense matrices. In this work, we run the MM benchmarks from [39], chapter five, and extend them via two metrics, skewness and sparsity, to cover more real-world applications. Three different scenarios were picked to analyze the underlying behavior on GPU und IPU:

1. Dense Squared $\times$ Dense Squared MM
2. Dense Skewed $\times$ Dense Skewed MM
3. Sparse Squared $\times$ Dense Squared MM

The MM in this work is defined as follows:

$$A(m \times n) \times B(n \times k) = C(m \times k) \tag{4.1}$$

Skewness

In real-world applications such as ML, (weight) matrices are often non-squared. [64] gives an overview of standard matrix dimensions for a variety of ML workloads. Thus, we extend squared dense MM benchmarks with skewed dense MM for both, IPU and GPU. Skewness is defined by the dimensions of the input matrices A and B:

$$s_k := \frac{m}{n} \tag{4.2}$$

Fig.4.3 depicts left-skewed (low s_k), squared (s_k of 1) and right-skewed (high s_k) matrices schematically.

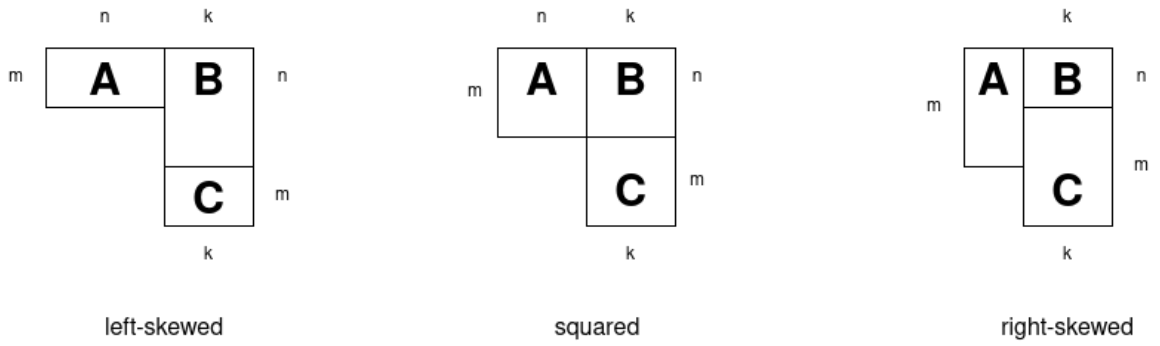


Figure 4.3: Schematic representation of left-skewed, squared and right-skewed matrices

Sparsity

As real world data is often sparse (e.g. for BFS [4]) and many compression techniques such as Butterfly Approximations rely on sparse data [10] [6], see Fig.2.6 and Fig.2.7, we cover this variant of MM in this work. In addition to dense MM, sparse $\times$ dense MM with squared input dimensions is also covered in this work. Sparsity is defined as follows:

$$s_p := \frac{n_z}{n_t} \tag{4.3}$$

In this equation, n_z is the number of zeros and n_t is the total number of elements in the matrix.
The following chapters examine the MM for the IPU and the GPU with regard to complexity in the implementation, the performance and memory usage. All upcoming sections share two common performance metrics, problem size and FLOPs.
The problem size is derived from the matrix dimensions and defined as follows:

$$S(m,n,k) = (m \times n + n \times k + m \times k) \times sizeof(datatype) \tag{4.4}$$

The FLOPs are calculated as follows [37]:

$$FLOPs = \frac{(2 \times m \times n \times k - m \times k)}{t_{exec}} \tag{4.5}$$

4.4 Dense MM

Five different implementations on the IPU and four different implementations on the GPU are examined and compared. For verification purposes, the result matrix C is compared against a CPU implementation. Table 4.2 provides an overview of the GPU and IPU implementations, categorizing them based on abstraction level (ML or Low-Level) and distinguishing between hand-written and library implementations.

Name	Abstraction	Library/Hand-Written
ipu-naive	Low-Level	Hand-Written
ipu-blocked	Low-Level	Hand-Written
ipu-poplin1	Low-Level	Library
ipu-poplin2	Low-Level	Library
ipu-poptorch	Machine Learning	Library
gpu-blocked	Low-Level	Hand-Written
gpu-shared	Low-Level	Hand-Written
gpu-cublas	Low-Level	Library
gpu-pytorch	Machine Learning	Library

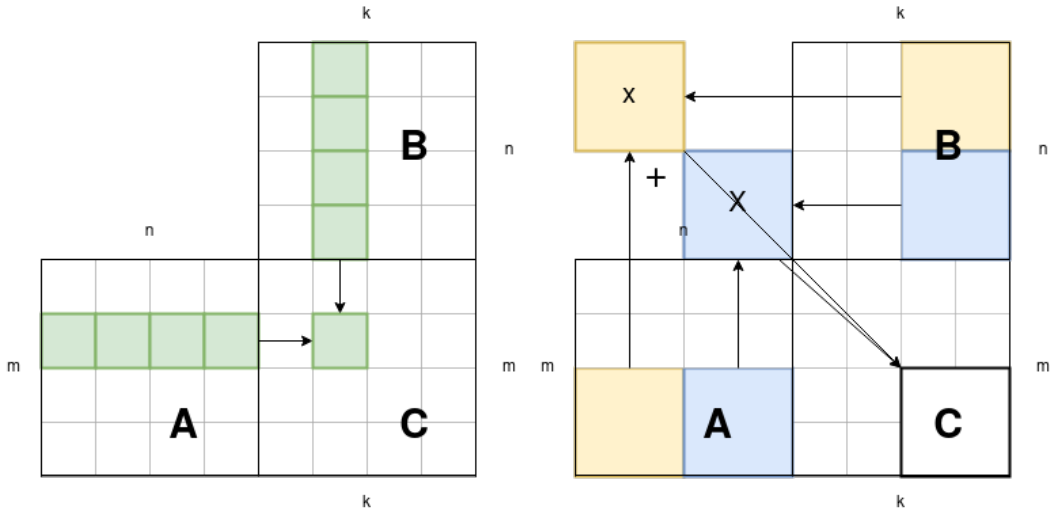
Table 4.2: DSGMM implementations

In the following, the implementations are explained in more detail:

Implementation Details

`ipu-naive`: Each of the IPU-Tiles computes a single element of C. Therefore, each IPU-Tile requires one row of A and the respective column of B. Fig.4.4 shows a graphical representation of the approach. A Vertex to compute each element of C is created. Additionally, an entire row of A and column of B are mapped on the same IPU-Tile. As each IPU-Tile has $611\,\text{kB}$ of In-Processor Memory, this approach does not seem promising with regard to memory consumption, scalability and data reuse.

`ipu-blocked`: The input matrices A and B are partitioned into smaller squared sub-matrices called blocks. Fig.4.4 shows a graphical representation of the approach. In contrast to the ipu-naive implementation, Vertices are not generated for each output element. Instead, $m \div blockSize$ Vertices, multiplying blocks, are generated. Each of those Vertices creates a temporary result that is added to the corresponding block in C by another Vertex. For a squared matrix, $(m \div blockSize)$ blocks need to be calculated. For each block, $(m \div blockSize)$ multiplication and addition Vertices are created. For skewed



(a) naive approach (b) blocked approach

Figure 4.4: Hand-written MM implementations

matrices, the remaining part $(k \bmod blockSize)$ or $(m \bmod blockSize)$ is added to the last Tensor holding a submatrix of C. This greatly increases memory reuse and decreases memory consumption on each individual IPU-Tile as the work is separated more equally than for ipu-naive. Therefore, scaling is more feasible with this implementation.

`ipu-poplin1`: PopLIN is the PopLIBS library by Graphcore for dense linear algebra functions such as MM and convolutions [18]. In this approach, the input Tensors are mapped linearly to all IPU-Tiles via `mapTensorLinearly`. The Vertices and the output Tensor for the MM are added using PopLIN via `matMul`. As with all PopLibs functions, this approach does not allow the use of multiple IPUs due to manufacturer specifications.

`ipu-poplin2`: In this approach, all Vertices and Tensors are mapped implicitly using PopLIN. As with all PopLibs functions, this approach does not allow to use multiple IPUs due to manufacturer specifications. MM execution is realized via `matMul` and the mapping of Tensors via `createMatMulInputLHS` and `createMatMulInputRHS`.

`gpu-naive`: This approach is very similar to ipu-blocked as multiple thread blocks compute the result matrix simultaneously and execute independently. This implementation is based on the unoptimized MM of [59], section 13.2.3.2.

`gpu-shared`: In this approach, the data is first loaded into shared memory to speedup the data accesses. Apart from that, the gpu-naive and the gpu-shared approach are identical.

This implementation is based on the MM using shared memory to improve global memory load efficiency of [59], section 13.2.3.2.

`gpu-cublas`: Here, the linear algebra library implemented by NVIDIA is used for the execution. Data allocation and movement is implemented manually. MM execution is realized via `cublasSgemm()`.

Dense Squared × Dense Squared MM

This chapter provides a comparison between different MM implementations on the IPU and the GPU with FP32 values and squared matrices.
Fig.4.5 shows the performance of the MM implementations defined in 4.2. For the GPU, there are two horizontal lines, one for the theoretical peak FP32 performance of $10\,300$ GFLOPs and one for the theoretical peak TF32 performance of $82\,000$ GFLOPs [57]. For the IPU, the horizontal line represents the peak theoretical FP32 performance of $62\,500$ GFLOPs [17].

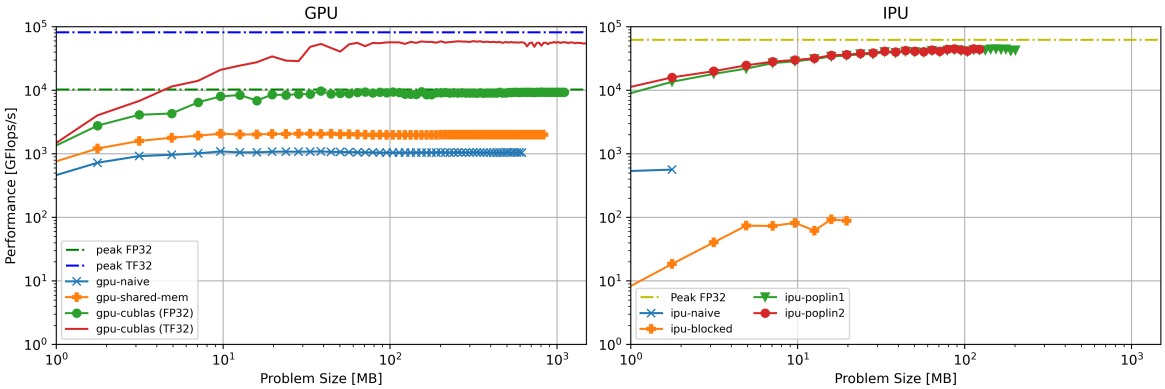


Figure 4.5: Squared MM GPU vs IPU low-level

On the GPU, gpu-naive achieved 1091.67 GFLOPs, gpu-shared 2076.31 GFLOPs, gpu-cublas (FP32) 9722.19 GFLOPs and gpu-cublas (TF32) $59\,312.24$ GFLOPs at peak. In contrast to gpu-cublas, which is able to take advantage of the Tensor Cores that are dedicated to MM operations, running hand-written implementations with the Tensor Cores is not possible.
On the IPU, ipu-naive achieved 525.59 GFLOPs, ipu-blocked 93.41 GFLOPs, ipu-poplin1 $44\,219.83$ GFLOPs and ipu-poplin2 $44\,239.83$ GFLOPs at peak.
The GPU almost reaches its theoretical peak performance with 9.7 TFLOPs ($94.4\,\%$) without the use of the Tensor Cores. On the contrary, with activated Tensor Cores, the GPU

achieves 59.3 TFLOPs, $27.7\,\%$ less than the theoretical peak TF32 performance. The IPU falls short of the theoretical peak FP32 of 62.5 TFLOPs by $29.8\,\%$, achieving 44.2 TFLOPs. The IPU performance results for the IPU were verified by Graphcore. Despite this fact, the IPU still surpasses the GPU performance as long as the matrices can fit into the In-Processor Memory of the IPU and as long as the Tensor Cores are turned off on the GPU. A known drawback of the IPU is the lack of memory as shown in Table 4.1. As in Fig.4.5 presented, the GPU can handle larger data sizes due to the bigger amount of memory available per thread.
Generally, it is not feasible to implement a MM by hand on both platforms as reaching the respective theoretical performance is not possible without in-depth optimizations. Optimizing implementations for MM will be a focus of further research to overcome the PopLIBS limitation of using a single IPU. We assume that this library restriction is due to large matrices being less present for most ML applications.
The performance difference between the hand-written and library code differs greatly between the platforms. By utilizing Shared Memory (gpu-shared), $21.3\,\%$ of the peak gpu-cublas (FP32) performance is reached. For the IPU, with ipu-blocked it is less than $1\,\%$ of ipu-poplin2 peak performance. For ipu-naive, it is $1.2\,\%$ of the aforementioned peak performance. This shows that reaching to the peak performance of the IPU with hand-written code is much harder than for the GPU. This is due to the novelty of the IPU architecture and the Programming Model. In contrast to a GPU, where most performance penalties are well-known, this is not the case for the IPU. As the code for PopLIBS is publicly available, a thorough analysis will be included in future work.
With both processors focusing on AI workloads, it is of great interest to compare the previous results with the performance of the PyTorch / PopTorch layer that is equivalent to a MM, `torch.nn.Linear`. The difference to an actual MM is that the input matrices consist of the network input and the learned weights for `torch.nn.Linear`. Fig.4.6 shows the performance on both systems with their low-level and high-level library functions.

On the GPU, both abstraction layers benefit from the Tensor Cores. We run the experiments with and without the Tensor Cores in PyTorch and cuBLAS to evaluate possible differences. On the GPU, the performance does not differ noticeably between PyTorch and cuBLAS, due to the dynamic graph compilation having less overhead than the static graph compilation. Additionally, PyTorch lets the data reside on the GPU when defining the Tensors. Therefore, data movement costs are not included in the benchmark, similar to the cuBLAS benchmark.
This does not hold true for the IPU as there is a noticeable performance decrease between

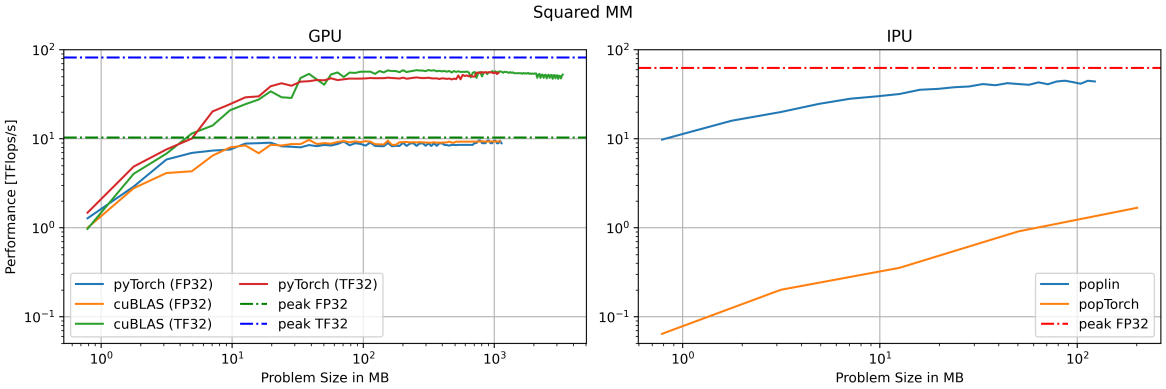


Figure 4.6: Squared MM GPU vs IPU high-level vs low-level library

PopTorch and PopLIN. In contrast to PyTorch, where we could define the location of data explicitly, the PopTorch benchmark always includes data copies to and from the host in the time measurement. This is due to PopTorch not allowing to define multiple `Engine:run` calls explicitly like it is the case in POPLAR. To verify our claims, we examined the results with the PopVision Graph Analyser profiling tool for both, ipu-poplin2 and ipu-poptorch. Fig.4.7 and Fig.4.8 show the execution traces for ipu-poplin2 and ipu-poptorch at squared matrix dimensions of $N = 2^{12}$.

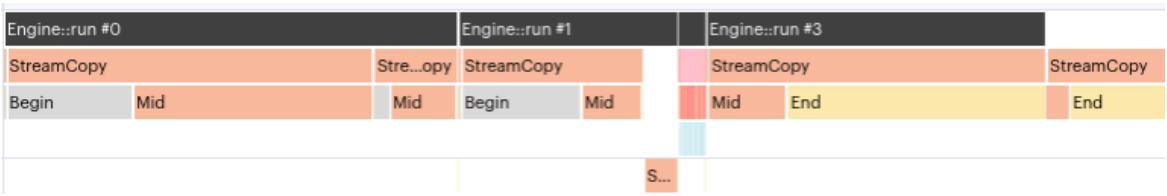


Figure 4.7: Execution Trace of ipu-poplin2 with $m = n = k = 2^{12}$ elements. Generated with PopVision Graph Analyser

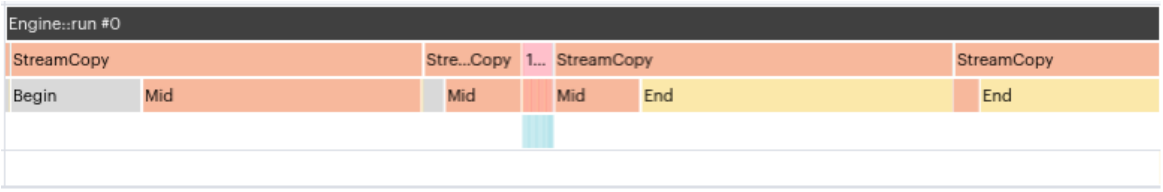


Figure 4.8: Execution Trace of ipu-poptorch with $m = n = k = 2^{12}$ elements. Generated with PopVision Graph Analyser

The two execution traces differ in the number of `Engine::run` calls. The ipu-poplin2 trace consists of four, one for each data copy of A and B from the host to the IPU, one for the execution of the MM and one for copying back the result matrix C to the host.

The ipu-poptorch trace, on the other hand, lacks multiple calls and merges every aspect that is separated in the ipu-poplin2 trace into a single `Engine::run()`. This was to be expected since PopTorch lacks the possibility to define multiple `Engine::run()` calls. By increasing the number of iterations, the proportion of the data movement is reduced but still noticeable. This, coupled with the compilation time for creating the static execution graph, explains the difference in the performance results when benchmarking and is verified by the relatively small difference in execution cycles of $70\,000$ between the two implementations, likely due to the graph compilation overhead. The difference in execution cycles was obtained with the execution traces of Fig.4.7 and Fig.4.8. Additionally, both call the function `Convolve`, as shown by the Execution Traces of both implementations. To conclude, PopTorch performs similar to PopLIN but lacks the possibility to manually define calls to `Engine:run()`, resulting in the data movement being included in the time measurement. To exclude data movements from performance measurements, one could use libpva, the PopVision Analysis Library to extract the timings of the different `Engine::run()` calls [23]. As this approach requires a profiling report for every scenario and therefore adds much overhead for performance analysis without giving additional insights, we decide to not pursue this further.
All the results for the IPU have in common that the largest problem size ($122.9\,$MB) for MM is less than $13.7\,\%$ of the total In-Processor Memory, shown in Fig.4.5. We assume that this is due to the additional generation of code to exchange the data that is needed to compute the results of the output matrix C as explained in 4.2. To validate this claim, we export data from the PopVision Graph Analyser for ipu-poplin2. The left plot in Fig.4.9 shows the number of Edges, Variables and Vertices which are generated to run the MM with squared dimensions in ipu-poplin2, the right plot shows the number of Compute Sets and the total memory allocated, including temporal and Variable data.

Every metric shown in the left plot of Fig.4.9 increases at a different rate with larger problem sizes. The increase was to be expected because of the additional load introduced by the higher matrix dimensions, although the rate differs between the metrics. The slope of the curve for the number of Edges is much higher than for the other two metrics. As Edges represent the connection between Variables and as the number of Variables increase with the input dimensions, this was to be expected since the IPU-Tiles cannot allocate sufficient temporal data, therefore exchanging data to compute the result matrix. The number of Vertices correlates with the number of Compute Sets as these represent the execution on the IPU-Tiles. The number of Compute Sets remains constant until it reaches a tipping point. We assume that this is due to the IPU not having sufficient spare memory

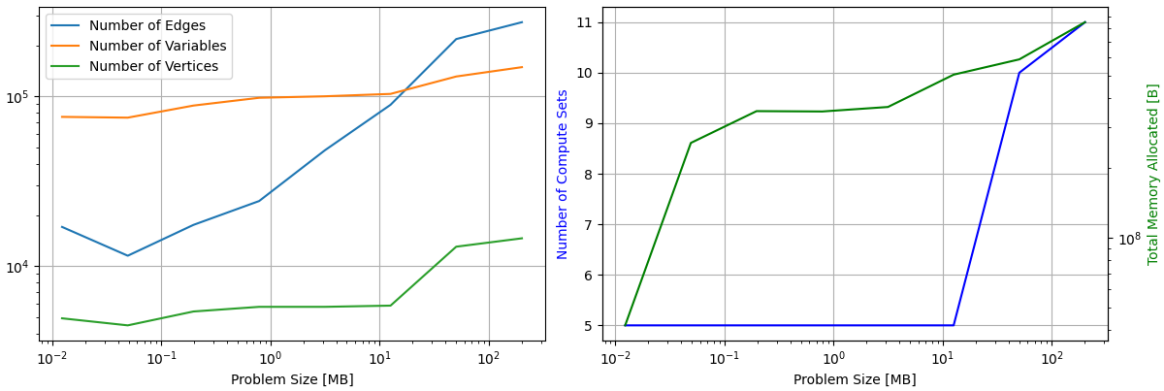


Figure 4.9: Underlying performance data of ipu-poplin2. Generated with PopVision Graph Analyser

to store the data temporally. The IPU has to run more execution steps in order to compute the result matrix C. Our assumption is also supported by the total memory allocated in the right plot of Fig.4.9 in which the total memory allocated increases more heavily than the problem size. The number of Variables increase at a constant rate as these are the allocated Tensors, representing the input and output matrices for MM.
In order to be able to calculate larger input dimensions in the MM, we need to gain insights of the additional data allocated by the compiler. To do so, we examined the total consumed memory with ipu-poplin2 at the largest problem size of $m = n = k = 2^{12}$, depicted in Fig.4.9, by using the memory report of the PopVision Graph Analyser.
Table 4.3 shows the biggest proportions of memory consumption in MB in descending order. All data except for the problem size are taken from the PopVision Graph Analyser report. Every category that takes up less than $1\,\%$ of allocated In-Processor Memory (with gaps) is omitted for simplicity. As the compiler is responsible for finding the best fitting graph for the execution, we assume that the compiler generates additional data to speedup the execution. In order to verify this, hand-written code is analyzed and compared against ipu-poplin2 as the execution is predefined. To do so, we profiled ipu-blocked with its biggest problem size, $m = n = k = 2^{10}$, and include the results in Table 4.3. If the memory requirements for the additional data can be reduced, bigger problem sizes could be processed on a single IPU.

ipu-poplin2 allocates $94.4\,\%$ of total In-Processor Memory with ipu-blocked allocating $47.1\,\%$ at its respective problem size. The difference can be explained by our restriction to power-of-two input matrix dimensions. To make the comparison between the two more meaningful, we include the ratio between the memory allocated for the specific cate-

Metric	ipu-poplin2		ipu-blocked	
Total Mem Available	900 MB		900 MB	
Problem Size	201.32 MB	$22.4\,\%_1$	12.58 MB	$2.5\,\%_1$
In-Processor Memory (with gaps)	850 MB	$94.4\,\%_1$	424 MB	$47.1\,\%_1$
In-Processor Memory (without gaps)	845 MB	$93.9\,\%_1$	424 MB	$47.1\,\%_1$
Variable	466 MB	$54.9\,\%_2$	146 MB	$34.4\,\%_2$
Internal Exchange Code	177 MB	$20.1\,\%_2$	31.7 MB	$7.5\,\%_2$
Message	27.4 MB	$3.1\,\%_2$	111 MB	$26.2\,\%_2$
Host Exchange Packet Header	19.4 MB	$2.3\,\%_2$	0.8 MB	$0.2\,\%_2$

Table 4.3: Categorization of memory allocated for ipu-blocked at $m = n = k = 2^{10}$ and ipu-poplin2 at $m = n = k = 2^{12}$
$_1$ of total memory available on the IPU
$_2$ of In-Processor Memory (with gaps)

gory to the allocated In-Processor Memory (w/ gaps). In total, $80.4\,\%$ of the total memory available is consumed by the depicted categories for ipu-poplin2. For ipu-blocked the share of these categories is $68.3\,\%$.

According to the previous discussions, the allocated Variables for the matrices take up around half of the memory for ipu-poplin2 and around a third for ipu-blocked, despite in theory, the required memory should be at least in the range of the problem size. For ipu-poplin2, the allocated memory is $2.3\times$ the problem size and for ipu-blocked $11.6\times$ the problem size. This discrepancy between the two implementations can be explained by the additional temporal blocks that have to be computed for ipu-blocked. The ratio between Variables and the problem size for a given implementation will be analyzed in future work.

Apart from the data memory allocation for Variables, the difference between the two implementations for the other categories is also of great interest. While ipu-poplin2 allocates much more memory for Internal Exchange Code than ipu-blocked, the converse is true for Messages. Messages are used to store temporal data that is exchanged internally and Internal Exchange Code are the code instructions used to move data between IPU-Tiles [35]. As both serve a similar purpose and, when added, have a similar share for both implementations, we come to the conclusion that ipu-poplin2 allows bigger problem sizes with less need for temporal data than ipu-blocked. At least with our algorithm, implementing a MM by hand to decrease memory requirements is not feasible. This aligns with section 4.2 with regard to the increase in Exchange Code and explains the difference in largest problem size.

It should also be mentioned that the memory is not distributed equally onto each IPU-Tile for ipu-poplin2. We assume that balancing the memory consumption leads to worse

performance in this workload as the matrix dimensions $m = n = k = 2^{12}$ are not integer divisible by the number of IPU-Tiles (1472). To see if this holds true, we compare the Tile map of ipu-poplin, where we place the data manually over each IPU-Tile and ipu-poplin2, where the compiler handles the data placement with the matrix dimensions mentioned before. Fig.4.10 depicts the relative memory consumption for both implementations on each IPU-Tile.

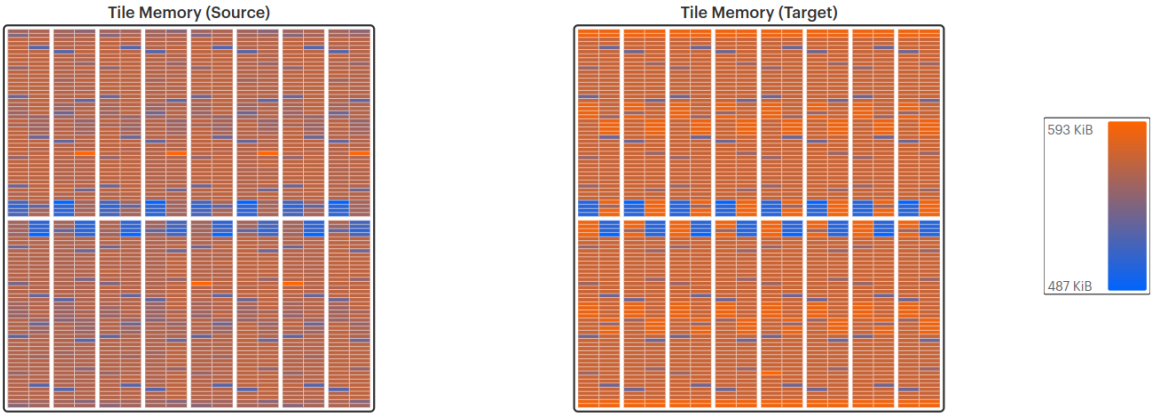


Figure 4.10: Tile Map ipu-poplin (left) and ipu-poplin2 (right) with squared dimensions, $N = 2^{12}$

For both Tile maps, the distribution of memory consumption is similar. Both have 16×4 IPU-Tiles with less memory allocated than the rest. As the remaining number of IPU-Tiles is equal to a multiple of the aforementioned number, we see our previous claim as verified. The only difference between the two Tile Maps is that for ipu-poplin1, the relative memory consumption is lower than for ipu-poplin2. We assume that this is due to the linear mapping, leading to a more uniform distribution of the data. As the difference between the two Tile maps is small, we assume that the mapping of IPU-Tiles is similar between ipu-poplin2 and ipu-poplin1, at least for this problem size. This also verifies the low difference in peak performance between the two implementations depicted in Fig.4.6.

Dense Skewed $\times$ Dense Skewed MM

All approaches of chapter 4.4 were adapted and benchmarked. The benchmarks are defined as follows: the matrix dimension k is kept constant while the skewness s_k is varied, see equation 4.2. Firstly, both platforms are compared individually with regard to their best performing implementation. As the CPU versions are used for verification purposes only,

no benchmarks are run there. After that, the best performing versions are compared to each other across IPU and GPU.

When comparing the three different versions on the GPU, the best performing one is gpu-cublas. This was to be expected since the functions are part of the heavily optimized linear algebra by the manufacturer. The behavior is independent to the given implementation. As soon as s_k reaches more extreme values, the achieved peak performance declines heavily. As gpu-cublas performs best, its performance numbers are used for the comparison against the IPU.
In Fig.4.13 and Fig.4.14, the benchmark results for ipu-blocked, ipu-poplin1 and ipu-poplin2 are depicted for the IPU. As ipu-naive is restricted to using a single IPU-Tile, the memory is a $\frac{1}{1472}$ of the total In-Processor Memory. While running the benchmarks, no $k \geq 2^6$ was able to be performed with this implementation due to the IPU-Tile not offering sufficient memory. Therefore, no plot is shown for ipu-naive.
In terms of the peak performance of ipu-poplin1 and ipu-poplin2, both achieve similar peak performance results of $42\,000$ GFLOPs while ipu-blocked performed worse with 62 GFLOPs. Similar to the GPU, the best performing implementation for the IPU is the provided library function. This was to be expected since these implementations are heavily optimized due to their high occurrence in ML applications.
For the GPU, extreme values of s_k result in significantly lower performance.
For the IPU, this does not hold true for every implementation. ipu-blocked performs almost equally well for different values of s_k. We assume that this is due to the division of the work into equally spaced blocks (except for the rest). For ipu-poplin1 and ipu-poplin2, the behavior is more similar to the GPU. For more heavily skewed matrices, the performance decreases on both sides. In contrast to the GPU and ipu-poplin1, the performance drops are not as symmetrical for ipu-poplin2. For the right-skewed MM, the performance drop is much more severe than for the left-skewed MM. We assume that this is due to the compiler having more problems with finding an optimal distribution of Tensors for ipu-poplin2. Overall, the IPU seems to have less issues and therefore less performance loss when dealing with skewed MM.
In order to analyze the performance decreases for more skewed matrices on both systems, the respective profiling tools (NSight Compute for the GPU and PopVision Graph Analyser for the IPU) are used and the different metrics evaluated. Fig.4.15 shows the benchmark data for the gpu-shared-mem implementation. The reason why we included gpu-shared-mem in addition to gpu-cublas is that the latter calls different functions and libraries that add more dimensions to the problem. The kernels for gpu-cublas are included in the

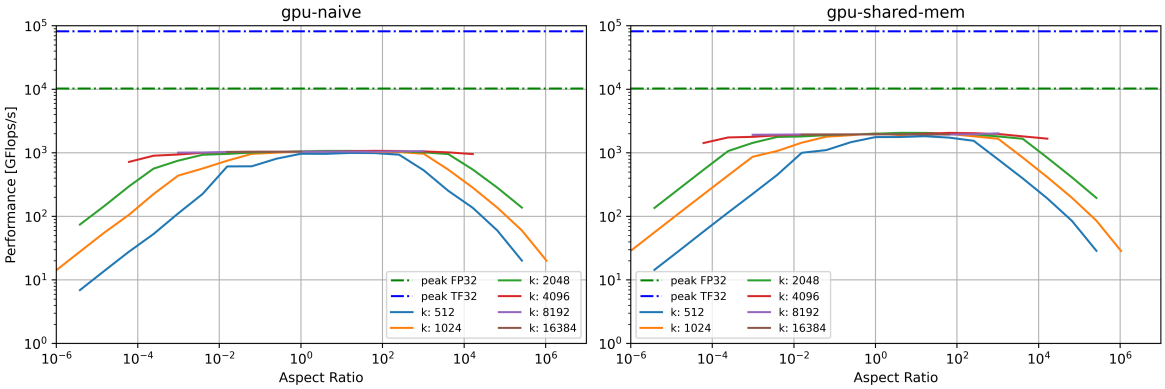


Figure 4.11: Skewed MM GPU with gpu-naive (left) and gpu-shared-mem (right)

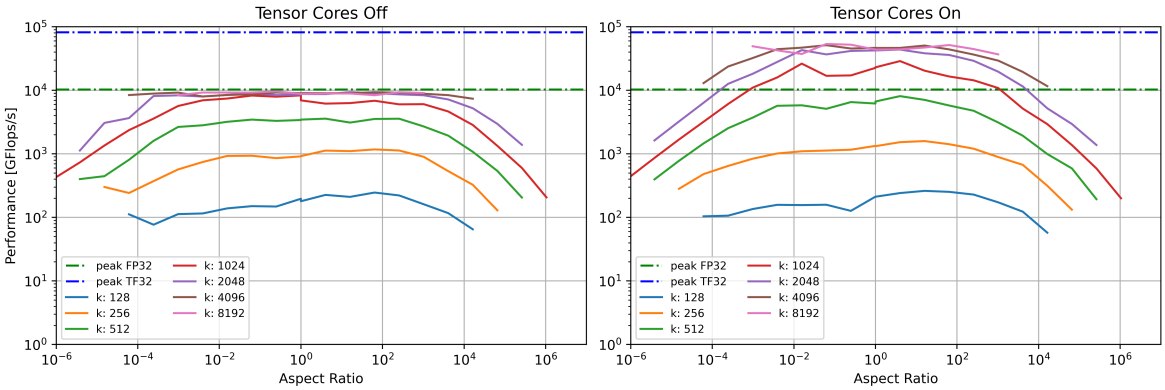


Figure 4.12: Skewed MM GPU with gpu-cublas FP32 (left) and gpu-cublas TF32 (right)

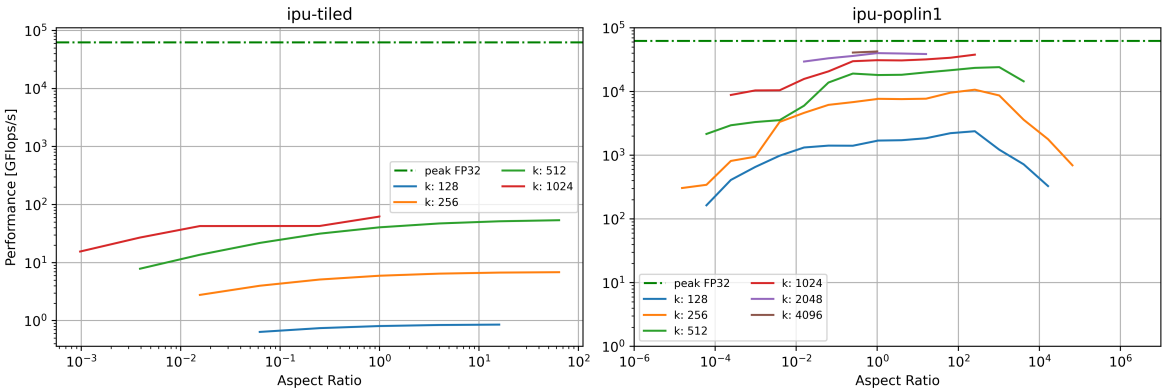


Figure 4.13: Skewed MM IPU with ipu-blocked (left) and ipu-poplin1 (right)

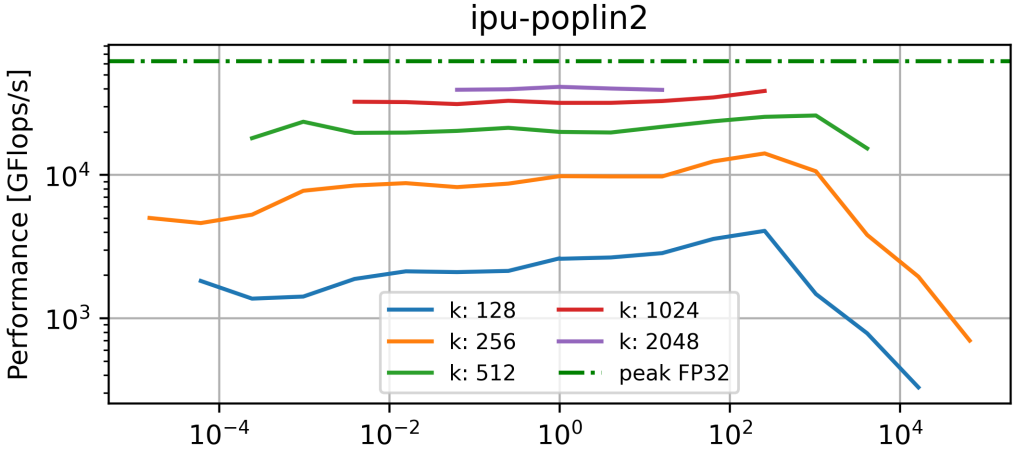


Figure 4.14: Skewed MM IPU with ipu-poplin2

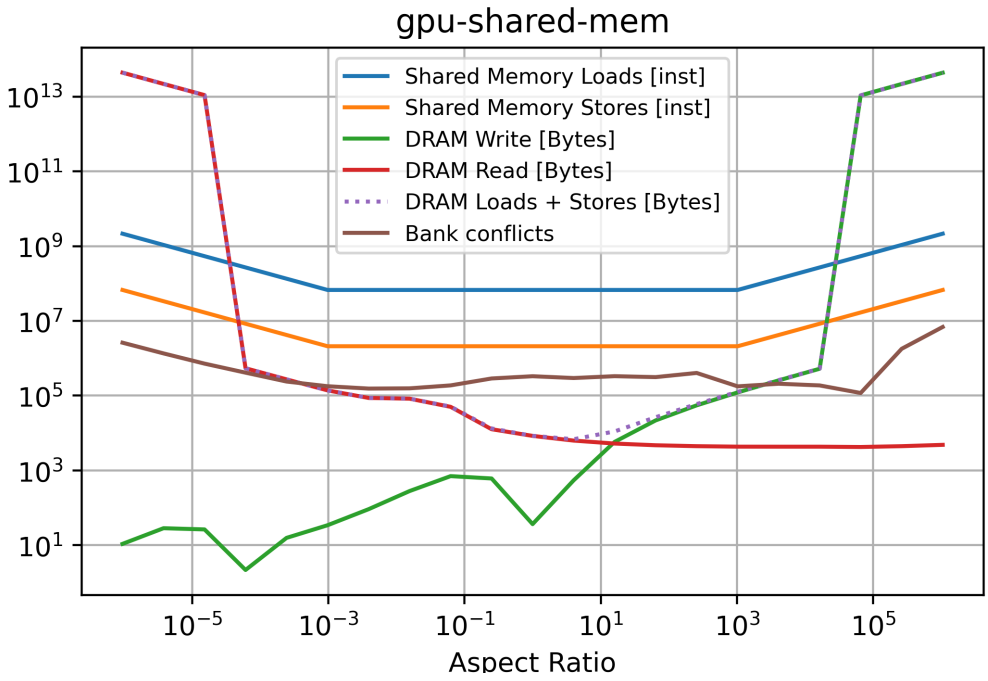


Figure 4.15: Skewed MM GPU with gpu-shared-mem

Appendix, Table A.3 and Table A.2 of this work.

As shown in Fig.4.15, the strongest change generated by more extreme values of s_k is the data size being transferred from and to DRAM, in other words to Global Memory. For low values of s_k, much more data is read from DRAM than written to it. This is due to the GPU not being able to distribute the data uniformly over the SMs, leading to less parallel slackness and therefore less performance. With high values for s_k, the converse is true: The data size being written to DRAM takes up much more than the DRAM reads. This is due to the GPU separating the problem into multiple smaller steps instead of calculating everything in a single kernel launch, as underlined by the number of kernel calls in Table A.3 and Table A.2. In contrast to our expectations, bank conflicts and uncoalesced data accesses, which are often responsible for performance degradation, do not increase as noticeable as the data movements with higher s_k.
The left plot from Fig.4.17 shows the number of Edges, Variables and Vertices from ipu-poplin2 while the right one shows the number of Compute Sets. We choose to analyse this implementation as this revealed the best performance. In contrast to the GPU, the behavior is different between the implementations. Similar to the results from the GPU, the IPU also calls different functions for different values of s_k. These are included in the

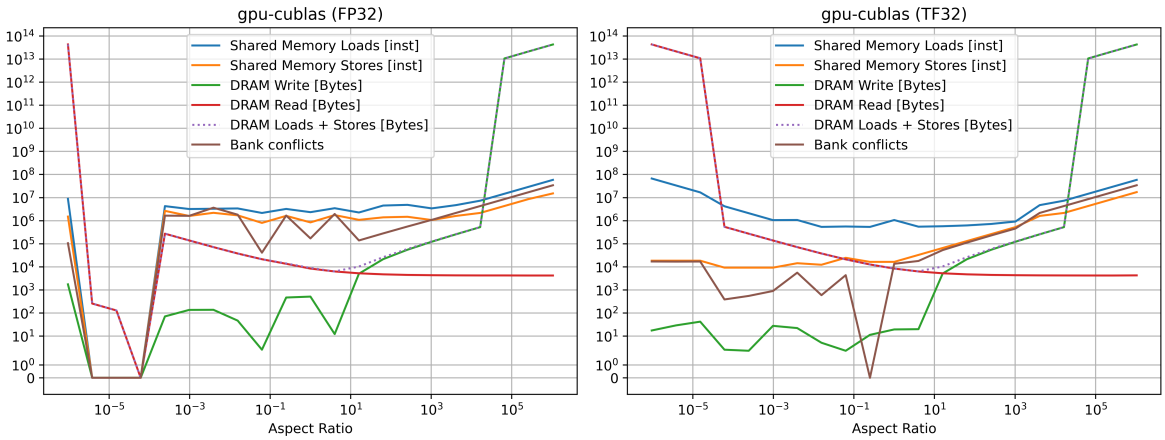


Figure 4.16: Skewed MM GPU with gpu-cublas (FP32, left) (TF32, right)

Appendix, Table A.1 of this work. With the help of Fig.4.17 and the kernel calls in the Table A.1, the drastic performance drop for the right-skewed matrices when executing ipu-poplin2 can be explained. For this scenario, the compiler creates more Compute Sets, resulting in more Edges, Variables and Vertices for the execution. We assume that this asymmetric behavior is due to less optimizations being able to be used for this matrix dimensions on the IPU, therefore, more cycles are needed to compute the result.

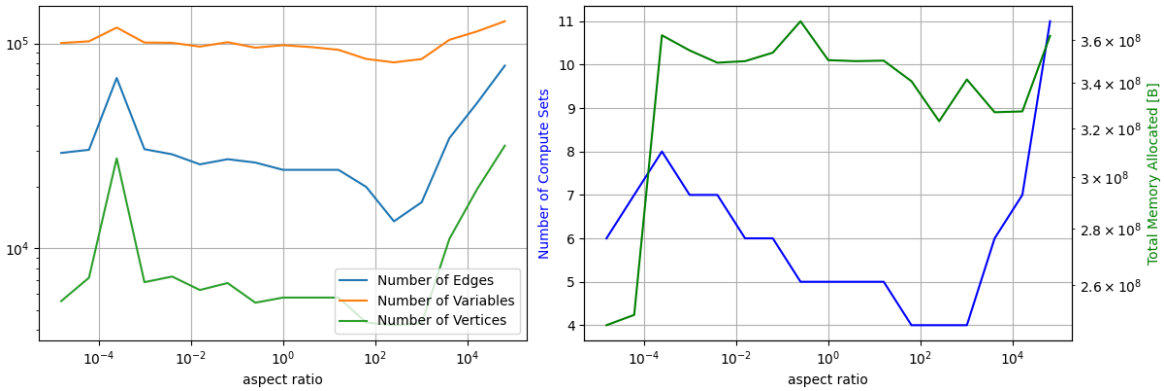


Figure 4.17: Skewed MM IPU with ipu-poplin2

4.5 Sparse MM

Various applications in scientific computing such as graph analytics or multigrid solvers rely on sparse data as well as sparsification approaches in ML such as `Butterfly` [10] and `Pixelated Butterfly` [6].

The sparse matrices can be compressed into less data intensive formats such as CSR, COO, or CSC with the goal to reduce the memory footprint [72].
This, coupled with the main focus of the IPU being AI workloads and its limited amount of memory, make it an appealing workload for performance evaluation. We expect the IPU to perform better than the GPU due to the MIMD execution allowing each IPU-Tile to operate independently from another [25] in contrast to the GPU being a block device [6]. As in previous chapters, the GPU is used for performance comparison. To ensure correctness of the results, the CPU is used to verify the results generated by the IPU and the GPU.
As PopLIN and cuBLAS performed best on their respective platforms, this chapter only examines the performance of the respective sparse MM libraries, PopSPARSE for the IPU and cuSPARSE for the GPU. In the following, a PopSPARSE and two cuSPARSE implementations are compared:

1. ipu-popsparse
2. gpu-cusparse-coo
3. gpu-cusparse-csr

Implementation Details

For both, IPU and GPU, the algorithms rely on compressing the input matrix A. To achieve this, we rely on the sparse data formats COO and CSR. These will be explained in short:
COO - Each nonzero element is stored with its row and column indices. This results in three arrays of the same lengths (row indices, column indices and values), equal to the number of nonzeros [47].
CSR - This compression format is similar to COO with the addition of compressing the row indices. Instead of adding an entry to the column indices array for each nonzero element, the number of nonzeros above a given row are stored. Thus, the size of this array has a length of the number of nonzeros + 1 [47].

`ipu-popsparse`: PopSPARSE is the library for sparse Tensor operations by Graphcore. As with PopLIN, the library functions provided are currently limited to a single IPU [22]. As in ipu-poplin2 for the dense $\times$ dense MM, the compiler is responsible for creating the two input and the output Tensors. Additionally, the execution of the MM itself is also done by library functions. MM execution is realized via `sparseDenseMatMul` and mapping of Tensors via `createSparseDenseMatMulLHS` and `createSparseDenseMatMulRHS`. In

contrast to the dense $\times$ dense MM, the input matrix A is composed of two dense matrices: metaInfo and NzValues. The first matrix contains positional sparsity information with the second one containing nonzero values [19].
By using two dense matrices to describe a sparse matrix, the memory requirements increase unexpectedly. This implementation is based on [15].

`gpu-cusparse-coo and gpu-cusparse-csr:` cuSPARSE is the NVIDIA library for handling sparse matrices and contains functionality for linear algebra. According to [58], it targets matrices with a sparsity of $>95\,\%$. MM execution is realized via `cusparseSpMM` and data allocation via `cusparseCreateDnMat` for the dense matrices in both implementations. The difference between gpu-cusparse-coo and gpu-cusparse-csr is the underlying storage format for the sparse input matrix A. gpu-cusparse-coo calls `cusparseCreateCoo` while gpu-cusparse-csr calls `cusparseCreateCsr`.

Sparse Squared $\times$ Dense Squared MM

Fig.4.18 and Fig.4.19 show the performance of a squared MM on the IPU and the GPU with a sparse input matrix A and a dense input matrix B. We regard the time for data compression as similar on both devices, thus, we only measure the execution time of the MM. Similar to the Dense Squared $\times$ Dense Squared MM, the problem size is calculated by using equation 4.4, neglecting the compression by the data formats such as CSR or COO. We choose this metric to compare against the dense equivalent. In contrast to dense MM, the GPU is not able to leverage the sparse computation to the Tensor Cores.
The horizontal lines represent the theoretical peak FP32 performance of the IPU with $62\,500$ GFLOPs and the GPU with $10\,300$ GFLOPs respectively. The theoretical peak TF32 performance of the GPU is added as well for reference reasons. The IPU surpasses the theoretical peak performance with $76\,231$ GFLOPs at a sparsity of $99\,\%$. On the other hand, the GPU surpasses the theoretical peak performance by a larger margin with $90\,039$ GFLOPs (gpu-cusparse-coo) and $93\,215$ GFLOPs (gpu-cusparse-csr) depending on the chosen implementation at a sparsity of $90\,\%$.
For both systems, a higher sparsity translates expectedly to a higher performance, as less nonzero entries mean less computational load due to the result element always being zero. Additionally, as the performance was calculated using equation 4.5, the problem size was handled as if there was no compression.
When comparing the behavior when increasing the sparsity, a difference between IPU

and GPU becomes apparent. For both GPU implementations, gpu-cusparse-coo and gpu-cusparse-csr, the GPU reaches a performance plateau for each given sparsity. The problem size at which this plateau is reached correlates positively with the sparsity. We assume that this is due to the SMs being sufficiently used when having to calculate a larger quantity of nonzero elements.

For the IPU, such a plateau is not reached for any problem size. We expect that the reason for this is that much additional data is allocated so that the hardware can not use reach its theoretical peak.

Similar to the dense MM, the GPU can handle much larger problem sizes, the difference is even more substantial than for the dense equivalent if we compare Fig.4.6 with Fig.4.18 and Fig.4.19.

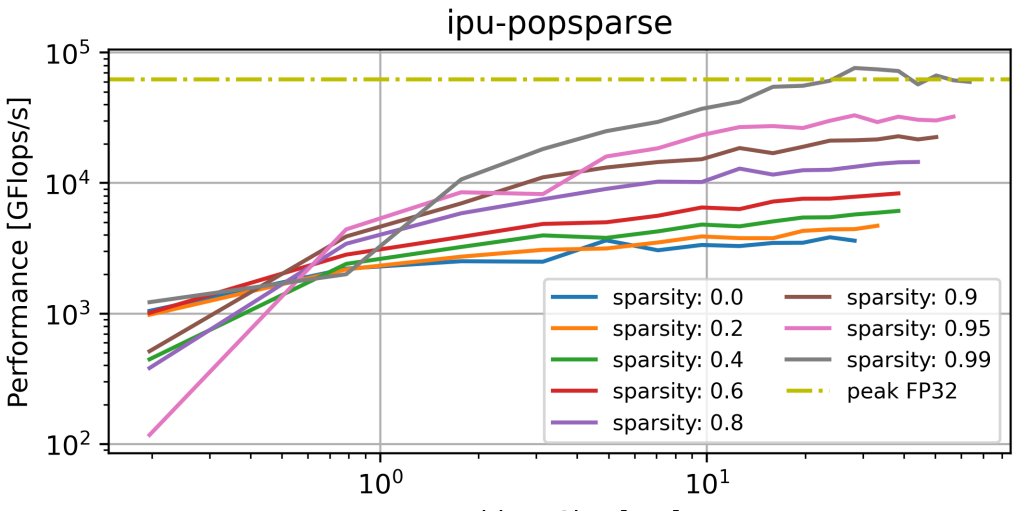


Figure 4.18: Sparse Squared × Dense Squared MM IPU

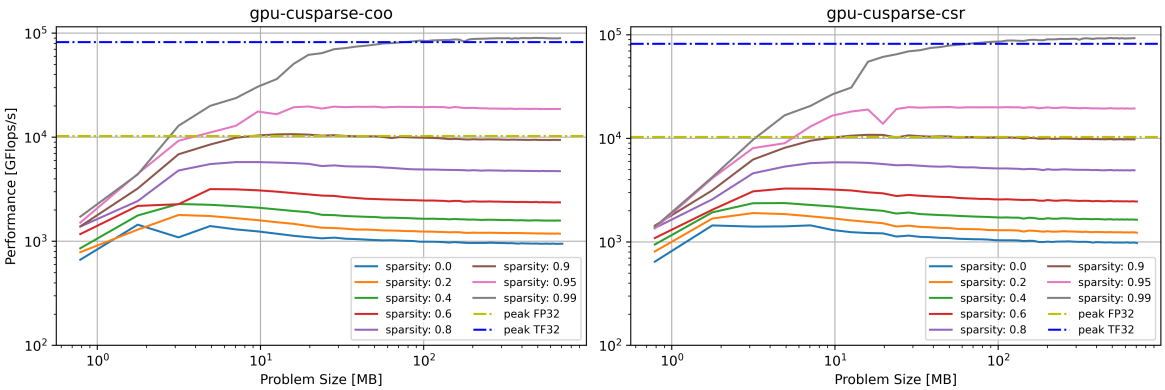


Figure 4.19: Sparse Squared × Dense Squared MM GPU

Compared to the results shown in Fig.4.6, the peak performances of gpu-cusparse-coo

and gpu-cusparse-csr are almost identical with a sparsity of $90\,\%$ to gpu-cublas (FP32). While decreasing the proportion of nonzeros in the input matrix increases the performance further, the converse is true for a higher nonzero count. This was to be expected since the cuSPARSE library aims at a sparsity of over $95\,\%$ [58]. At a sparsity of $99\,\%$, the performance is comparable to the peak performance of gpu-cublas (TF32). This means that, if Tensor Cores are available on a given GPU, it is still recommended to use the dense MM if the memory suffices. On the IPU, ipu-popsparse outperforms ipu-poplin only a sparsity of $99\,\%$. We assume that this is due to the sparse library not being able to benefit from the AMP unit [26] located on each IPU-Tile and the reference code from [15] relying on multiple libraries such as Boost.

To get deeper insights of the IPU behavior depicted in Fig.4.18, we use the PopVision Graph Analyser. A thorough analysis of gpu-cusparse is omitted as the focus of this work is the IPU. Since ipu-popsparse only outperformed ipu-poplin2 for a sparsity of $99\,\%$, this is the only scenario that is analyzed in the following. Fig.4.20 shows the performance data for ipu-popsparse depending on the problem size. Similar to Fig.4.9, we include the number of Edges, Variables, Vertices and Compute Sets and the total memory allocated to compare.

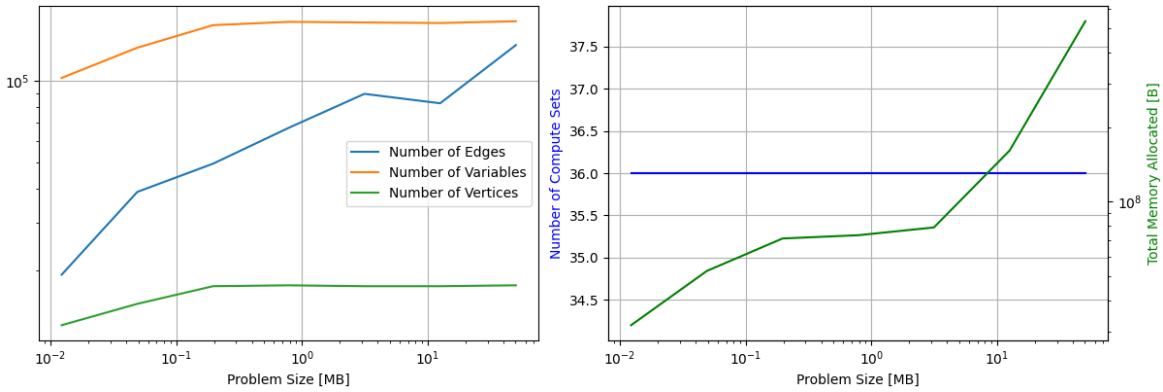


Figure 4.20: Underlying performance data of ipu-popsparse at a sparsity of $99\,\%$. Generated with PopVision Graph Analyser

Generally, the behavior of the curves in Fig.4.20 is similar to Fig.4.9. The number of Edges increase at a higher rate than the other two metrics. As for the dense equivalent, this is due to the bigger problem size and therefore less spare memory on the IPU-Tiles to compensate data movement. Thus, more data needs to be exchanged between the IPU-Tiles. Different to Fig.4.9, the number of Variables and Vertices increase only for a problem size smaller than $0.2\,\mathrm{MB}$. All three metrics have in common that their values are in general higher than for the ipu-poplin2 data at every given problem size. We assume

that this is due to the increased load for the processing elements to calculate sparse data. This also explains why ipu-popsparse outperforms ipu-poplin2 only with a sparsity of $99\,\%$. In contrast to ipu-poplin2, the number of Compute Sets remains constant for every given problem size with ipu-popsparse. This is due to the sparse implementation allocating much more memory than the dense equivalent, as depicted by Fig.4.9 and Fig.4.20. This results in ipu-popsparse beingle able to process smaller problem sizes than ipu-poplin2. Hence, using PopSparse with the IPU is not recommended unless small problem sizes with sparsities of more than $99\,\%$ should be computed.

5 Butterfly Approximations on GPU and IPU

As described in earlier chapters, `Butterfly` and `Pixelated Butterfly` aim at replacing linear layers in ML, that are represented with `torch.nn.Linear`, to decrease both, the computational time and the memory footprint for a given workload. Both approaches rely on an $O(NlogN)$ algorithm whereas a linear layer relies on an $O(N^2)$ algorithm [10]. In theory, this leads to `Butterfly` and `Pixelated Butterfly` outperforming `torch.nn.Linear` for large matrix sizes. To benefit from the Butterfly Factorization, both approaches apply the divide-and-conquer scheme from the Tukey FFT algorithm, and, in doing so, generate the learnable Butterfly Factors. Especially for `Pixelated Butterfly`, due to Flat Block Butterfly and the low-rank terms, additional overhead is added to `Butterfly`. To support that both, `Pixelated Butterfly` and `Butterfly` perform favorably compared to linear layers in regard to execution time and memory footprint, [10] and [6] show with the help of various experiments that training and inference benefit from the approaches.

To analyze if `Butterfly` and `Pixelated Butterfly` are able to speedup computation not only on a GPU, we test them on the GC200 IPU. We assume that for the IPU, with its IPU-Tiles, which are able to compute independently, `Butterfly` should have a more positive impact on the IPU than on the GPU. As `Pixelated Butterfly` is designed for the GPU, we assume that the performance improvement is rather minimal. As this work focuses on accelerators, we neglect a comparison to the CPU. Future work will evaluate if the claims hold true on a CPU as well.

Similar to the previous chapters, we include GPU results for comparison, with an evaluation if the claims from [10] and [6] hold true on a newer generation GPU with Tensor Cores supporting TF32. Compared to [10], in which they use P100 GPUs, the Tensor Cores on the A30 GPU support TF32, which increase the performance of linear layers with single-precision floating point. [6] relies on V100 GPUs with first generation Tensor Cores supporting only FP16. We extend [10] by using a newer architecture and by running the experiments with both, Tensor Cores turned off and on. This chapter is organized as follows:

First, we compare the training and inference time on the IPU and GPU of `Butterfly` and

`Pixelated Butterfly` against `torch.nn.Linear`. Then we analyze the results of the three different layers. On the GPU, we use two implementations of `Butterfly` from [9], one of them with the CUDA interface and the other in plain PyTorch. We include both variants to evaluate the performance implications of injecting CUDA to a given PyTorch model. On the IPU, we use the plain PyTorch implementation as CUDA is not supported on the IPU and as PyTorch can, in most cases, be directly included in PopTorch with minor changes to the source code. For the `Pixelated Butterfly` implementation, we rely on the plain PyTorch implementation from [2] for both processing units. In contrast to the implementation referenced in [6], it does not rely on additional libraries such as Triton or HuggingFace. Thus, it can be directly incorporated in PopTorch. Future work will examine the performance implications by the aforementioned libraries. This section corresponds to section 4.3 of [10].
Second, we evaluate the execution time and neural network compression on the respective device for the single-hidden layer benchmark with the CIFAR10 and MNIST dataset. In contrast to [10], we use the plain MNIST dataset instead of MNIST-bg-rot and MNIST-noise because the last two mentioned are not available on [1]. These experiments correspond to section 4.2 of [10] with the addition of the execution time to evaluate the computational throughput. As in [10], we rely on the framework of [67] for the single-hidden layer benchmark. A detailed description is given in the respective section.
As we restrict ourselves to a single processing unit and do not use any other framework apart from PyTorch and PopTorch with the exception of CUDA, we are not able to run any experiment of [6]. Future work will evaluate if the claims of both papers hold true for bigger applications with the use of multiple devices.

5.1 Characterization of Butterfly Approximations

This chapter examines if `Butterfly` and `Pixelated Butterfly` outperform a dense MM (`torch.nn.Linear`) in terms of processing speed. To do so, we measure the execution time of `torch.nn.Linear`, `Butterfly` and `Pixelated Butterfly` with the same matrix dimensions and compare against each other. This experiment resembles section 4.3 of [10] with the A30 GPU and GC200 IPU instead of the P100 [10] GPU or V100 GPU [6].
In contrast to their work, where the batch size is set to 2^8, we vary the batch size as well to allow a comparison against chapter 4.3. In addition to [10], we include the results for `Pixelated Butterfly` which are lacking in [10] and [6]. We iterate 1000 times and

use the mean execution time to overcome warmup effects and to stabilize the results. As we want to measure the execution time of the layers exclusively, we do not measure the overall time for the forward and backwards path as in [10]. By doing so, we exclude `torch.autograd.grad()` which is embedded in the benchmark code of [9]. Our benchmark resembles a single forward path.

Fig.5.1 shows the execution time for `torch.nn.Linear`, `Butterfly` and `Pixelated Butterfly` for the GPU, both with deactivated (left plot) and activated (right plot) Tensor Cores for squared input matrix dimensions.

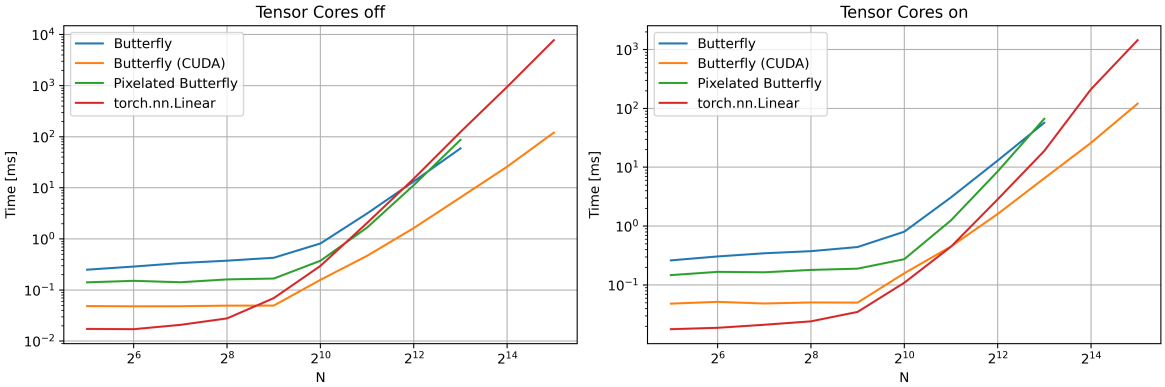


Figure 5.1: Training Time of torch.nn.Linear, Butterfly and Pixelated Butterfly for training on the GPU

With the Tensor Cores turned off, `torch.nn.Linear` outperforms `Butterfly` for $N \leq 2^{12}$ and `Pixelated Butterfly` for $N \leq 2^{10}$. The highest difference in execution time with `torch.nn.Linear` being $1.53\times$ faster than `Butterfly` is $1.09\,\text{ms}$ at a matrix dimension of $N = 2^{11}$. For `Pixelated Butterfly` this corresponds to $8.8\times$ with $0.13\,\text{ms}$ at $N = 2^6$ while for `Butterfly` (CUDA), this difference is $0.031\,\text{ms}$ at $N = 2^5$ being $2.8\times$ faster. This was to be expected since both replacements for `torch.nn.Linear` factorize the dense matrices, leading to additional work for the GPU. The higher execution time for `Pixelated Butterfly` compared to `Butterfly` can be explained by the low-rank terms and the Flat Block Butterfly which, on top of the Butterfly Factorization, aligns the data and approximates the product by a sum. When using the CUDA interface, as represented by `Butterfly` (CUDA) in Fig.5.1, the execution time of `Butterfly` (CUDA) is lower than `Butterfly` for every given problem size. Only for problem sizes with $N \leq 2^9$, `torch.nn.Linear` outperforms `Butterfly` (CUDA). This was to be expected since the CUDA interface allows to speedup the computation by adapting the algorithm to the GPU architecture.

For an increased dimension of N, `torch.nn.Linear` is outperformed by `Butterfly`,

`Pixelated Butterfly` and `Butterfly` (CUDA). The maximum speedup is $2.10\times$ with a difference of 64.63 ms for `Butterfly` at $N = 2^{13}$, $1.42\times$ with 36.89 ms for `Pixelated Butterfly` at $N = 2^{13}$ and $64.5\times$ with 7636.6 ms for `Butterfly` (CUDA) at $N = 2^{15}$. This verifies the claims of [10] with regard to the execution time as `Butterfly` and `Pixelated Butterfly` rely on an $O(N \log N)$ algorithm. `torch.nn.Linear`, in contrast, relies on an $O(N^2)$ algorithm. With bigger problem sizes, the improvements in terms of execution time by replacing `torch.nn.Linear` with either `Butterfly` or `Pixelated Butterfly` are greater. `torch.nn.Linear` is capable of processing larger datasets than `Butterfly` and `Pixelated Butterfly` as the memory is not sufficient for the two Butterfly approaches. We assume that this is due to the increased memory requirements for the sparse matrices, similar to the CSR compression with low sparsity. As there are $logN$ nonzeros, these requirements increase with bigger input matrix dimensions. Future work will examine why this does not hold true for `Butterfly` (CUDA).

With the Tensor Cores turned on, only the execution times of `Pixelated Butterfly` and `torch.nn.Linear` are reduced as the Tensor Cores only speed up dense MM. For `Pixelated Butterfly`, the relative performance improvement due to the Tensor Cores is small compared to `torch.nn.Linear`. `torch.nn.Linear` outperforms `Butterfly` and `Pixelated Butterfly` for every given problem size. `Butterfly` (CUDA) outperforms `torch.nn.Linear` by $11.95\times$ at $N = 2^{15}$ by 1317.53 ms while falling short by 0.05 ms for $N = 2^{10}$, being $1.46\times$ slower. We assume that only the multiplication of the low-rank terms benefits for `Pixelated Butterfly`. This verifies our claims that only dense MM benefits from the Tensor Cores on the A30 GPU. Compared to the left plot in Fig.5.1, `torch.nn.Linear` outperforms both `Butterfly` and `Pixelated Butterfly` for every given problem size. This is due to the Tensor Cores allowing more throughput than the SMs. `Butterfly` (CUDA), in contrast, outperforms `torch.nn.Linear` for problem sizes with $N > 2^{11}$. The point where the lines of `Butterfly` and `Pixelated Butterfly` intersect is shifted to the right when activating the Tensor Cores due to the aforementioned reason. We conclude that on the GPU, as soon as TF32 is supported, `Butterfly` and `Pixelated Butterfly` are not viable in terms of execution time. Only in case the CUDA interface is available, `Butterfly` is applicable for large matrix input dimensions. The next chapter determines if this holds true on a real-world example, also with regard to model compression. If TF32 is not supported, replacing `torch.nn.Linear` is more appealing. Depending on the requirements and the matrix dimensions, `Butterfly` and `Pixelated Butterfly` are viable replacements. Fig.5.2 shows the execution time of `torch.nn.Linear`, `Butterfly` and `Pixelated Butterfly` in PopTorch on the IPU.

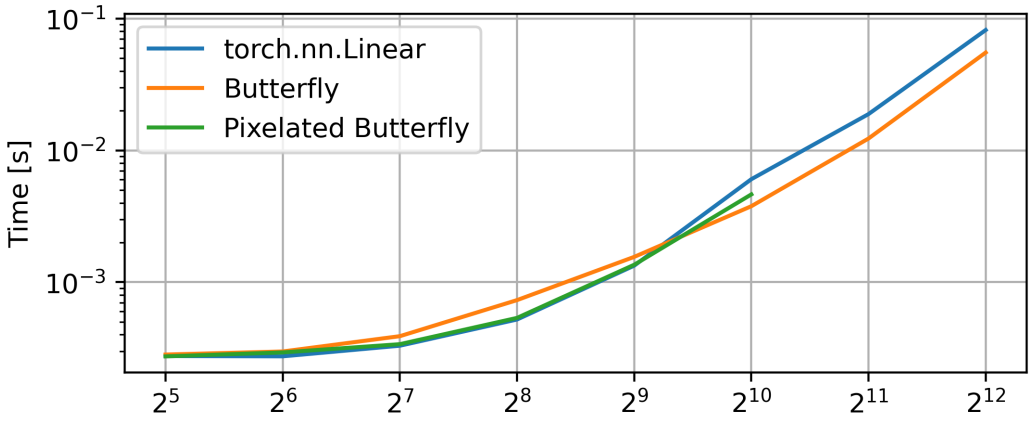


Figure 5.2: Training Time of torch.nn.Linear, Butterfly and Pixelated Butterfly for training on the IPU

As mentioned before, the data movement cannot be extracted from the time measurements in PopTorch. To minimize their share, we run the layer 1000 times and take the mean execution time, identical to the experiments on the GPU. We assume that without the data movement, the performance differences between the layers would be more drastic. The general behavior is similar to Fig.5.1. For matrix dimensions with $N < 2^{10}$, `torch.nn.Linear` outperforms `Butterfly` and `Pixelated Butterfly`. At $N = 2^9$, `torch.nn.Linear` outperforms `Butterfly` by $1.17\times$ and `Pixelated Butterfly` by $1.02\times$ with a difference of 0.22 ms and 0.026 ms, respectively. For $N > 2^{10}$, it is the other way round with `Butterfly` outperforming `torch.nn.Linear` by 26.62 ms ($1.48\times$ faster) at $N = 2^{12}$ and `Pixelated Butterfly` outperforming `torch.nn.Linear` by 1.41 ms at $N = 2^{10}$, being $1.3\times$ faster.

With bigger problem sizes, the additional work for the Butterfly Factorization pays off, resulting in better performance and verifying the claims of [10] for the IPU. As depicted by Fig.5.1, `Pixelated Butterfly` performs similar to `torch.nn.Linear` although the latter taps out earlier due to memory constraints. We assume that this is due to the memory requirements caused by the additional low-rank terms for `Pixelated Butterfly` and the code generated by the Flat Block Butterfly.

In order to dissect the underlying performance differences for `Butterfly` and `Pixelated Butterfly` on the IPU, we use the PopVision Graph Analyser, similar to section 4.3.

In contrast to Fig.5.2, we include a single iteration in Fig.5.3. The reason for this is the overhead of the profiling report generation and unpredictable problems with the Slurm integration resulting in "IPU-Fabric errors". As soon as these issues are resolved, results

covering the same number of iterations as in Fig.5.2 will be included in future work. Similar to the findings of 4.3, the number of Compute Sets strongly correlates with the number of Variables, Edges and Vertices at a given problem size. Fig.5.3 shows the number of Compute Sets and the total memory consumption over the problem size. Fig.5.3 is generated with the help of the PopVision Graph Analyser.

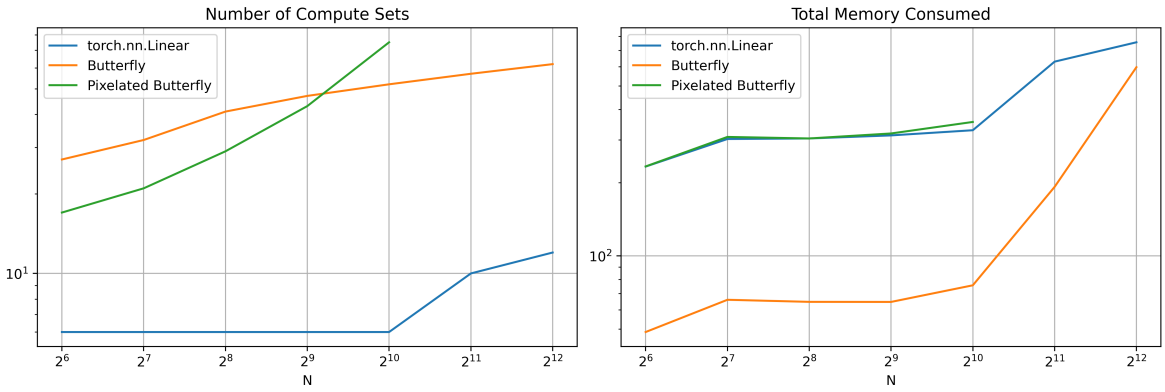


Figure 5.3: Number of Compute Sets for `torch.nn.Linear`, `Butterfly` and `Pixelated Butterfly` on the IPU with squared matrix dimensions

For `torch.nn.Linear`, the number of Compute Sets remains constant for squared matrix dimensions $N < 2^{10}$. With increased matrix dimensions, the number of Compute Sets increases rapidly, with double the Compute Sets for a squared matrix dimension of $N = 2^{11}$ being also the biggest problem size fitting on the IPU, see chapter 4.3. This explains the increase in Compute Sets for $N > 2^{10}$ as the IPU is not able to allocate sufficient temporal data due to the limited amount of available memory. Therefore, in order to calculate the results, data has to be transferred via the interconnect to guarantee that the correct IPU-Core continues with the execution. To accomplish this, more Compute Sets have to be created. When comparing Fig.5.3 with Fig.4.9, our claim that `torch.nn.Linear` represents a dense MM is verified. Therefore, the same behavior and explanations apply here as well.

For `Butterfly`, the number of Compute Sets is always higher than for `torch.nn.Linear`, due to the factorization taking place to generate the Butterfly Matrix. With an increase in problem size, the number of Butterfly Factors increases and therefore the number of Compute Sets as well. With a squared matrix dimension of $N = 2^{11}$, the number of Compute Sets is $5.2\times$ higher than for `torch.nn.Linear`.

`Pixelated Butterfly` can process input matrix sizes up to an input size of $N \leq 2^{10}$ which is four times smaller than with `Butterfly` and `torch.nn.Linear`. This is due to the additional low-rank terms which are added for an accuracy increase [6]. More

computational overhead is created by the Flat Block Butterfly, which, in addition to the Butterfly Factorization, approximates the product of Butterfly Factors by a sum and block-aligns the data. This explains the number of Compute Sets being even higher than for `Butterfly` for matrix dimensions greater than $N = 2^9$.
Different to chapter 4.3, the number of Compute Sets does not directly translate to the performance, see Fig.5.2. This can be explained by the difference in workloads: The replacements based on Butterfly Factorization reduce the memory needed for data by increasing the arithmetic intensity. It is achieved by relying on sparse data, see Fig.2.6. This claim is verified in the right plot of Fig.5.3.
The curves of the total memory consumption differ to the number of Compute Sets. For every problem size, `Butterfly` consumes less memory than `Pixelated Butterfly` and `torch.nn.Linear`. This is because, in contrast to `torch.nn.Linear`, the Butterfly Factorization relies on sparse data, and therefore more space can be used for other layers, allowing to process bigger applications with smaller memory requirements. As `Butterfly` relies on powers of two for m, n, k and, as the IPU cannot handle squared matrices with $N > 2^{11}$ elements for a dense MM, even with `Butterfly`, extending this layer exceeds the amount of available memory on the IPU. `Pixelated Butterfly` has roughly the same memory footprint as the linear layer, a surprising property assumed to be due to the additional low-rank terms. As mentioned before, `Pixelated Butterfly` adds Flat Block Butterfly and low-rank terms to the Butterfly Factorization which reflects the high amount of memory allocated.
To compare against the findings of [10], we include benchmarks with `batch size` $= 2^8$ for the GPU and the IPU and depict the speedup over dense MM, with the same benchmark setup as for the previous figures. In contrast to [10], we use `torch.nn.Linear` and not `cublasSgemm` as a baseline to allow a fair comparison between GPU and IPU. As the performance decay is negligible and as `torch.nn.Linear` calls `cublasSgemm` in the backend on the GPU, the results are still comparable. The FFT is not included in the upcoming Figures. Fig.5.4 depicts the speedup of both Butterfly Approximations against `torch.nn.Linear` on the GPU, both with the Tensor Cores turned off (left plot) and turned on (right plot) with a fixed batch size of 2^8.

With the Tensor Cores turned off, the behavior of the curve for `Butterfly` is very similar compared to [10]. Only for large input matrix dimensions, it surpasses the performance of `torch.nn.Linear`. However, in our experiments, the performance gain is lower than in [10]. We assume that this is due the difference between our benchmark setting and the one from [10]. As the general behavior is similar, we view the results of [10] as verified

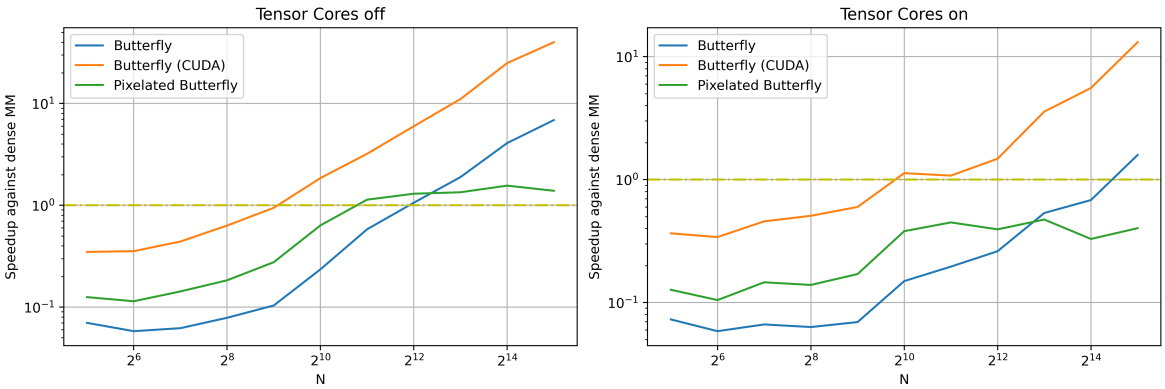


Figure 5.4: Speedup of Butterfly and Pixelated Butterfly against torch.nn.Linear with a fixed batch size of 2^8 on the GPU

and applicable for our setup. In contrast to the squared matrices depicted in Fig.5.1, with the Tensor Cores turned off, `Pixelated Butterfly` outperforms `Butterfly` for $N < 2^{13}$ elements. The reason for this is that there are more nonzero elements for bigger matrix dimensions. Therefore the Flat Block Butterfly does not have such a strong influence on the processing speed. With $N > 2^{10}$, `Pixelated Butterfly` outperforms `torch.nn.Linear` and settles at a speedup of around $1.5\times$. For `Butterfly` the maximum speedup is higher compared to `Pixelated Butterfly`, with $7\times$ at peak. With the CUDA interface, the maximum speedup is $40\times$.

When turning the Tensor Cores on, the performance improvement diminishes with a maximum speedup of $1.6\times$ for `Butterfly` and $0.5\times$ for `Pixelated Butterfly`. This backs our claim that `Butterfly` and `Pixelated Butterfly` do not perform favorably in terms of execution time when turning the Tensor Cores on. Only with the CUDA interface, a speedup of $13.1\times$ is achieved.

Fig.5.5 shows the speedup of `Butterfly` and `Pixelated Butterfly` on the IPU with a fixed batch size of 2^8. Compared to the results on the GPU, the behavior for `Butterfly` is similar. For matrix dimensions of $N > 2^{11}$, `Butterfly` outperforms `torch.nn.Linear`. In contrast to the GPU results, the benefit of using `Butterfly` is small compared to the GPU. The maximum speedup for `Butterfly` is $1.14\times$ and $0.94\times$ for `Pixelated Butterfly`. We assume that this can be explained by the AMP units of the IPU, as these only accelerate `torch.nn.Linear`, making this hardware unit similar to the Tensor Cores on the GPU. `Pixelated Butterfly` performs worse than `torch.nn.Linear` for every given problem size. For matrix $N < 2^{10}$, `Pixelated Butterfly` performs better than `Butterfly` but worse than `torch.nn.Linear`. We assume that the same explanations apply as for the squared matrices.

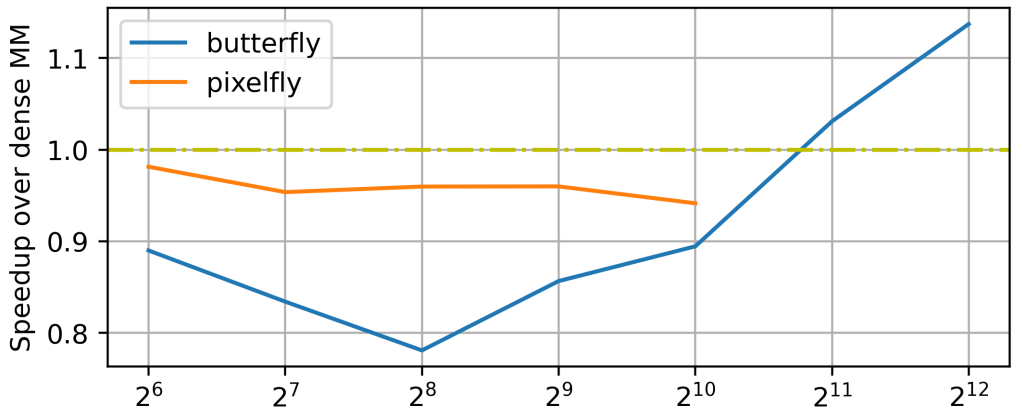


Figure 5.5: Speedup of Butterfly and Pixelated Butterfly against torch.nn.Linear with a fixed batch size of 2^8 on the IPU

In summary, the combination of computation and memory have to be considered for performance evaluation on the IPU. One metric alone does not directly translate to the performance. For the IPU, `Butterfly` seems viable due to the reduced memory footprint and the increased performance. For `Pixelated Butterfly`, the additional work for the Flat Block Butterfly and the additional low-rank terms increase the space complexity so that this approach does not seem viable on this device. To inspect if this holds true for real-world examples, we include both approaches in the upcoming chapter.

5.2 Neural Network Compression

Since both processing units in this work aim at ML workloads and all previously conducted experiments point to them, we include an exemplary real-world ML workload in this thesis. As a representative ML application, we choose the single-hidden layer benchmark from [67] with the MNIST and CIFAR10 datasets and a variety of structured sparse approaches. This benchmark was also conducted in [10], section 4.3. We consider this as an appropriate workload for evaluation as exchanging the single-hidden layer leads to the maximum impact with regard to accuracy, parameter count and execution time.
Secondly, this model fits not only on a single GPU but also on a single IPU, aligning with our restrictions.
Thirdly, the framework from [67] is open-source and can thus be easily extended by `Butterfly` and `Pixelated Butterfly` and embedded in PopTorch. In addition to the

findings of [10], we add the execution time to Table 1 of their work to evaluate the computational throughput. Fig.5.6 shows the general architecture of the benchmark.

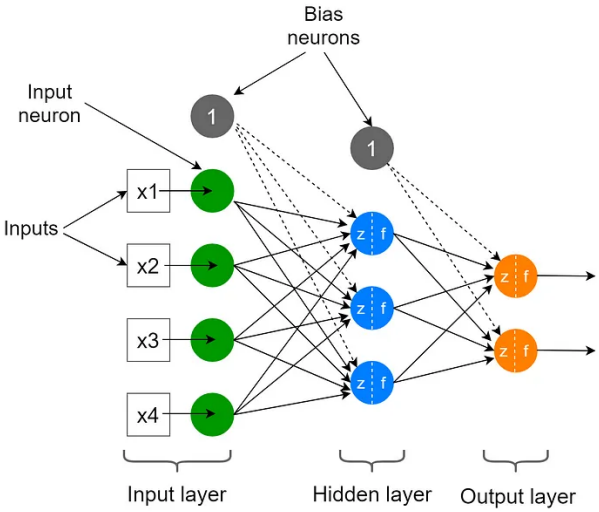


Figure 5.6: Single Hidden Layer Neural Network Architecture [61].

We use the same hyperparameters as [10] except for the learning rate. In contrast to [10], we fix the learning rate to 0.001. Similarly, we exchange the hidden layer with structured approaches. Table 5.1 shows the chosen hyperparameters, applying to GPU and IPU.

Hyperparameter	
Batch size	50
Validation set	15 % of training data
Momentum	0.9
Optimizer	SGD
Activation function	ReLU
Loss function	Cross-Entropy Loss

Table 5.1: Hyperparameters for SHL benchmark

This chapter investigates if the claims of [10], section 4.3, remain valid for different datasets and on different processing units than presented in their work. We extend [10] by running the experiments on both, a dedicated ML processor, the IPU, and a newer generation GPU. This is especially interesting as many properties of the GPU do not apply to the IPU as shown in earlier chapters. We evaluate if the applicability of the approaches is independent of the underlying processing unit. This is also backed by conducting the experiments on the A30 GPU as the underlying microarchitecture and set of features differ to other GPU generations. Additionally, we extend [10] by including `Pixelated Butterfly` and

`torch.nn.Linear` to replace the hidden layer, thereby evaluating if the findings of the previous section hold true for a ML application.
As not all structured sparse approaches included in [10] work on our experimental setup due to version incompatibilities of the FFT library in PyTorch, we do not include LDR-TD and Toeplitz-like in our work. Future work will extend our findings with other structured sparse approaches. Also, as PopTorch depends on the available POPLAR version and is only compatible to certain PyTorch versions as described in [28], using the same version of PyTorch for the GPU and IPU is not possible.

Preprocessing of datasets In advance of running the training and inference on the processing units, we preprocess the datasets, as typically applied in ML workloads. It is especially important for making our results reproducible. Fig.5.7 shows the steps for the MNIST dataset from [45]. It consists of data normalization to apply the same technique as in the MNIST reference implementation from [62] and reshaping the data to a 1D-Tensor to align with the requirements of [67].

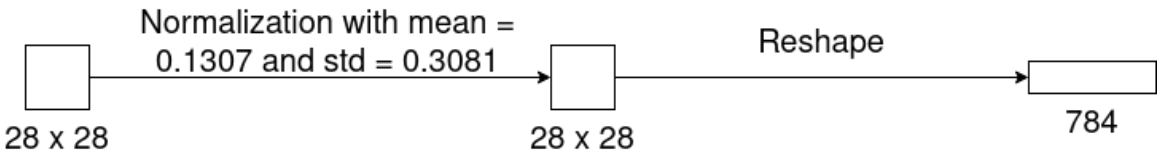


Figure 5.7: Preprocessing of MNIST dataset for single-hidden layer benchmark

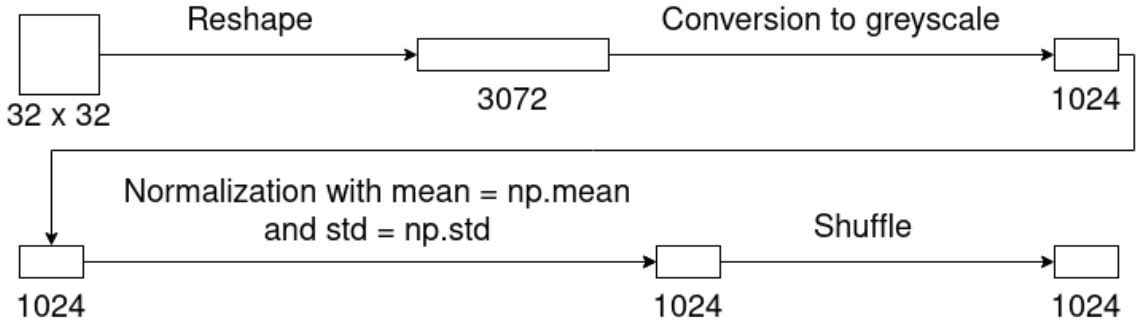


Figure 5.8: Preprocessing of CIFAR10 dataset for single-hidden layer benchmark

For the CIFAR10 dataset from [42], the preprocessing step is already included in the codebase of [67]. To compare against their results, we apply the same techniques. Fig.5.8 shows the processing steps for the CIFAR10 dataset.
Table 5.2 shows the results, both with the Tensor Cores activated and deactivated for the

GPU. In contrast to [10], our results are described with absolute and not relative data to make our analysis more transparent.
First, our results for CIFAR10 with the Tensor Cores deactivated have to be compared against the findings of [10] for validation purposes. As the results of [10] are identical to the ones from [67], we assume that the authors of [10] solely extended the results of [67] with `Butterfly`. Additionally, [67] does not describe the experimental setup, thus, we assume that they used the P100 GPU as in [10].
Similar to [10], `Butterfly` achieves the highest test accuracy among the structured sparse approaches presented in their work. This also applies to the less challenging MNIST dataset. In general, all approaches achieve similar test accuracies compared to [10] except for low-rank which achieves around $18\,\%$ compared to the $32.28\,\%$ of [10]. The low-rank model is already implemented in [67], thus we assume that the different PyTorch versions lead to this difference. As our focus is on `Butterfly` and `Pixelated Butterfly`, analyzing this issue will be included in future work. Compared to the other structured approaches, `Pixelated Butterfly` achieves a similar test accuracy compared to `Butterfly`. We assume that this is due to same underlying algorithm leading to the Butterfly Factorizations. The linear layer performs very similar to the unstructured layer.
In Table 5.2, the benefit in terms of execution time for the Flat Block Butterfly becomes apparent. In contrast to `Butterfly`, which adds $11\,\mathrm{s}$ to the unstructured layer, `Pixelated Butterfly` outperforms `Butterfly` by almost $9\,\mathrm{s}$ with the Tensor Cores and $5\,\mathrm{s}$ without the Tensor Cores. This is due to the block-data alignment and the approximation of the product of Butterfly Factors by a sum, leading to better memory utilization and increased parallelism. However, the increased performance of `Pixelated Butterfly` with its additional low-rank terms for accuracy improvement comes at the cost of an increased parameter count. For the CIFAR10 dataset, this translates to a compression factor of 64.66 for `Butterfly` to the unstructured layer and a compression factor of 2.62 for `Pixelated Butterfly`. On the GPU, the fastfood and the `Pixelated Butterfly` approaches did not work on the MNIST dataset due to the requirements of the matrix sizes being a power of two. As `Pixelated Butterfly` allows to change its configuration, results covering the MNIST dataset will be included in future work.
When comparing the performance difference with active and inactive Tensor Cores, our claims from the previous chapters are verified. On the CIFAR10 dataset, the low-rank, `Pixelated Butterfly` and linear layer benefit from the Tensor Cores. This is because the low-rank terms of `Pixelated Butterfly` perform a MM which is sped up by the Tensor Cores as described earlier. This also applies to the linear layer, `torch.nn.Linear`.

Dataset	Metric	Test Accuracy		Execution Time [s]		N_{Params}
		w/ TC	w/o TC	w/TC	w/o TC	
CIFAR10	Unstructured	43.94	43.40	50.43	49.46	1 059 850
CIFAR10	Butterfly	42.27	40.75	61.93	61.46	16 390
CIFAR10	Butterfly (CUDA)	39.40	38.96	66.52	62.02	32 778
CIFAR10	Fastfood	38.64	37.94	53.55	51.15	14 346
CIFAR10	Circulant	28.74	29.21	54.26	53.92	12 298
CIFAR10	Low-rank	18.64	18.49	49.71	53.21	13 322
CIFAR10	Pixelated Butterfly	42.61	43.31	52.79	56.01	404 490
CIFAR10	linear	44.40	44.09	48.52	50.87	1 060 874
MNIST	Unstructured	94.82	95.01	53.05	52.67	623 290
MNIST	Butterfly	95.53	95.44	68.27	68.20	13 510
MNIST	Butterfly (CUDA)	95.50	95.56	68.55	65.76	29 898
MNIST	Circulant	89.45	92.03	55.94	56.22	9418
MNIST	Low-rank	41.95	41.93	49.71	52.31	13 322
MNIST	linear	95.33	95.12	54.95	53.92	624 074

Table 5.2: Single-Hidden-Layer (SHL) benchmark with different structured matrix approaches compared to unstructured matrix approach on the GPU, (TC stands for Tensor Cores)

Dataset	Metric	Test Accuracy	Execution Time [s]	N_{Params}
CIFAR10	Unstructured	44.7	24.69	1 059 850
CIFAR10	Butterfly	41.13	37.73	16 390
CIFAR10	Fastfood	37.68	60.7	14 346
CIFAR10	Circulant	28.40	21.82	12 298
CIFAR10	Low-rank	18.59	21.75	10 202
CIFAR10	Pixelated Butterfly	43.79	71.62	404 490
CIFAR10	linear	45.74	28.23	1 060 874
MNIST	Unstructured	94.07	24.89	623 290
MNIST	Butterfly	95.89	39.23	13 510
MNIST	Circulant	89.52	22.83	9418
MNIST	Low-rank	40.58	22.97	13 322
MNIST	linear	94.17	25.70	624 074

Table 5.3: Single-Hidden-Layer (SHL) benchmark with different structured matrix approaches compared to unstructured matrix approach on the IPU

For the MNIST dataset, the execution times only differ between activated and deactivated Tensor Cores for the low-rank approach, due to the small layer dimensions for the MNIST dataset compared to CIFAR10 as depicted in Fig.5.7 and Fig.5.8. Thus, the performance is almost independent of the Tensor Cores as shown in the left plot of Fig.4.6.
Although the execution time is reduced for some approaches with the Tensor Cores turned on, the test accuracy remains similar. For the approaches that do not benefit from the Tensor Cores, the test accuracy and the execution time remain the same. Thus, it is recommended to use the Tensor Cores whenever they are available on a given setup.
Table 5.3 shows the respective results for the SHL benchmark on the IPU.
For the IPU, the number of parameters is identical to the GPU for every given approach, making a direct comparison between the two processing units possible. Here, among the structured sparse approaches presented in [10], `Butterfly` performs best in terms of accuracy for both datasets. Similar to the GPU, `Butterfly` adds $13\,\mathrm{s}$ to the execution time of the unstructured layer while compressing the overall model. `Pixelated Butterfly` achieves $2.4\,\%$ higher test accuracy than `Butterfly`, similar to the results on the GPU. Different to the GPU, `Pixelated Butterfly` needs $1.9\times$ the time of `Butterfly`. We assume that the reason for this behavior is the memory alignment of the Block Butterfly. The GPU, being a block device, benefits from this measure while the IPU does not. Therefore, the additional work does not translate to a better performance. The upcoming chapter will include this insight for optimizations on the IPU.
Although the accuracies are similar on GPU and IPU, the execution times differ severly. Here, the differences between the two processing units become apparent as all approaches, except for `Butterfly` (CUDA) rely on the same source code. The only difference between the PyTorch and the PopTorch implementation is that the former returns the losses in addition to the output. With roughly the same test accuracy, both the unstructured and linear layer need half the time on the IPU than on the GPU while having lower TDP of $15\,\mathrm{W}$. The speedup is present on every approach except for `Pixelated Butterfly` due to the aforementioned reasons and fastfood. On the IPU, this is due to the underlying architecture being specifically designed for ML workloads. Thus it is recommended to use the IPU if the model fits onto that device as the IPU trades memory capacity with execution speed. Future work will include a variety of workloads to assess this statement.

6 Optimizations of Butterfly Approximations for the IPU

The previous chapters have demonstrated that exchanging `torch.nn.Linear` with either `Butterfly` or `Pixelated Butterfly` can be beneficial in terms of execution time and parameter count on GPU and IPU. The performance gain depends greatly on the given problem size and workload. As chapter five shows, both `Butterfly` and `Pixelated Butterfly` reduce the parameter count, resulting in lower memory requirements for a given ML model. These insights, coupled with the possibility to include both Butterfly approaches in any given PyTorch / PopTorch model without needing to change the source code, are the main motivation for this chapter. Here, we evaluate if `Butterfly` and `Pixelated Butterfly` can be improved on the IPU. This chapter is organized as follows: First, we analyze how the configuration parameters low rank size, block size and butterfly size affect `Pixelated Butterfly` in terms of execution time, test accuracy and parameter count on the IPU. We choose the single-hidden layer benchmark with the CIFAR10 dataset to allow a comparison to the previous chapter. High accuracy, low parameter count and low execution time are the objectives. `Pixelated Butterfly` is customized for the GPU as shown when comparing the execution times of Fig.5.1 with Fig.5.2 or the speedups of Fig.5.4 with Fig.5.5 respectively. As previous chapters contain the same configuration for both GPU and IPU, we assume that tweaking the configuration for the IPU increases the performance further. In contrast to `Butterfly`, `Pixelated Butterfly` relies on Flat Butterfly, which increases parallelism by approximating products by sums. We assume that this property fits well to the IPU as it executes in a MIMD fashion. Block Butterfly can be disabled by setting the `block size` to one. We assume that, as the IPU does not need dense data, that disabling Block Butterfly increases performance due to the smaller overhead.

Second, we provide a deeper analysis for `Butterfly` as both, `Butterfly` and `Pixelated Butterfly` rely on the same algorithm for generating the Butterfly Factors. To determine if `Butterfly` can be optimized within PopTorch, we first check the load balancing of `Butterfly`. This is due to the IPU performance depending greatly on the distribution of work as a given IPU-Tile cannot access data of another one. To allow a comparison to the dense MM, we set the dimensions to 2^{12} and present the Tile Map. After this analysis,

we provide a study on injecting POPLAR Code to a given PopTorch model. The main motivation here is the performance increase with regard to execution time when injecting CUDA to PyTorch on the GPU as shown in Fig.5.1 and Fig.5.4.

6.1 Parameter Search for Pixelated Butterfly

We run the single-hidden layer benchmark presented in the previous chapter with `Pixelated Butterfly` as a replacement for the hidden layer. To determine the impact of the configuration parameters, we run the same experiment with the CIFAR10 dataset and vary low rank size, block size and butterfly size on the IPU. We use the same hyperparameters for the model as in the previous chapter. Fig.6.1, Fig.6.2 and Fig.6.3 show the execution time, the parameter count and the test accuracy for the single-hidden layer benchmark with `Pixelated Butterfly` depending on block size, low rank size and butterfly size for every configuration with a memory footprint fitting on the IPU.

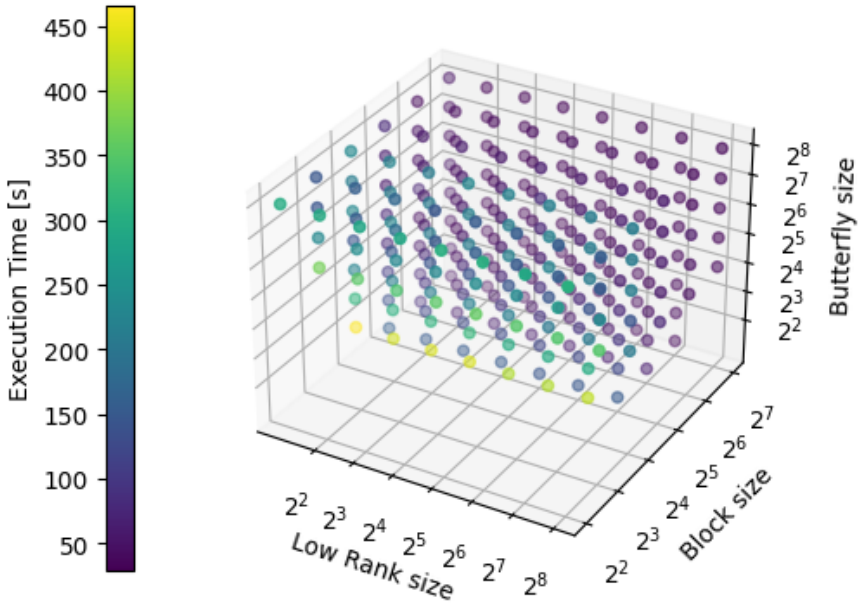


Figure 6.1: Execution Time of Pixelated Butterfly on the SHL benchmark with CIFAR10 dataset - varying block size, butterfly size and low rank size

In contrast to our expectations, `Pixelated Butterfly` can not be computed with low block sizes due to the allocation of more memory than available on the IPU. As the block

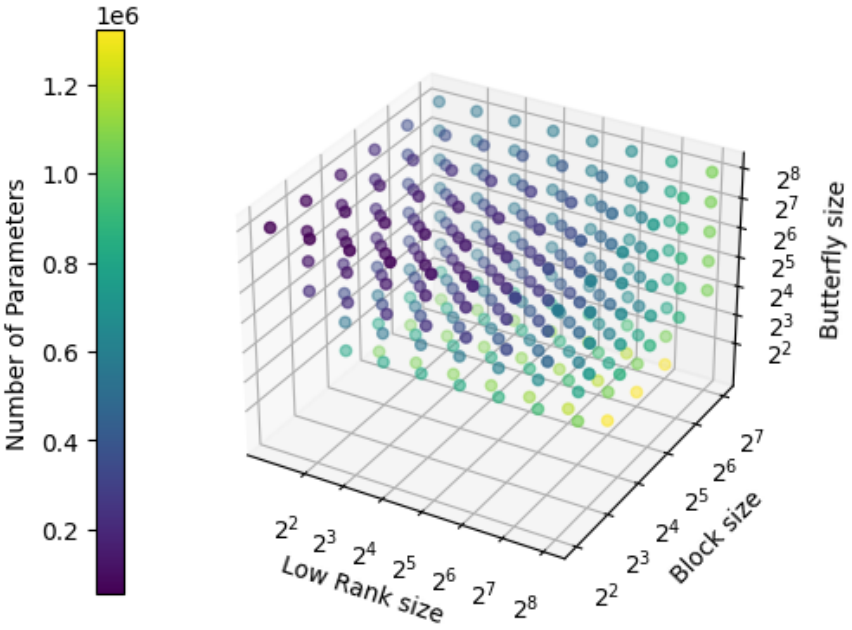


Figure 6.2: Parameter Count of Pixelated Butterfly on the SHL benchmark with CIFAR10 dataset - varying block size, butterfly size and low rank size

size determines the memory access costs, we assume that low block sizes lead to more data exchanges between the IPU-Tiles. As discussed in previous chapters, this requires more memory. Future work will check these claims with the help of the PopVision Graph Analyser.

To simplify the interpretation of the results, we filtered out the best $1\,\%$ configurations for every target metric. Fig.6.4 shows the best $1\,\%$ for the test accuracy on the given workload. As there is only a single configuration within the $1\,\%$ limit for the execution time and the parameter count, no Figure is shown.

With a block size of 2^4, a low rank size of 2^1 and a butterfly size of 2^2, the execution time is the highest with $465.37\,\text{s}$ ($accuracy_{test} = 43.76$, $n_{parameters} = 802\,826$). The best configuration is a block size of 2^7, a low-rank size of 2^3 and a butterfly size of 2^8 resulting in an execution time of $28.71\,\text{s}$ ($accuracy_{test} = 42.29$, $n_{parameters} = 552\,970$).

The top $1\,\%$ of test accuracies are achieved with block sizes between 2^4 and 2^7, low rank sizes between 2^6 and 2^8 and butterfly sizes between 2^2 and 2^8. The best test accuracy of 46.09 is achieved with a block size of 2^6, a low rank size of 2^7 and a butterfly size of 2^2 ($t_{exec} = 50.67\,\text{s}$, $n_{parameters} = 1\,060\,876$). In contrast, the test accuracy was the lowest at 36.51 with a block size of 2^2, a low rank size of 2^1 and a butterfly size of 2^8 ($t_{exec} =$

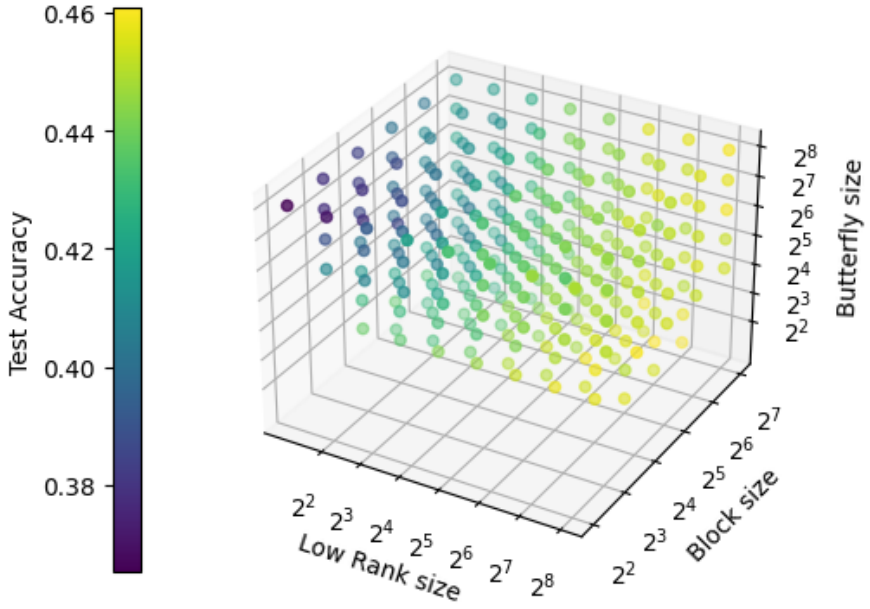


Figure 6.3: Test Accuracy of Pixelated Butterfly on the SHL benchmark with CIFAR10 dataset - varying block size, butterfly size and low rank size

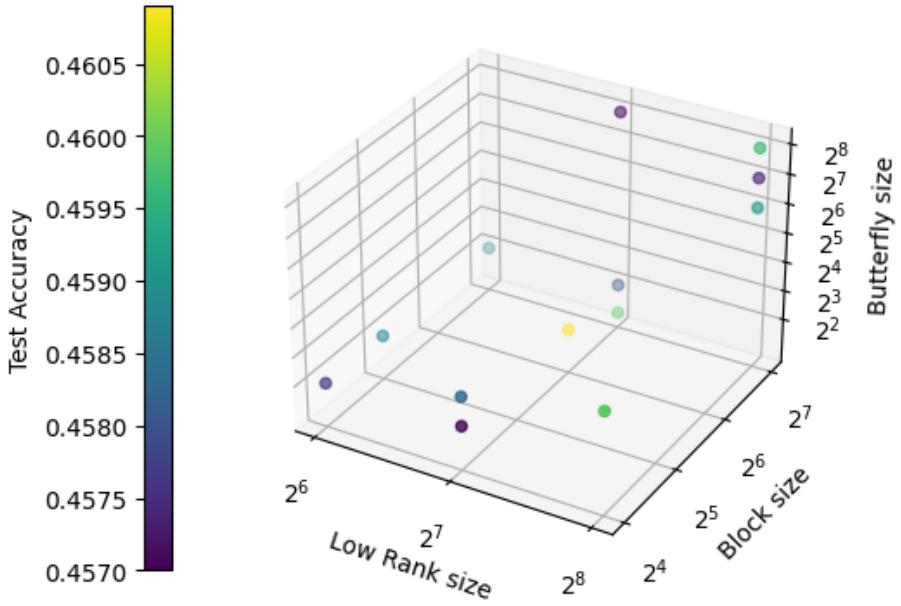


Figure 6.4: Top 1% Test Accuracy of Pixelated Butterfly on the SHL benchmark with CIFAR10 dataset - varying block size, butterfly size and low rank size

$297.13\,\text{s}$, $n_{parameters} = 53\,258$).
The lowest parameter count with $53\,258$ parameters was achieved with a block size of 2^2, a low rank size of 2^1 and a butterfly size of 2^8 ($t_{exec} = 297.13\,\text{s}, accuracy_{test} = 36.51$). Three configurations achieved the same maximum parameter count of $1\,323\,018$ parameters. In all configurations, the low rank size and the butterfly size are constant with 2^7 and 2^1 respectively, only the block size is different with 2^5, 2^6 or 2^7.
When comparing the best results with the previous chapter, we can achieve an accuracy increase of 2.3, a decrease in execution time of $42.91\,\text{s}$ and in the parameter count of $351\,032$.
To determine which configuration parameter has the greatest impact on the execution time, the test accuracy or the parameter count respectively, we make two of three parameters constant and vary the third. We do this for every combination of the constant parameters and extract the maximum standard deviation. Table 6.1, Table 6.2 and Table 6.3 show the results.

Butterfly size	Block size	Parameter	Mean	Std
2^2	2^4	Exec Time [s]	465.37	18.16
2^7	2^3	Test Accuracy	0.378	0.027
2^4	2^4	Number of Parameters	$344\,074$	$181\,317.00$

Table 6.1: Maximum standard deviation when varying low rank size for Pixelated Butterfly in SHL benchmark with CIFAR10 dataset

Block size	Low Rank size	Parameter	Mean	Std
2^3	2^1	Exec Time [s]	372.62	107.37
2^4	2^1	Test Accuracy	0.438	0.022
2^5	2^1	Number of Parameters	$1\,064\,970$	$326\,625.04$

Table 6.2: Maximum standard deviation when varying butterfly size for Pixelated Butterfly in SHL benchmark with CIFAR10 dataset

Low Rank size	Butterfly size	Parameter	Mean	Std
2^1	2^2	Exec Time [s]	465.37	192.23
2^1	2^6	Test Accuracy	0.389	0.014
2^1	2^7	Number of Parameters	$81\,930$	$184\,638.30$

Table 6.3: Maximum standard deviation when varying block size for Pixelated Butterfly in SHL benchmark with CIFAR10 dataset

For the execution time, the low rank size has the smallest impact with a maximum standard deviation of 18.16. This was to be expected since the low rank term is represented by

a dense matrix multiply and the IPU achieves higher throughput with bigger problem sizes as shown in chapter 4.3. As it also has the highest impact on the test accuracy with a standard deviation of 0.027, it is recommended to set the low rank size to the maximum. The greatest impact on execution time is created when varying the block size with $max_{Std} = 192.23$. The butterfly size has the biggest impact on the number of parameters with $max_{Std} = 184\,638.30$.
We conclude that for the single-hidden layer benchmark with the CIFAR10 dataset, it is beneficial to adapt the configuration of `Pixelated Butterfly` depending on the target parameter. There is no configuration being optimal with regard to execution time, test accuracy and parameter count at once, hence, it has to be chosen depending on the primary target. With the findings derived from Table 6.1, Table 6.2 and Table 6.3, the most heavily weighted input parameter depending on the desired target is found. Future work will analyze if this holds true for other ML applications and datasets.

6.2 Preliminary Considerations for Optimization of Butterfly

To evaluate which optimizations for `Butterfly` can be applied to the IPU, we first extend the analysis of chapter five.
One of the most important factors of the IPU performance is the load balancing. If not all IPU-Tiles are able to process data simultaneously, the performance decreases by $\frac{Tiles_{Idle}}{1472}$. Fig.6.5 shows the Tile map for `Butterfly` with squared input matrices at $N = 2^{12}$. We choose this input dimensions to make the results comparable against the IPU-Tile Maps of the POPLAR dense MM shown in chapter four, Fig.4.10.
Comparing the IPU-Tile Map of Fig.6.5 with Fig.4.10 reveals two insights:
First, the overall memory allocated is lower for `Butterfly` than for ipu-poplin1 or ipu-poplin2. ipu-poplin1 and ipu-poplin2 allocate roughly the same amount of memory with $841\,\text{MB}$ and $850\,\text{MB}$ (including gaps), respectively. In contrast to the two POPLAR implementations, `Butterfly` allocates $597\,\text{MB}$ at the same problem size, resulting in $28.1\,\%$ less memory consumption. This was to be expected since `Butterfly` sparsifies the input matrices, resulting in lower memory requirements. Second, the relative memory consumption between the IPU-Tiles is lower for `Butterfly` than the two POPLAR implementations. ipu-poplin1 and ipu-poplin2 allocate between $487\,\text{KiB}$ and $593\,\text{KiB}$ of memory while `Butterfly` allocates between $389\,\text{KiB}$ and $431\,\text{KiB}$. This translates to `Butterfly` having

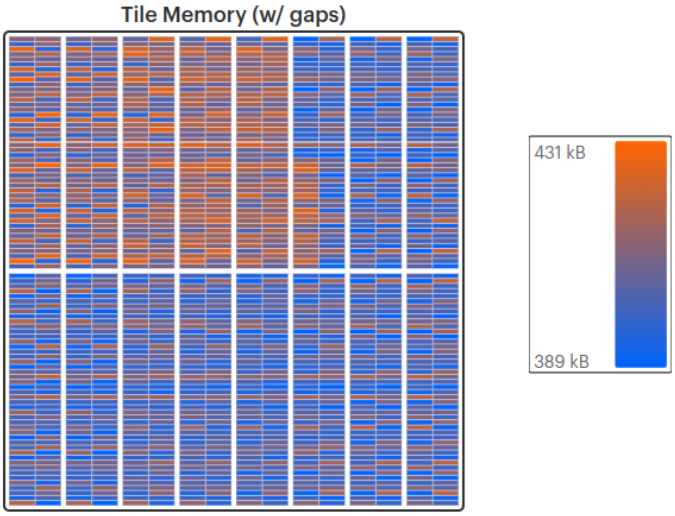


Figure 6.5: Tile Map Butterfly at $N = 2^{12}$

a $60\,\%$ lower difference in relative memory consumption. We assume that this is due to the sparsification by the Butterfly Factorization and the matrix dimensions being a power of two, aligning with the `Butterfly` restrictions. Based on the findings, we do not consider it worthwhile to improve the load balancing, as we do not expect a recognizable performance improvement. As these insights are only applicable to this specific problem size, future work will evaluate if the statements hold true for other input matrix dimensions. As depicted in the previous chapter by Fig.5.1 and Fig.5.4, injecting low-level CUDA code to a PyTorch model can improve the performance of `Butterfly` as the algorithm can be adapted to the underlying architecture. On the IPU, the possibility to inject low-level POPLAR code is given as well via custom ops. These can be created to implement a functionality that is not directly supported in PopTorch [27]. Two steps are necessary to create the custom op in PopTorch: Implementing the op in C++ using the PopART API and making it available in PopTorch [27]. As the previous chapters showed, the properties of the GPU are not applicable to the IPU. To determine if the performance of `Butterfly` can be increased, we first evaluate the overhead of injecting POPLAR Code to a PopTorch model. To make sure that only the overhead of injecting POPLAR code is measured and not the influence of the underlying implementation and as `Butterfly` relies on sparse data, we revert to analyzing sparse MM to (1) compare against chapter 4 and (2) make use of the PopSPARSE library functions. Graphcore provides a benchmark in [36] to measure the performance of PopSPARSE in PopTorch. Unfortunately, due to incompatibilities of our POPLAR version and the one needed to run the benchmarks, these experiments will be included in future work.

7 Conclusion & Future Work

This thesis aimed to evaluate whether the GC200 IPU can effectively replace a GPU with similar power consumption, transistor size, and release date in both High-Performance Computing (HPC) and Machine Learning (ML). To achieve this, we conducted performance evaluations of the IPU using Matrix Matrix Multiplication (MM) and analyzed its interconnect capabilities with point-to-point transfers. The benchmarks initially presented by Jia et al. [39] for the predecessor of the GC200 IPU were extended by hand-written implementations, skewed matrices, and sparse matrices. These results were then compared against the A30 GPU.

It is shown that the interconnect bandwidth is independent to the location of data on the IPU. For the dense squared MM, we demonstrated that the IPU can outperform the GPU by a factor of 4.5 with the respective linear algebra library functions and deactivated Tensor Cores, but it has a more restricted maximum problem size due to the limited amount of In-Processor Memory. When the Tensor Cores are activated, the IPU achieves a $25\,\%$ lower peak performance than the GPU. For the hand-written implementations, we could reach a higher share of the GPU peak performance than for the IPU due to the novelty of the IPU Programming Model. The dense MM revealed a gap between the maximum problem size on the IPU and the overall In-Processor memory. With the help of the PopVision Graph Analyser, this gap was explained by the additional allocated data for the computation such as Internal Exchange Code. Furthermore, since both processing units focus on ML workloads, we compared the performance implications of using PopTorch / PyTorch against low-level implementations of squared dense MM. It was found that, unlike PyTorch, PopTorch exhibits a performance gap as it does not provide the option to manually define data movement.

Regarding skewed MM, we observed that the performance decay is not as symmetrical on the IPU as on the GPU. This was mainly due to the implications of the PopLIN library on the IPU whereas the GPU performance was affected by the data movement from and to DRAM. For most scenarios, more extreme values of s_k resulted in a smaller performance degradation on the IPU than on the GPU.

The sparse MM showed that for both processing units, the performance correlates with the sparsity. In contrast to the GPU, the IPU reached the performance of the dense MM only for a sparsity of $99\,\%$ with $40\,\%$ lower maximum problem size than the dense equivalent due

to the allocation of additional temporary data and the implementation of the PopSPARSE library functions. On the other hand, the GPU could not benefit from the Tensor Cores in the sparse MM but matched the dense MM performance with activated Tensor Cores at $99\,\%$ sparsity.

Due to these findings and the properties of the IPU, we evaluated the performance of `Butterfly` and `Pixelated Butterfly` on the IPU. The approaches were presented by Dao et al. [10] and Chen et al. [6], demonstrating an improvement in terms of model compression and execution time while being able to represent a variety of discrete transforms. We compared the execution times against `torch.nn.Linear` and showed for the GPU that `Butterfly` and `Pixelated Butterfly` could achieve a speedup for large problem sizes with deactivated Tensor Cores. With the CUDA interface provided by [10], `Butterfly` outperformed `torch.nn.Linear` even with activated Tensor Cores. On the IPU, `Butterfly` and `Pixelated Butterfly` outperformed `torch.nn.Linear` for large dimensions with squared input dimensions. `Pixelated Butterfly` had higher memory requirements and a greater number of Compute Sets than `Butterfly` due to the Flat Block Butterfly and the low-rank terms. At a batch size of 2^8, only `Butterfly` outperformed `torch.nn.Linear` on the IPU at $N \geq 2^{11}$.

These insights motivated us to compare `Butterfly` and `Pixelated Butterfly` in terms of parameter count, test accuracy and execution time with other structured sparse, unconstrained and linear layers on the single-hidden layer benchmark presented by Thomas et al. [67] with the MNIST and CIFAR10 dataset. It is shown that `Butterfly` and `Pixelated Butterfly` achieved the best test accuracy of all structured sparse approaches on the CIFAR10 dataset on both, GPU and IPU while compressing the ML model by factor 64.66 and 2.62 respectively. `Pixelated Butterfly` did not work with the MNIST dataset due to implementation restrictions. `Butterfly` achieved the best test accuracy among the structured sparse approaches on both processing units for this dataset. As `Pixelated Butterfly` is designed for the GPU and as we used the same configuration for both processing units, `Pixelated Butterfly` outperformed `Butterfly` by factor 1.2 on the GPU whereas it was the other way round for the IPU with `Butterfly` being $1.9\times$ faster than `Pixelated Butterfly`. `Butterfly` performed worse than the linear layer in terms of test accuracy and execution time for the single-hidden layer benchmark on both processing units due to the additional work caused by the Butterfly Factorization. The IPU was able to run the benchmark with half the execution time for the linear layer.

Due to these findings, we investigated on possible optimizations of both Butterfly Approximations on the IPU. For `Pixelated Butterfly`, it is found that for the single-hidden

layer benchmark with the CIFAR10 dataset, we could achieve an accuracy increase of 2.3, a decrease in execution time of $42.91\,\mathrm{s}$ and in the parameter count of $351\,032$ by varying the configuration parameters block size, low rank size and butterfly size. We extended the findings by the maximum impact generated by each individual configuration parameter.
For `Butterfly`, it is found that the load balancing, being the most important factor for the IPU performance, is more uniform than for the dense MM at squared matrix dimensions of $N = 2^{12}$.
In summary, we state that the IPU can indeed act as a replacement for the GPU depending on the given workload with the main bottleneck being the limited In-Processor Memory.

The insights gained from this work have rasied several subjects for future research. The limited amount of available memory is a major focus of future work, and three approaches will be presented:
(1) Decreasing the memory requirements by optimizing and adapting workloads, such as MM, on the IPU
(2) Extending the available memory by using Streaming Memory and multiple IPUs
(3) Tweaking the `Available Memory Proportion` [20] parameter to improve the ratio of Tensors and temporal data.
A thorough analysis of the publicly available PopLIBS implementation will be conducted as a basis for optimizing MM. This includes examining profiling reports generated with the PopVision Graph Analyser.
Our work had a focus on Butterfly Approximations on GPU and IPU. As ML workloads can also be run on a CPU, future work will evaluate and analyze the performance of both approaches on this processing unit.
For `Pixelated Butterfly`, we relied on the implementation presented in [2] which was written in plain PyTorch to exclude performance implications of other frameworks and to include it in PopTorch with as few changes as possible. Future work will examine the implementation from [6].
As we restricted ourselves to a single processing unit to allow a fair comparison between IPU and GPU, we could not run any experiment presented in [6] because of memory constraints. As both Butterfly Approximations aim at larger ML models, future work will include these workloads by running them on multiple processing units.
The source code of `Butterfly` enables the use of CUDA in a given PyTorch model. We were able to run process larger problem sizes with this implementation than with the ones for `Butterfly` and `Pixelated Butterfly` written in plain PyTorch. Future work will analyze this issue.

Due to problems with the Slurm plugin and constraints of PopTorch, we could only run the PopVision Graph Analyzer with a single iteration and included the data movements. Future work will exclude the data movements and present results with an increased number of iterations to stabilize the results.
The single-hidden layer benchmark presented a variety of structured sparse approaches. Future work will extend these with other layers such as `Deformable Butterfly` presented by Lin et al. [48] and analyze them. On this benchmark, the low-rank approach performed worse in terms of test accuracy than what the authors of [10] presented. Future work will analyze and overcome this issue. We will add more datasets to this benchmark to gain more general insights.
In contrast to our expectations, an increased block size led to a better performance for `Pixelated Butterfly` on the IPU. This will be analyzed with the help of the profiling tools in future work.
To optimize `Butterfly` on the IPU, we evaluated the load balancing for a given squared matrix input dimension of 2^{12} to allow a comparison with previous chapters. Future work will include an analysis for different matrix dimensions. In addition, the injection of POPLAR code to a PopTorch model to improve performance will be evaluated with the help of the benchmark presented in [36]. In case POPLAR code injection to a PopTorch model increases performance, future work will implement `Butterfly` in POPLAR with the CUDA code of [10] as a basis.

A Appendix

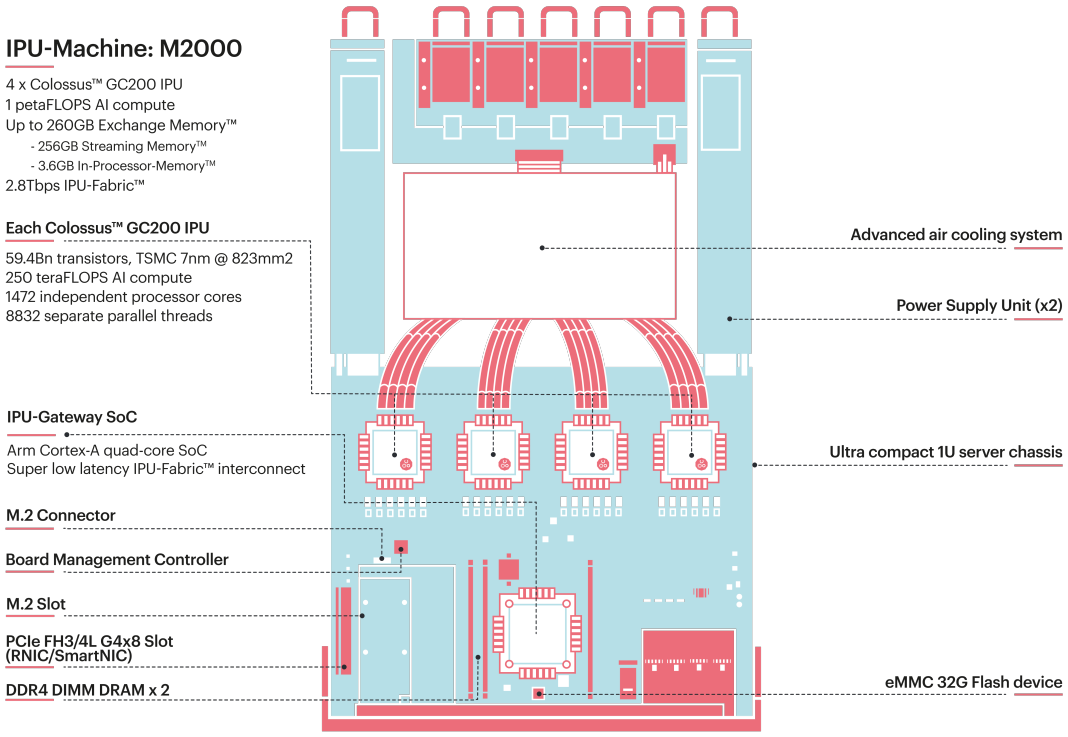


Figure A.1: M2000 IPU-Machine consisting of four GC200 IPUs [33]

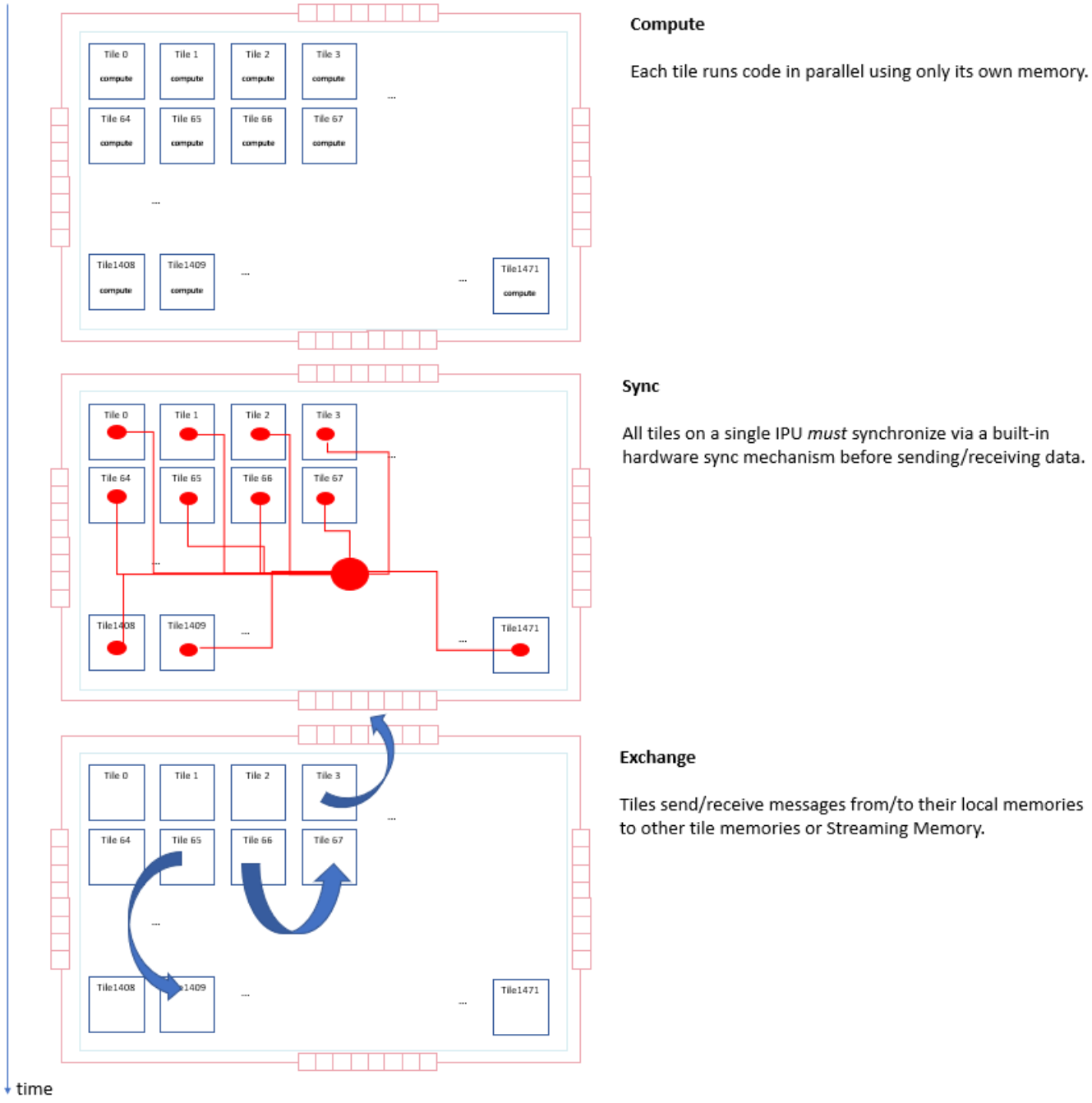


Figure A.2: BSP Execution of the IPU [26]

m	n	k	Kernel sequence
1	65536	256	Convolve, Reduce0, Reduce1, InstrumentationDumpStep, InstrumentationDumpDecrement
2	32768	256	Convolve, Reduce0, Reduce1, InstrumentationDumpStep, InstrumentationDumpDecrement, PostArrange
4	16384	256	Convolve, Reduce0, Reduce1, Reduce2, InstrumentationDumpStep, InstrumentationDumpDecrement, PreArrange
8	8192	256	Convolve, Reduce0, Reduce1, Reduce2, InstrumentationDumpStep, InstrumentationDumpDecrement
16	4096	256	Convolve, Reduce0, Reduce1, Reduce2, InstrumentationDumpStep, InstrumentationDumpDecrement
32	2048	256	Convolve, Reduce, Reduce1, InstrumentationDumpStep, InstrumentationDumpDecrement
64	1024	256	Convolve, Reduce, Reduce1, InstrumentationDumpStep, InstrumentationDumpDecrement
128	512	256	Convolve, Reduce0, InstrumentationDumpStep, InstrumentationDumpDecrement
256	256	256	Convolve, Reduce0, InstrumentationDumpStep, InstrumentationDumpDecrement
512	128	256	Convolve, Reduce0, InstrumentationDumpStep, InstrumentationDumpDecrement
1024	64	256	Convolve, Reduce0, InstrumentationDumpStep, InstrumentationDumpDecrement
2048	32	256	Convolve, InstrumentationDumpStep, InstrumentationDumpDecrement
4096	16	256	Convolve, InstrumentationDumpStep, InstrumentationDumpDecrement
8192	8	256	Convolve, InstrumentationDumpStep, InstrumentationDumpDecrement
16384	4	256	Convolve, InstrumentationDumpStep, InstrumentationDumpDecrement, OnTileCopy, PreArrange
32768	2	256	Convolve, InstrumentationDumpStep, InstrumentationDumpDecrement, OnTileCop, OnTileCopy, PreArrange
65536	1	256	Convolve, Acts, Weights, InstrumentationDumpStep, InstrumentationDumpDecrement, OnTileCopy, OnTileCopy, OnTileCopy, OnTileCopy, PreArrange

Table A.1: Kernel Calls ipu-poplin2 @k=256

m	n	k	Kernel sequence
1	1048576	1024	void cutlass::Kernel<cutlass_80_tensorop_s1688gemm_128x64_32x6_nn_align4>()
2	524288	1024	void cutlass::Kernel<cutlass_80_tensorop_s1688gemm_128x64_32x6_nn_align4>()
4	262144	1024	void cutlass::Kernel<cutlass_80_tensorop_s1688gemm_128x64_32x6_nn_align4>()
8	131072	1024	void cutlass::Kernel<cutlass_80_tensorop_s1688gemm_256x64_32x3_nn_align4>()
16	65536	1024	void cutlass::Kernel<cutlass_80_tensorop_s1688gemm_256x64_32x3_nn_align4>()
32	32768	1024	void cutlass::Kernel<cutlass_80_tensorop_s1688gemm_256x64_32x3_nn_align4>()
64	16384	1024	void cutlass::Kernel<cutlass_80_tensorop_s1688gemm_128x128_32x5_nn_align4>()
128	8192	1024	void cutlass::Kernel<cutlass_80_tensorop_s1688gemm_128x128_32x3_nn_align4>()
256	4096	1024	void cutlass::Kernel<cutlass_80_tensorop_s1688gemm_128x128_32x3_nn_align4>()
512	2048	1024	void cutlass::Kernel<cutlass_80_tensorop_s1688gemm_128x128_32x3_nn_align4>()
1024	1024	1024	void cutlass::Kernel<cutlass_80_tensorop_s1688gemm_128x64_16x6_nn_align4>()
2048	512	1024	void cutlass::Kernel<cutlass_80_tensorop_s1688gemm_128x128_16x5_nn_align4>()
4096	256	1024	void cutlass::Kernel<cutlass_80_tensorop_s1688gemm_128x128_16x5_nn_align4>()
8192	128	1024	void cutlass::Kernel<cutlass_80_tensorop_s1688gemm_256x64_16x4_nn_align4>()
16384	64	1024	void cutlass::Kernel<cutlass_80_tensorop_s1688gemm_256x64_16x4_nn_align4>()
32768	32	1024	void cutlass::Kernel<cutlass_80_tensorop_s1688gemm_256x64_16x4_nn_align4>()
65536	16	1024	ampere_sgemm_64x64_nn
131072	8	1024	ampere_sgemm_64x64_nn
262144	4	1024	ampere_sgemm_64x64_nn
524288	2	1024	ampere_sgemm_64x64_nn
1048576	1	1024	ampere_sgemm_64x64_nn

Table A.2: Kernel Calls gpu-cublas (TF32) @k = 1024

m	n	k	Kernel sequence
1	1048576	1024	void gemv2N_kernel<int, int, float, float, float, 128, 4, 4, 4, 1, false, cublasGemvParams<cublasGemvTensorStridedBatched<float const>, cublasGemvTensorStridedBatched<float>, float> >()
2	524288	1024	void scal_kernel<float, float, 1, true, 6, 5, 5, 3>(cublasTransposeParams<float>, float const*, float*, float const*)
4	262144	1024	void scal_kernel<float, float, 1, true, 6, 5, 5, 3>(cublasTransposeParams<float>, float const*, float*, float const*)
8	131072	1024	void scal_kernel<float, float, 1, true, 6, 5, 5, 3>(cublasTransposeParams<float>, float const*, float*, float const*)
16	65536	1024	ampere_sgemm_128x32_sliced1x4_nn
32	32768	1024	ampere_sgemm_64x32_sliced1x4_nn
64	16384	1024	ampere_sgemm_32x32_sliced1x4_nn
128	8192	1024	ampere_sgemm_64x32_sliced1x4_nn
256	4096	1024	ampere_sgemm_128x64_nn
512	2048	1024	ampere_sgemm_64x32_sliced1x4_nn
1024	1024	1024	ampere_sgemm_128x64_nn
2048	512	1024	ampere_sgemm_64x32_sliced1x4_nn
4096	256	1024	ampere_sgemm_64x64_nn
8192	128	1024	ampere_sgemm_128x32_nn
16384	64	1024	ampere_sgemm_128x32_nn
32768	32	1024	ampere_sgemm_128x64_nn
65536	16	1024	ampere_sgemm_64x64_nn
131072	8	1024	ampere_sgemm_64x64_nn
262144	4	1024	ampere_sgemm_64x64_nn
524288	2	1024	ampere_sgemm_64x64_nn
1048576	1	1024	ampere_sgemm_128x64_nn

Table A.3: Kernel Calls gpu-cublas (FP32) @k = 1024

Bibliography

[1] Mnist-noise and mnist-bg-rot datasets. http://www.iro.umontreal.ca/~lisa/twiki/bin/view.cgi/Public/DeepVsShallowComparisonICML2007. [Accessed 26-Jul-2023].

[2] Pure torch implementaton of pixelfly (for tpus, cpus and custom blocks). https://gist.github.com/justheuristic/9e4fb81381451a4bc8cbfee0a5100eba. [Accessed 26-Jul-2023].

[3] Techpowerup - nvidia a30 pcie. https://www.techpowerup.com/gpu-specs/a30-pcie.c3792, 2023. [Accessed 26-Jul-2023].

[4] Luk Burchard, Xing Cai, and Johannes Langguth. ipug for multiple graphcore ipus: Optimizing performance and scalability of parallel breadth-first search. In *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 162–171. IEEE, 2021.

[5] Shuai Che, Jeremy W Sheaffer, Michael Boyer, Lukasz G Szafaryn, Liang Wang, and Kevin Skadron. A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. In *IEEE International Symposium on Workload Characterization (IISWC'10)*, pages 1–11. IEEE, 2010.

[6] Beidi Chen, Tri Dao, Kaizhao Liang, Jiaming Yang, Zhao Song, Atri Rudra, and Christopher Re. Pixelated butterfly: Simple and efficient sparse training for neural network models. *arXiv preprint arXiv:2112.00029*, 2021.

[7] Krzysztof Choromanski, Mark Rowland, Wenyu Chen, and Adrian Weller. Unifying orthogonal monte carlo methods. In *International Conference on Machine Learning*, pages 1203–1212. PMLR, 2019.

[8] Tri Dao, Beidi Chen, Nimit S Sohoni, Arjun Desai, Michael Poli, Jessica Grogan, Alexander Liu, Aniruddh Rao, Atri Rudra, and Christopher Ré. Monarch: Expressive structured matrices for efficient and accurate training. In *International Conference on Machine Learning*, pages 4690–4721. PMLR, 2022.

[9] Tri Dao, Albert Gu, Matthew Eichhorn, Atri Rudra, and Christopher Re. Butterfly matrix multiplication in pytorch. https://github.com/HazyResearch/butterfly. [Accessed 26-Jul-2023].

[10] Tri Dao, Albert Gu, Matthew Eichhorn, Atri Rudra, and Christopher Re. Learning fast algorithms for linear transforms using butterfly factorizations. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 1517–1527. PMLR, 09–15 Jun 2019.

[11] Tri Dao, Nimit S Sohoni, Albert Gu, Matthew Eichhorn, Amit Blonder, Megan Leszczynski, Atri Rudra, and Christopher Ré. Kaleidoscope: An efficient, learnable representation for all structured linear maps. *arXiv preprint arXiv:2012.14966*, 2020.

[12] Anastasia Dietrich, Frithjof Gressmann, Douglas Orr, Ivan Chelombiev, Daniel Justus, and Carlo Luschi. Towards structured dynamic sparse pre-training of bert, 2021.

[13] Kaivalya M Dixit. The spec benchmarks. *Parallel computing*, 17(10-11):1195–1209, 1991.

[14] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J R Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, David Silver, Demis Hassabis, and Pushmeet Kohli. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47—53, October 2022.

[15] Graphcore. popsparse/matmul.cpp. `https://github.com/graphcore/poplibs/blob/sdk-release-3.2/lib/popsparse/MatMul.cpp`. [Accessed 26-Jul-2023].

[16] Graphcore. Tile vertex instruction set architecture. `https://docs.graphcore.ai/projects/isa/en/latest/index.htmll`. [Accessed 26-Jul-2023].

[17] Graphcore. The graphcore second generation ipu. `https://www.graphcore.ai/hubfs/MK2-%20The%20Graphcore%202nd%20Generation%20IPU%20Final%20v7.14.2020.pdf?hsLang=en`, 2020. [Accessed 26-Jul-2023].

[18] Graphcore. Poplibs api reference. `https://docs.graphcore.ai/projects/poplar-api/en/latest/poplibs_api.html`, 2020. [Accessed 27-07-2023].

[19] Graphcore. Sparse tensor operations (popsparse), sparsetensor. `https://docs.graphcore.ai/projects/poplar-api/en/latest/poplibs/popsparse/SparseTensor.html`, 2020. [Accessed 26-Jul-2023].

[20] Graphcore. Introduction to the available memory proportion option. `https://docs.graphcore.ai/projects/available-memory/en/latest/available-memory.html`, 2021. [Accessed 26-Jul-2023].

[21] Graphcore. Memory and performance optimisation on the ipu: Mapping a model to an ipu system. `https://docs.graphcore.ai/projects/memory-performance-optimisation/en/latest/map-model-to-ipu-system.html`, 2021. [Accessed 26-Jul-2023].

[22] Graphcore. Popsparse matrix multiplication (dynamic pattern) on the ipu, known limitations. `https://docs.graphcore.ai/projects/dynamic-sparsity/en/latest/dynamic-sparsity.html#known-limitations`, 2021. [Accessed 26-Jul-2023].

[23] Graphcore. Popvision analysis library (libpva) user guide. `https://docs.graphcore.ai/projects/libpva/en/latest/index.html`, 2021. [Accessed 27-07-2023].

[24] Graphcore. Ipu m2000 ipu-machine datasheet. https://docs.graphcore.ai/projects/graphcore-ipu-m2000-datasheet/en/latest/, 2022. [Accessed 26-Jul-2023].

[25] Graphcore. Ipu programmer's guide. https://docs.graphcore.ai/projects/ipu-programmers-guide/en/latest/, 2022. [Accessed 26-Jul-2023].

[26] Graphcore. Ipu programmer's guide, host/device communication. https://docs.graphcore.ai/projects/ipu-programmers-guide/en/latest/about_ipu.html, 2022. [Accessed 26-Jul-2023].

[27] Graphcore. Pytorch for the ipu: User guide. https://docs.graphcore.ai/projects/poptorch-user-guide/en/latest/index.html, 2022. [Accessed 26-Jul-2023].

[28] Graphcore. Pytorch for the ipu: User guide - installation. https://docs.graphcore.ai/projects/poptorch-user-guide/en/latest/installation.html, 2022. [Accessed 26-Jul-2023].

[29] Graphcore. Pytorch (poptorch) mnist training demo. https://github.com/graphcore/examples/tree/v3.2.0/tutorials/simple_applications/pytorch/mnist, 2022. [Accessed 26-Jul-2023].

[30] Graphcore. Bow ipu processors. https://www.graphcore.ai/bow-processors, 2023. [Accessed 26-Jul-2023].

[31] Graphcore. C600 ipu-processor pcie card. https://www.graphcore.ai/products/c600, 2023. [Accessed 26-Jul-2023].

[32] Graphcore. Ipu processors. https://www.graphcore.ai/products/ipu, 2023. [Accessed 26-Jul-2023].

[33] Graphcore. Next generation ipu systems ipu-m2000 + ipu-pod4. https://www.graphcore.ai/products/mk2/ipu-m2000-ipu-pod4, 2023. [Accessed 26-Jul-2023].

[34] Graphcore. Poplar graph framework software. https://www.graphcore.ai/products/poplar, 2023. [Accessed 26-Jul-2023].

[35] Graphcore. Popvision graph analyser user guide. https://docs.graphcore.ai/projects/graph-analyser-userguide/en/latest/index.html, 2023. [Accessed 26-Jul-2023].

[36] Graphcore. Sparse benchmark spmm. https://github.com/graphcore-research/poptorch-experimental-addons/blob/main/examples/sparse_benchmark_spmm.py, 2023. [Accessed 26-Jul-2023].

[37] R. Hunger. *Floating Point Operations in Matrix-vector Calculus*. Munich University of Technology, Inst. for Circuit Theory and Signal Processing, 2005.

[38] Pablo Villalobos Jaime Sevilla and Juan Felipe Cerón. Parameter count in machine learning. `https://www.lesswrong.com/posts/GzoWcYibWYwJva8aL/parameter-counts-in-machine-learning`, 2021. [Accessed 26-Jul-2023].

[39] Zhe Jia, Blake Tillman, Marco Maggioni, and Daniele Paolo Scarpazza. Dissecting the graphcore ipu architecture via microbenchmarking. *arXiv preprint arXiv:1912.03413*, 2019.

[40] Li Jing, Yichen Shen, Tena Dubcek, John Peurifoy, Scott Skirlo, Yann LeCun, Max Tegmark, and Marin Soljačić. Tunable efficient unitary neural networks (eunn) and their application to rnns. In *International Conference on Machine Learning*, pages 1733–1741. PMLR, 2017.

[41] Ilyes Kacher, Maxime Portaz, Hicham Randrianarivo, and Sylvain Peyronnet. Graphcore c2 card performance for image-based deep learning application: A report, 2020.

[42] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[43] Dave Lacey. Intelligent memory for intelligent computing. `https://www.graphcore.ai/posts/intelligent-memory-for-intelligent-computing`, 2020. [Accessed 26-Jul-2023].

[44] Hugo Larochelle, Dumitru Erhan, Aaron Courville, James Bergstra, and Yoshua Bengio. An empirical evaluation of deep architectures on problems with many factors of variation. In *Proceedings of the 24th international conference on Machine learning*, pages 473–480, 2007.

[45] Burges LeCun, Cortes. The mnist database of handwritten digits. `http://yann.lecun.com/exdb/mnist/`. [Accessed 26-Jul-2023].

[46] Yingzhou Li, Haizhao Yang, Eileen R Martin, Kenneth L Ho, and Lexing Ying. Butterfly factorization. *Multiscale Modeling & Simulation*, 13(2):714–732, 2015.

[47] GNU Scientific Library. Sparse matrix storage formats. `https://www.gnu.org/software/gsl/doc/html/spmatrix.html#sparse-matrix-storage-formats`, 2021. [Accessed 26-Jul-2023].

[48] Rui Lin, Jie Ran, King Hung Chiu, Graziano Chesi, and Ngai Wong. Deformable butterfly: A highly structured and sparse linear transform. *Advances in Neural Information Processing Systems*, 34:16145–16157, 2021.

[49] Thorben Louw and Simon McIntosh-Smith. Using the graphcore ipu for traditional hpc applications. In *3rd Workshop on Accelerated Machine Learning (AccML)*, 2021.

[50] Michael Mathieu and Yann LeCun. Fast approximation of rotations and hessians matrices. *arXiv preprint arXiv:1404.7195*, 2014.

[51] Eric Michielssen and Amir Boag. A multilevel matrix decomposition algorithm for analyzing scattering from large structures. *IEEE Transactions on Antennas and Propagation*, 44(8):1086–1093, 1996.

[52] Lakshan Ram Madhan Mohan, Alexander Marshall, Samuel Maddrell-Mander, Daniel O'Hanlon, Konstantinos Petridis, Jonas Rademacker, Victoria Rege, and Alexander Titterton. Studying the potential of graphcore ipus for applications in particle physics. *arXiv preprint arXiv:2008.09210*, 2020.

[53] Samuel K. Moore. Graphcore uses tsmc 3d chip tech to speed ai by 40 percent. `https://spectrum.ieee.org/graphcore-ai-processor#toggle-gdpr`, 2022. [Accessed 26-Jul-2023].

[54] Marina Munkhoeva, Yermek Kapushev, Evgeny Burnaev, and Ivan Oseledets. Quadrature-based features for kernel approximation. *Advances in neural information processing systems*, 31, 2018.

[55] NVIDIA. Cuda samples sdk reference manual v7.0, 2015.

[56] NVIDIA. Nvidia ampere architecture whitepaper. `https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf`, 2020. [Accessed 26-Jul-2023].

[57] NVIDIA. A30 tensor core gpu. `https://www.nvidia.com/en-us/data-center/products/a30-gpu/`, 2023. [Accessed 26-Jul-2023].

[58] NVIDIA. Api reference guide for cusparse, the cuda sparse matrix library. `https://docs.nvidia.com/cuda/cusparse/index.html`, 2023. [Accessed 26-Jul-2023].

[59] NVIDIA. C++ best practices guide 2023. `https://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf`, 2023. [Accessed 26-Jul-2023].

[60] Michael O'Neil, Franco Woolfe, and Vladimir Rokhlin. An algorithm for the rapid evaluation of special function transforms. *Applied and Computational Harmonic Analysis*, 28(2):203–226, 2010.

[61] Rukshan Pramoditha. One hidden layer (shallow) neural network architecture. `https://medium.com/data-science-365/one-hidden-layer-shallow-neural-network-architecture-d45097f649e6`, 2021. [Accessed 26-Jul-2023].

[62] Yuliya Pylypiv. Basic mnist example. `https://github.com/pytorch/examples/tree/main/mnist`. [Accessed 26-Jul-2023].

[63] Syed M Qasim, Ahmed A Telba, and Abdulhameed Y AlMazroo. Fpga design and implementation of matrix multiplier architectures for image and signal processing applications. *International Journal of Computer Science and Network Security*, 10(2):168–176, 2010.

[64] Kevin Stehle, Günther Schindler, and Holger Fröning. On the difficulty of designing processor arrays for deep neural networks, 2020.

[65] Nupur Sumeet, Karan Rawat, and Manoj Nambiar. Performance evaluation of graphcore ipu-m2000 accelerator for text detection application. In *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering*, pages 145–152, 2022.

[66] Yi Tay, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, Sebastian Ruder, and Donald Metzler. Long range arena: A benchmark for efficient transformers. *arXiv preprint arXiv:2011.04006*, 2020.

[67] Anna Thomas, Albert Gu, Tri Dao, Atri Rudra, and Christopher Ré. Learning compressed transforms with low displacement rank. *Advances in neural information processing systems*, 31, 2018.

[68] Tiffany Trader. Graphcore readies launch of 16nm colossus-ipu chip. `https://www.hpcwire.com/2017/07/20/graphcore-readies-launch-16nm-colossus-ipu-chip/`, 2017. [Accessed 26-Jul-2023].

[69] Tri Dao, Albert Gu, Matthew Eichhorn, Megan Leszczynski, Nimit Sohoni, Amit Blonder, Atri Rudra, and Chris Ré. Butterflies are all you need: A universal building block for structured linear maps. `https://dawn.cs.stanford.edu/2019/06/13/butterfly/`. [Accessed 26-Jul-2023].

[70] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[71] Kapil Vaswani, Stavros Volos, Cédric Fournet, Antonio Nino Diaz, Ken Gordon, Balaji Vembu, Sam Webster, David Chisnall, Saurabh Kulkarni, Graham Cunningham, et al. Confidential machine learning within graphcore ipus. *arXiv preprint arXiv:2205.09005*, 2022.

[72] Di Yan, Tao Wu, Ying Liu, and Yang Gao. An efficient sparse-dense matrix multiplication on a multicore system. In *2017 IEEE 17th International Conference on Communication Technology (ICCT)*, pages 1880–1883. IEEE, 2017.

Ehrenwörtliche Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit mit dem Titel "*On the Performance of Butterfly Approximations on the Graphcore IPU*" selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Declaration of Originality

I confirm that the submitted work is original work and was written by me without further assistance. Appropriate credit has been given where reference has been made to the work of others. The work was not examined before, nor has it been published. The submitted electronic version of the work matches the printed version

Ort, Datum
Place, Date

Christian Alles