POPSPARSE: ACCELERATED BLOCK SPARSE MATRIX MULTIPLICATION ON IPU

Zhiyi Li[*], Douglas Orr[*], Valeriu Ohan[*], Godfrey Da costa, Tom Murray, Adam Sanders, Deniz Beker & Dominic Masters
Graphcore, UK
{zhiyil, douglaso, valeriuo}@graphcore.ai

ABSTRACT

Reducing the computational cost of running large scale neural networks using sparsity has attracted great attention in the deep learning community. While much success has been achieved in reducing FLOP and parameter counts while maintaining acceptable task performance, achieving actual speed improvements has typically been much more difficult, particularly on general purpose accelerators (GPAs) such as NVIDIA GPUs using low precision number formats. In this work we introduce PopSparse, a library that enables fast sparse operations on Graphcore IPUs by leveraging both the unique hardware characteristics of IPUs as well as any block structure defined in the data. We target two different types of sparsity: static, where the sparsity pattern is fixed at compile-time; and dynamic, where it can change each time the model is run. We present benchmark results for matrix multiplication for both of these modes on IPU with a range of block sizes, matrix sizes and densities. Results indicate that the PopSparse implementations are faster than dense matrix multiplications on IPU at a range of sparsity levels with large matrix size and block size. Furthermore, static sparsity in general outperforms dynamic sparsity. While previous work on GPAs has shown speedups only for very high sparsity (typically 99% and above), the present work demonstrates that our static sparse implementation outperforms equivalent dense calculations in FP16 at lower sparsity (around 90%). IPU code is available to view and run at `ipu.dev/sparsity-benchmarks`, GPU code will be made available shortly.

1 INTRODUCTION

The topic of sparsity has gained significant attention in the field of deep learning research due to its potential for increased computational efficiency, reduced model size, and closer alignment with brain-like computation. The notion of sparsity in deep learning most commonly refers to the idea of sparsifying the model weights with the aim of reducing the associated storage and compute costs. This is typically done through a single pruning step at the end of training (Zhu & Gupta, 2017) or potentially through some sparse training regime (Evci et al., 2019). These approaches have typically achieved between 90-99% reduction in model weights while maintaining and acceptable level of accuracy depending on the model and technique used (Hoefler et al., 2021).

While the benefits to model size can be easily realised, increased computational efficiency can often be more difficult to achieve as sparse computation can be challenging to execute effectively on modern, highly parallel deep learning accelerators (Qin et al., 2022). As such, sparse training methods often fall into a gap between theoretical and practical efficiency. For example Mostafa & Wang (2019) demonstrated FLOP efficient deep residual CNN training with dynamic sparse reparameterisation techniques but struggled to achieve speed ups in practice. A key reason behind this is that unstructured sparsity patterns do not align well with the vectorised instruction sets that have been so successful in accelerating highly parallel operations. Therefore, one approach to improving sparse compute efficiency is to enforce some degree of structure on the sparsity pattern to better align it with the hardware architecture. This has motivated a wide range of *structured* pruning techniques such as neuron (Ebrahimi & Klabjan, 2021), channel-wise (He et al., 2017), block (Gray et al., 2017), 2:4 (Mishra et al., 2021). These methods however almost always pay some task performance penalty compared to equivalent unstructured methods leading to a trade-off between model size and execution speed which is still not guaranteed to achieve practical performance benefits.

[*]equal contriubtion

In this work we present PopSparse, a library for the effective acceleration of sparse matrix multiplication (matmul) on Graphcore IPUs (Graphcore, 2022b). IPUs have a number of architectural features that help accelerate sparse operations, firstly they have a large amount of high bandwidth on-chip SRAM which drastically improves performance for communication heavy, low arithmetic efficiency operations. This is critical for sparse operations as the increased ratio of communication to compute is a natural consequence of weight sparsification. Furthermore, they use a fine grained MIMD processing model that partitions work over 1472 independent compute units, here on described as *tiles*. This means high degrees of parallelism can be exploited from operations even if they cannot be structured as a single large matrix multiplication. Finally, the IPU uses an ahead-of-time compilation model that allows further optimisations to be employed if computational details, like the structure of the sparsity, are known at compile time. PopSparse leverages these advantages to accelerate sparse matrix multiplications on IPU and allows further performance improvements to be realised by enforcing algorithmic constraints to the block size and ensuring sparsity patterns are available at compile time.

We benchmark PopSpare performance on sparse-dense matmul (SpMM) across a range of problem specifications and compare performance to dense matmul implementations on IPU as well as GPU baselines.

1.1 CONTRIBUTIONS

The present work makes the following contributions:

- We introduce the PopSparse library for sparse tensor operations on IPU. The library supports two flavours of sparse operations: static, where the sparsity pattern for the sparse operand is fixed; and dynamic, where sparsity pattern for the sparse operand is updated at runtime.
- We compare benchmark results for static and dynamic sparse modes with a single sparse-dense SpMM against the dense implementation on IPU and both sparse (cuSPARSE™, NVIDIA (2022b)) and dense (cuBLAS™, NVIDIA (2022a)) implementations on GPU, showing that PopSparse static sparsity outperforms competitive dense implementations in FP16 with low sparsity (ranges from 97 $\%$ to 88 $\%$, depending on the matrix size and block size).
- Reviewing the benchmark results, we answer the questions: 1) Under what conditions do we expect to benefit from sparse operations on IPU? and 2) How well does IPU performance compare to GPU in sparse operations?

1.2 RELATED WORK

Various algorithms have been proposed for SpMM acceleration on GPUs. Yang et al. (2018) introduced novel algorithms (MergeSpMM) for sparse-dense matrix multiplication on the GPU with merge-based load balancing and row-major coalesced memory access. Hong et al. (2019) proposed an adaptive sparse tiling (ASpT) technique and used it to enhance the performance of SpMM. They used intra-row reordering to enable adaptive tiling. Gale et al. (2020) designed high-performance SpMM kernels targeted specifically at deep learning applications.

Using SpMM as the fundamental component, pruning has been used to optimally sparsify the dense weight matrix in deep neural networks. Zhu & Gupta (2017) pruned the large models in an unstructured way and compared the accuracy of the models with their small-dense counterparts across a broad range of neural network architectures (CNN, LSTM etc.) and found that large-sparse models to consistently outperform small-dense models and achieve up to 10x reduction in number of non-zero parameters. Yao et al. (2019) showed a novel fine-grained unstructured sparsity approach and corresponding pruning method to fixing the number of non-zeros in small regions of the sparse matrix in matrix row. Wang (2020) presented SparseRT, a system to support efficient inference with unstructured weight pruning on GPU.

However, unconstrained magnitude pruning resulting in unstructured sparsity patterns is challenging to support on traditional hardware accelerators (Narang et al., 2017; Yao et al., 2019; Wang, 2020). This discovery leads to structured pruning methods that results in block sparsity. Anwar et al. (2015) introduced structured sparsity at various scales (channel wise, kernel wise and intra kernel strided sparsity) for convolutional neural networks. Focusing on CNN area, a new channel pruning method of structured simplification to accelerate very deep convolutional neural networks is introduced by He et al. (2017). Gray et al. (2017) developed highly optimized GPU kernels for neural network training and inference with block sparsity. Lin et al. (2019) proposed structure sparsity regularized (SSR) filter pruning scheme to speedup the computation and reduce the memory overhead of CNNs and it was deployed to a variety of state of the art CNN structures. Mixture-of-Experts (MoE) training computation in terms of block-

sparse operations was employed by Gale et al. (2022) and they also developed new block-sparse GPU kernels that efficiently handle the dynamism present in MoEs.

2 IPU ARCHITECTURE

Intelligence Processing Unit (IPU) is an accelerator for machine intelligence, made up of 1,472 independent compute cores called *tiles* which co-locate hardware compute logic with local SRAM. Each tile has 625KB of local on-chip SRAM memory, leading to 900MB in total for one IPU. High-bandwidth, low-latency communication among the tiles is achieved through an all-to-all exchange fabric. The IPU supports a bulk synchronous parallel (BSP) execution model where each tile first does **compute** with the data available on its local SRAM followed by a stage **synchronizing** with other tiles across all IPUs, finally **exchanging** data with other tiles on the same IPU or on a different IPU (Helal et al., 2022). The efficient communication, fine-grained independent processing and high level of parallelism are the key factors that enable fast sparse operations.

Programs run over a set of IPUs and follow a path of specified control flow. They manipulate variables just like standard programs on a CPU. The variables being manipulated are large arrays of data (tensors). The programs manipulate these variables with a set of highly parallel tasks (called vertices) executed on the threads of the tile processors. These sets of tasks are known as compute sets (Graphcore, 2022d).

The vertices from the multiple compute sets, and the data they manipulate in the programs form a computational graph. The computational graph, which we call Poplar (Graphcore, 2022e) graph, holds the relationship between compute sets and data. The computational graph gets compiled to binary code using Poplar compilation tools on the host and then gets loaded onto the IPU. The duration of converting the computational graph to binary code is called the compile time. Once the programs are loaded and the IPU configured, a host can then instruct the IPUs to run these programs. This is defined as runtime.

3 POPSPARSE IMPLEMENTATION

PopSparse and the equivalent GPU baselines implement multiple sparse operations, however our focus will be on a sparse-dense matmul (SpMM), written:

$$Y = (M \odot W) * X \,,$$
$$M_{ij} = \hat{M}_{\lfloor i/b \rfloor, \lfloor j/b \rfloor} \,,$$

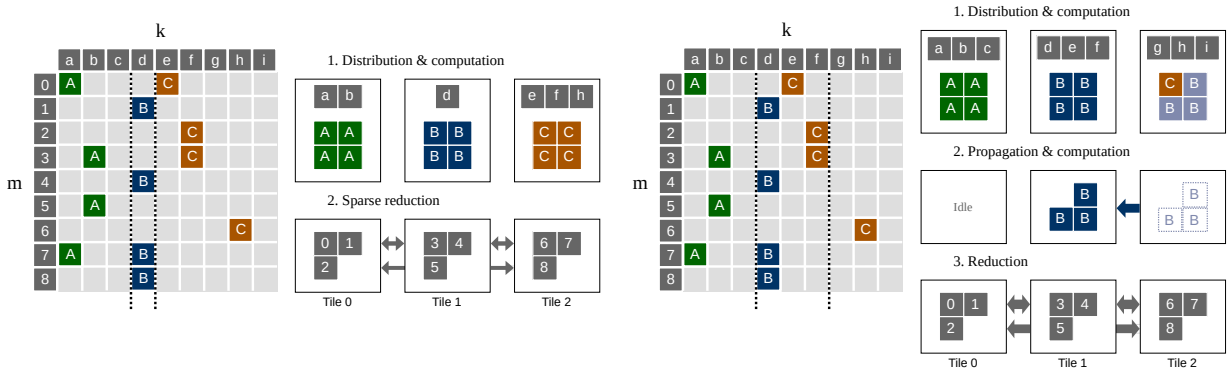
where $\odot$ denotes elementwise multiplication, $*$ for inner product, $Y \in \mathbb{R}^{m \times n}$, $X \in \mathbb{R}^{k \times n}$ are dense output and input matrices respectively. The sparse weight matrix $(M \odot W)$ is defined via $M \in \mathbb{B}^{m \times k}$ ($\mathbb{B} = \{0, 1\}$), a mask that represents the *sparsity pattern*, itself derived from $\hat{M} \in \mathbb{B}^{\lceil m/b \rceil \times \lceil k/b \rceil}$, a block mask and $W \in \mathbb{R}^{m \times k}$ defines weight values.

In this formulation, $(M \odot W)$ has a block-sparse structure, where contiguous square blocks of weights of shape $b \times b$ contain all the non-zero elements, given *block-size* b. The dimensions m and k are referred to as output and input *feature size* and n is referred to as *batch size*, corresponding to their typical role in weight-sparse neural network computation.

We define *density*, $d = \sum_{ij} M_{ij}/(m \cdot k)$, where the standard term *sparsity* corresponds to $(1 - d)$. We use floating point operation (FLOP) count to quantify the amount of useful arithmetic work in an SpMM as: $2 \cdot m \cdot k \cdot n \cdot d$. Note that this only counts operations performed on non-zero values and does not depend on block size b.

3.1 OVERVIEW

The PopSparse library supports unstructured (block size $b = 1$) or structured sparsity ($b = 4, 8, 16$). Only blocks up to size $b = 16$ are explicitly considered as this already achieves the theoretical maximum FLOP rate for the on-tile compute operations and we expect larger block sizes to negatively impact task performance (Dietrich et al., 2021). Furthermore, larger blocks could be made up of many 16x16 blocks if desired. Two sparsity modes, static and dynamic, are available. Static sparsity has the sparsity pattern M fixed at compile time while with dynamic sparsity only the maximum density $d^{\max}$ is fixed. Consequently, additional planning is required for dynamic sparsity during compilation and there is an overhead for partitioning based on a specific sparsity pattern during runtime in order to balance the quantity of sparse elements across tiles.



(a) Static

(b) Dynamic

Figure 1: Illustration of static and dynamic sparse partitioning of the input feature dimension, k. Note that $m = k = 9$ and we split work over 3 tiles with $q^k = 3$, $q^m = q^n = 1$. Static partitioning can adapt the split positions to the known sparsity pattern, removing the need for dynamic sparsity's allocation overflow (b.1) and redistribution phase (b.2). Static execution also allows optimal exchange of inputs, (a.1) vs (b.1) and in output reduction, (a.2) vs (b.3).

3.2 STATIC SPARSITY

With static sparsity, during compile time, the sparsity pattern and matrix shapes (m, k and n) are known to the partitioner, which splits the non-zero elements (specific values not known yet) of the sparse matrix across k dimension into q^k partitions and the dense matrix across n dimension into q^n partitions. Where the total number of partitions, $q^k \cdot q^n$, is upper bounded by the number of tiles. Figure 1a illustrates the distribution, computation and reduction of the sparse matrix in static sparse. Splits over the k dimension do not have to be evenly sized, and are chosen to ensure a balanced distribution of the non-zero elements (denoted with coloured small blocks in the figure). In the diagram, three distributed partitions of non-zero elements are mapped onto three tiles. A Poplar (Graphcore, 2022e) graph (compute graph) is then constructed based on the partition information.

At runtime, the specific non-zero values of W are provided by the host and re-ordered to match the distribution of non-zero elements on tiles of the IPU. As this distribution is known in advance, no extra exchange of data is needed. Local dot product computation with the dense matrix is then executed and there is a final reduction across tiles to give the output dense matrix. A flow chart for static sparsity can be found in Figure 5a of the Appendix A.1.

3.3 DYNAMIC SPARSITY

With dynamic sparsity, the maximum number of non-zero elements for the sparse operand is fixed, while the sparsity pattern is updated during runtime. To minimise compute cycles, we employ a planner during compile time to optimise the partitioning while accommodating for the full range of sparsity patterns that can be defined by the host at runtime. Specifically, the planner determines how many *equal* parts each dimension (m, k, n) should be divided into with each tile being assigned to a single partition.

At runtime, as the sparsity pattern could change for each run, compared to static sparsity, there is an additional distribution of sparse matrix indices and non-zero values by the partitioner. This is demonstrated in Figure 1b. Furthermore, as fixed sized partition over m and k dimension is required, the number of non-zero elements in each partition may not be balanced. In Figure 1b the B elements on the second partition of k dimension do not fit on $\mathrm{Tile1}$. Three of the B elements overflow to $\mathrm{Tile2}$. However, the $\mathrm{Tile2}$ does not have all the partition information (slices of m, k, n) corresponding with B elements to compute the result. Consequentially, extra steps of propagation are induced to finish the computation. A dynamic number of steps is required (depending on the sparsity pattern) to propagate information and compute the final result.

Figure 5b in Appendix A.1 shows a diagram of the dynamic sparse-dense matmul operation during compile time and runtime. More details about dynamic sparsity can be found in Appendix A.2.

Overall our dynamic sparse implementation has a number of additional costs compared to the static case:

Table 1: APIs used for benchmarking.

Device	Implementation	API	Block size	Dynamic	Datatype support
IPU	Dense	`poplin::matMul`	–	–	FP16, FP32
IPU	Static sparse (`popsparse::`)	`static_::sparseDenseMatMul`	1,4,8,16	✗	FP16, FP32
IPU	Dynamic sparse (`popsparse::`)	`dynamic::sparseDenseMatMul`	1,4,8,16	✓	FP16, FP32
GPU	Dense	`cublasGemmEx`	–	–	FP16, FP32
GPU	Compressed Sparse Row (CSR)	`cusparseSpMM`	1	✓	FP16*, FP32
GPU	Block Sparse Row (BSR)	`cusparseSbsrmm`	$2^{b'}$	✓	FP32

$*$ compute in FP32, inputs/outputs in FP16

- On-tile compute requires additional control flow which incurs some cost overhead.
- Exchange phases must account for the largest communication volume possible and thus are less efficient.
- When data is unbalanced, additional propagation and compute phases are required to rebalance the data. The number of steps required is dependent on the degree of the imbalance.

4 BENCHMARKS

Our microbenchmarking experiments explore a single sparse-dense matmul SpMM: $Y = (M \odot W) * X$, on IPU and GPU. The APIs used for IPU experiments and GPU baselines are shown in Table 1. The range of parameters that were benchmarked is detailed in Table 2.

Table 2: Ranges of parameters swept for benchmarks.

Parameter	Range (start:step:end or list of values)
feature size ($m = k$)	$2^{8:1:13}$
batch size (n)	$2^{2:2:16}$
block size (b)	1 (unstructured), $4, 8, 16$
density factor (d)	1 (dense), $1/4, 1/8, 1/16, 1/32$
data type	FP16, FP16*, FP32

$*$ compute in FP32, inputs/outputs in FP16

IPU The operation is executed a single time on a single Bow IPU (Graphcore, 2022b) in a Bow-2000 chassis, with randomly generated sparsity pattern and values. We extract cycle count information and convert these cycle counts into TFLOP/s values given a constant clock of 1.85 GHz (Graphcore, 2022c). Host transfers are excluded from measured time.

GPU baselines Sparse implementations on GPU are provided by the cuSPARSE library[1] (NVIDIA, 2022b). These are divided according to the sparse format used, with no distinction between static and dynamic sparsity patterns. We consider compressed sparse row (CSR) and block sparse row (BSR), for unstructured and block sparsity respectively.[2] There are other vendor-provided and third-party libraries for sparse computation, but we consider these for sake of direct comparison against PopSparse (see Appendix B for more information). Dense benchmarks used cuBLAS (NVIDIA, 2022a).

To measure runtime, we generate a random sparsity pattern and values, copy to device, execute 25 iterations of the operation and copy results to host for validation. We start timing after 5 iterations, and measure wall clock time using `cudaEventRecord`, at millisecond resolution, over all remaining iterations. We use `cudaDeviceSynchronize` before the start and after the stop of wall clock recording to ensure previous matmul operation is complete. All experiments were performed on an A100 GPU running in a 4 x A100-SXM4-40G chassis with CUDA version 11.3.

[1] cuSPARSE version 11.6.2

[2] We skip the detailed implementation here for brevity, referring to Yang et al. (2018) and Huang et al. (2020) for a discussion of such considerations (although these do not discuss the implementation of cuSPARSE directly).

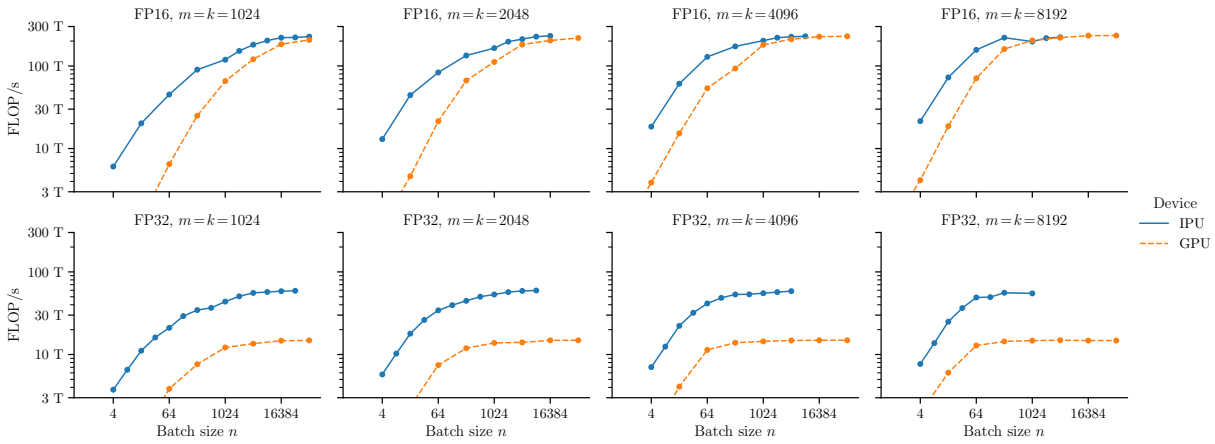


Figure 2: Dense performance for large square matrices on IPU and GPU.

5 Results

In our benchmarking experiments, we explore the performance of static and dynamic sparsity on IPU, compared with dense computation on IPU and both dense and sparse computation on GPU.

For most results, we allow batch size n to vary, choosing the batch size leading to best performance for each configuration. This is appropriate for batched inference in a weight sparse neural network, as batch size is a free parameter (up to a memory limit) and the choice depends on the machine learning application. Note that we consider only non-zero elements in the sparse matrix to calculate FLOP/s, as defined in Section 3.

5.1 Dense baseline performance

Figure 2 shows dense matmul performance, as batch size and feature size vary for FP16 and FP32. In FP16, we see chip-for-chip parity between IPU and GPU at large batch size, while the IPU is more resilient to low batch size. This is because lower arithmetic intensity operations leads to more data movement with the same amount of compute. The SRAM on IPU enables more efficient data movement.

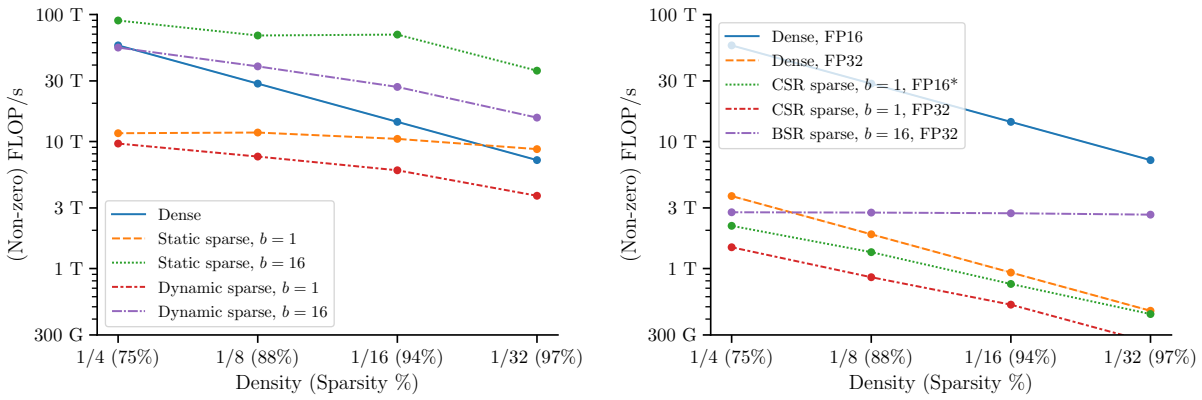
In FP32, the IPU has a clear advantage due to Accumulating Matrix Product (AMP) units (Graphcore, 2022a) being available in FP32, whereas Tensor Cores (NVIDIA, 2022c) are only available in FP16.

5.2 Static vs. dynamic sparsity on IPU

It has already been discussed in Section 3.3 that static sparse performance should exceed dynamic. This is visible in Table 3, showing the speedup of dynamic and static sparsity versus dense for a variety of configurations. Over various block sizes and data types, static sparsity shows higher speedup than dynamic. The sparse computation speedup also increases with block size. More discussion can be found in Section 5.3. Moreover, we can see that the sparsity speedup for FP32 is better than for FP16. This is because core arithmetic is more expensive in FP32, so FLOP savings count more (versus sparse overhead).

Table 3: Dynamic sparse vs. static sparse on IPU, $m = k = 4096$, density $d = 1/16$, best over batch size n, throughput values compared with dense.

Block size	Type	Dynamic/dense	Static/dense
1	FP16	0.4	0.7
1	FP32	0.9	1.4
4	FP16	1.0	1.5
4	FP32	2.7	3.2
16	FP16	1.9	4.9
16	FP32	3.8	5.6



(a) IPU

(b) GPU

Figure 3: (a) IPU FP16 and (b) GPU block-sparse matmul performance as density varies for various block size b, square feature size $m = k = 4096$, best over batch size n.

5.3 When does a sparse implementation outperform dense on IPU?

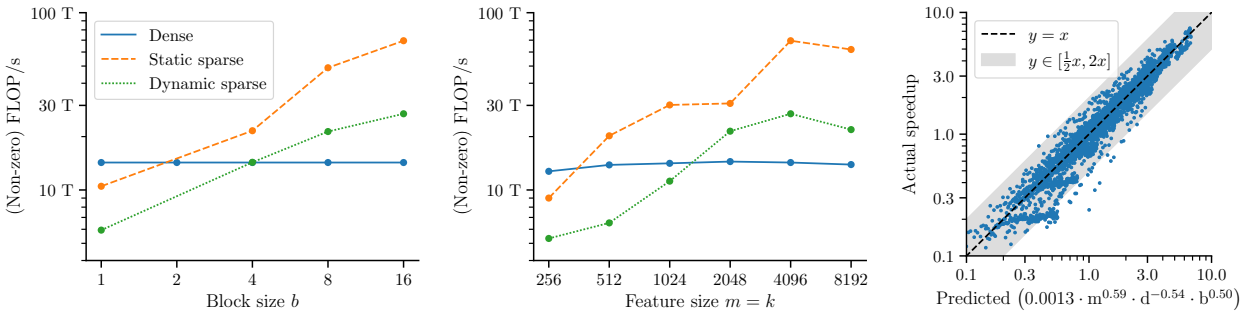
Our results show that sparse implementations are faster than dense on IPU at *low density*, with *large block size* and *large feature size*.

Low density Since the purpose of sparse implementations is to take advantage of zeros, it is unsurprising that decreasing non-zero density leads to a performance advantage over dense. In Figure 3a, we show the plot of compute FLOP/s (in log scale) vs. density with fixed feature size. This shows the density scaling performance of dense, dynamic sparse and static sparse for block size $b = 1, 16$. The dense implementation shows linear scaling with density — since zeros are not counted in the FLOP/s calculation, the amount of useful work decreases with d, while the runtime is unchanged. While on the other hand for a sparse implementation, perfect scaling would predict constant FLOP/s as density varies, with both work and runtime scaling with d. We see nearly-perfect scaling from static sparsity in this range, whereas dynamic sparsity shows higher overheads. In Figure 3a, unstructured ($b = 1$) static sparsity outperforms dense at a density of 5%.

Large block size Block sparsity speeds up sparse computation for two main reasons. First, there is less overhead incurred to store and process the metadata that encode the sparsity pattern. Second, it allows implementations to process non-zero submatrices rather than just vectors, making more effective use of the hardware. Figure 4a shows the magnitude of these improvements, which are considerable: $2.1\times$ for blocks size $b = 4$ and $6.6\times$ for $b = 16$ (static sparsity). Despite this advantage, few sparsity techniques in the literature consider block sparsity, so these results could motivate the field to explore new ML algorithms for inducing blocks of zeros.

Large feature size Figure 4b shows the effect of feature size on sparse performance. Sparse performance improves more with feature size than dense performance. This is because increasing feature size makes it easier to spread work evenly across tiles.

Will my application speed up? The multidimensional nature of our sweep makes it hard to summarise results on a single plot. To make it easier to evaluate whether a particular problem can be accelerated using PopSparse, we provide a grid of static sparse speedup ratios in Figure 7. We also fit a power-law model to our static sparse results, illustrated in Figure 4c. This predicts the condition for achieving a speedup: $0.0013 \cdot m^{0.59} \cdot d^{-0.54} \cdot b^{0.50} > 1$. This is in line with our main observations: large feature size, low density and large block size all favour sparse implementations. We recommend using this for interpolation only, as we do not expect good generalisation beyond the sweep parameters of Table 2.



(a) (b) (c)

Figure 4: IPU block-sparse matmul performance as (a) block size or (b) feature size varies. Base configuration: FP16, $m = k = 4096$, density $1/16$, block size 16, best over n. Figure (c) shows the fit of a power-law model for the sparse speedup ratio (static sparse to dense TFLOP/s), for FP16, best over n, for each (m, d, b) tested.

5.4 How well does IPU perform compared to GPU in sparse operations?

Figure 3b shows GPU performance as density varies, showing that BSR sparsity in FP32 is worse than the FP16 dense baseline, even below 2% density. However, we also observe that GPU sparse performance scales well as density decreases. We suspect the main reason for the sparse-dense performance gap is that the BSR implementation does not support FP16, and therefore cannot use Tensor Cores to accelerate the arithmetic (see Figure 2 for the FP32 versus FP16 dense performance). Note that FP16 sparsity is available in the blocked ELL format, which we did not benchmark as it did not match the API of general block sparsity and so would require explicit padding. However, it is probable that the gains due to Tensor Core acceleration would negate any padding overheads in this case, so we consider this for future work.

Since dense methods perform best on GPU, and GPU dense performs similarly to IPU dense (see Figure 2), our previous comparisons already give an approximate answer to the question of when sparse operations on IPU outperform GPU: whenever sparse operations outperform dense operations on IPU. Therefore Figure 7 in Appendix C can also be used as an approximate guide to compare IPU sparsity with GPU.

6 Conclusions

In this work, we introduce the PopSparse library for sparse matrix multiplication on IPU and show the benchmarks of SpMM on IPU with static sparsity and dynamic sparsity. We also compare the sparse and dense results to a GPU baseline.

We have found that there are configurations where sparse implementations on IPU can outperform dense on IPU and all tested implementations on GPU (cuSPARSE CSR and BSR APIs).

As an approximate guide, sparsity brings speedup over dense (FP16, large batch size for both) when:

- Static, block size $b = 1$: features $(m = k) \geq 4096$, density $d \leq 1/32$.
- Static, block size $b \geq 4$: features $(m = k) \geq 4096$, density $d \leq 1/8$.
- Dynamic sparsity: block size $b \geq 8$, features $(m = k) \geq 4096$, density $d \leq 1/32$.

The requirements for both static and dynamic sparsity to outperform dense on IPU are much more modest in FP32, however FP16 is generally preferred for weight-sparse deep learning models.

The ranges which show good performance are somewhat challenging to use today in the context of weight sparse training and inference. Only a few deep learning literature (for example the Pixelated Butterfly from Dao et al. (2021) and Towards Structured Dynamic Sparse Pre-Training of BERT from Dietrich et al. (2021)) explore effective schemes for block sparsity. This might because finding block sparse patterns that do not degrade performance is a hard problem, and that on theoretical efficiency metrics (per non-zero FLOP), unstructured sparsity will always appear as good or better than block sparsity. We hope that this work shows that block sparsity is a promising method

for achieving practical acceleration of sparse models and will motivate further investigation into effective block sparse pruning algorithms.

For the next step, we aim to demonstrate the static block sparsity to accelerate large model inference on IPU, achieving both high memory and high compute efficiency.

REFERENCES

Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Structured pruning of deep convolutional neural networks, 2015. URL https://arxiv.org/abs/1512.08571.

Tri Dao, Beidi Chen, Kaizhao Liang, Jiaming Yang, Zhao Song, Atri Rudra, and Christopher Ré. Pixelated butterfly: Simple and efficient sparse training for neural network models, 2021. URL https://arxiv.org/abs/2112.00029.

Anastasia Dietrich, Frithjof Gressmann, Douglas Orr, Ivan Chelombiev, Daniel Justus, and Carlo Luschi. Towards structured dynamic sparse pre-training of BERT, 2021. URL https://arxiv.org/abs/2108.06277.

Abdolghani Ebrahimi and Diego Klabjan. Neuron-based pruning of deep neural networks with better generalization using kronecker factored curvature approximation, 2021. URL https://arxiv.org/abs/2111.08577.

Utku Evci, Trevor Gale, Jacob Menick, Pablo Samuel Castro, and Erich Elsen. Rigging the lottery: Making all tickets winners, 2019. URL https://arxiv.org/abs/1911.11134.

Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. Sparse GPU kernels for deep learning, 2020. URL https://arxiv.org/abs/2006.10901.

Trevor Gale, Deepak Narayanan, Cliff Young, and Matei Zaharia. Megablocks: Efficient sparse training with mixture-of-experts, 2022. URL https://arxiv.org/abs/2211.15841.

Graphcore. Mixed-precision arithmetic for AI: A hardware perspective, 2022a. URL https://docs.graphcore.ai/projects/ai-float-white-paper/en/latest/ai-float.html#lower-precision-for-higher-power-efficiency.

Graphcore. Bow IPU processor, 2022b. URL https://www.graphcore.ai/bow-processors.

Graphcore. Bow-2000 IPU-machine technical specifications, 2022c. URL https://docs.graphcore.ai/projects/bow-2000-datasheet/en/latest/product-description.html#technical-specifications.

Graphcore. IPU programmer's guide, 2022d. URL https://docs.graphcore.ai/projects/ipu-programmers-guide/en/latest/programming_model.html#loading-and-running-programsy.

Graphcore. Poplar and poplibs user guide, 2022e. URL https://docs.graphcore.ai/projects/poplar-user-guide/en/latest/index.html.

Scott Gray, Alec Radford, and Diederik P. Kingma. GPU kernels for block-sparse weights. 2017.

Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks, 2017. URL https://arxiv.org/abs/1707.06168.

Hatem Helal, Jesun Firoz, Jenna Bilbrey, Mario Michael Krell, Tom Murray, Ang Li, Sotiris Xantheas, and Sutanay Choudhury. Extreme acceleration of graph neural network-based prediction models for quantum chemistry, 2022. URL https://arxiv.org/abs/2211.13853.

Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks, 2021. URL https://arxiv.org/abs/2102.00554.

Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P. Sadayappan. Adaptive sparse tiling for sparse matrix multiplication. *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, 2019.

Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. Ge-spmm: General-purpose sparse matrix-matrix multiplication on GPUs for graph neural networks. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020.

Shaohui Lin, Rongrong Ji, Yuchao Li, Cheng Deng, and Xuelong Li. Towards compact ConvNets via structure-sparsity regularized filter pruning, 2019. URL https://arxiv.org/abs/1901.07827.

Asit Mishra, Jorge Albericio Latorre, Jeff Pool, Darko Stosic, Dusan Stosic, Ganesh Venkatesh, Chong Yu, and Paulius Micikevicius. Accelerating sparse deep neural networks, 2021. URL https://arxiv.org/abs/2104.08378.

Hesham Mostafa and Xin Wang. Parameter efficient training of deep convolutional neural networks by dynamic sparse reparameterization, 2019. URL https://arxiv.org/abs/1902.05967.

Sharan Narang, Erich Elsen, Gregory Diamos, and Shubho Sengupta. Exploring sparsity in recurrent neural networks, 2017. URL https://arxiv.org/abs/1704.05119.

NVIDIA. API reference guide for cuBLAS, 2022a. URL https://docs.nvidia.com/cuda/cublas/.

NVIDIA. API reference guide for cuSPARSE, 2022b. URL https://docs.nvidia.com/cuda/cusparse/.

NVIDIA. NVIDIA tensor cores, 2022c. URL https://www.nvidia.com/en-gb/data-center/tensor-cores/.

Eric Qin, Raveesh Garg, Abhimanyu Bambhaniya, Michael Pellauer, Angshuman Parashar, Sivasankaran Rajamanickam, Cong Hao, and Tushar Krishna. Enabling flexibility for sparse tensor acceleration via heterogeneity, 2022. URL https://arxiv.org/abs/2201.08916.

Ziheng Wang. Sparsert: Accelerating unstructured sparsity on GPUs for deep learning inference, 2020. URL https://arxiv.org/abs/2008.11849.

Carl Yang, Aydın Buluç, and John D Owens. Design principles for sparse matrix multiplication on the GPU. In *Euro-Par 2018: Parallel Processing: 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings*, 2018.

Zhuliang Yao, Shijie Cao, Wencong Xiao, Chen Zhang, and Lanshun Nie. Balanced sparsity for efficient DNN inference on GPU. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):5676–5683, jul 2019. doi: 10.1609/aaai.v33i01.33015676. URL https://doi.org/10.1609%2Faaai.v33i01.33015676.

Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression, 2017. URL https://arxiv.org/abs/1710.01878.

A More details about static and dynamic sparsity

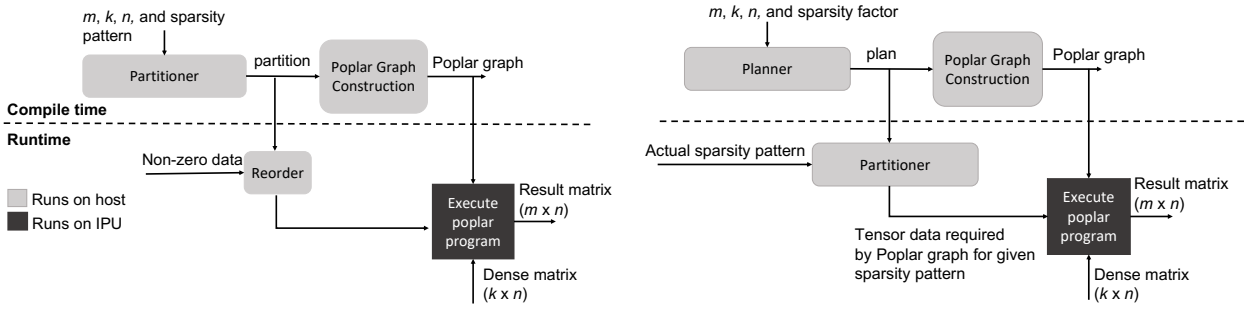
A.1 Flow chart for static and dynamic sparsity

A.2 Details about dynamic sparsity

With dynamic sparsity, the total number of non-zero elements for the sparse operand is fixed (or has a known maximum) while the sparsity pattern is updated during runtime.

There are three key components for the implementation of the dynamic sparsity in PopSparse:

- A **planner** which produces the plan for optimum partition and device operation.
- A **host utility** which encodes a sparsity pattern to be uploaded to the device and used by the device-side implementation.
- The **implementation on the device** that is based on the plan produced by the planner. This constructs the variables, compute sets and programs needed to run the ops on the device.



(a) Static (b) Dynamic

Figure 5: Flow chart for static and dynamic sparsity.

The planner consists primarily of a partition between tiles and vertices of the computation graph used on each tile. The partitioner determines how each dimension (m, k, n) should be divided into when calculating the result of the sparse-dense matmul. It is decided at compile time based on the matrix sizes and density factor and does not change with sparsity pattern. All partitions of each dimension are of equal size except the last which may be smaller if the number of partitions in that dimension does not evenly divide. For example, if q^m is the number of partitions to divide the dimension m into, there will be $q^m - 1$ partitions of size $\lfloor m/q^m \rfloor$ and the last partition with size $m - (q^m - 1) \cdot \lfloor m/q^m \rfloor$. The total number of partitions, $q^{\text{total}} = q^m \cdot q^k \cdot q^n$.

Each tile will be assigned a single partition, for which it is responsible for computing its portion of the partial result. Therefore the number of tiles used is equal to q^{total}.

The host utility encodes the sparse matrix row and column indices as metaInfo. The metaInfo along with the non-zero values of the sparse matrix (nzValues) are splits into buckets. The size of these buckets is set by the planner based on the maximum density that was fixed at compile time. The buckets are mapped to each of the q^{total} number of partitions so that all partitions that are assigned to tiles have equal and fixed size buckets with metaInfo and nzValues. The bucket size is based on distributing the total number of non-zero values for the sparse tensor across $q^m \cdot q^k$ partitions evenly:

$$N_{\text{non-zero}} = m \cdot k \cdot d^{\max} / (q^m \cdot q^k) \,,$$

where $N_{\text{non-zero}}$ is the average number of non-zero elements in a single bucket on a single tile, based on maximum density $d^{\max}$. The buckets are repeated over q^n partitions. A similar estimate is done for the size of metaInfo for a single bucket but since we have a varied sparsity pattern, some extra headroom is given in the size of these buckets.

With the device implementation, we have compute graph vertices created on each tile with:

- The slice of the dense input required to compute this partition's result.
- The bucket of the input sparse tensor mapped to this tile (both metaInfo and nzValues).
- The slice of the dense output partials that will hold the results for this partition.

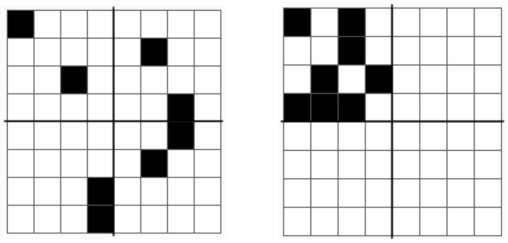
The execution steps can be summarized as following:

- Distribution phase: The tile reads through the metaInfo and nzValues in the bucket on a tile that fall within the partition to be processed on this tile. If all the non-zero values for all $q^m \cdot q^k$ partitions are distributed in buckets on tiles assigned the same partition, the calculation is complete. If not, a further series of dynamic exchange + compute steps (propagation) are executed to complete the calculation.
- Propagation phase, use exchanges to move buckets between tiles. This phase repeats until compute is complete.
  - A, Check information encoded in metaInfo to see if the calculation is complete, exit if complete.
  - B, If not, exchange buckets to shift between tiles.
  - C, Compute, dense input is the same but the contents of the buckets on tiles have changed.
- Reduce partial results across q^k to get the final output.

The distribution and propagation phases are done with the dynamically executed steps where the number of steps are decided by the partitioner for a given sparsity pattern.

To further illustrate the distribution and propagation phases, we use two extreme cases. The best case scenario is when non-zero values in the sparse operand are evenly spread among $q^m \cdot q^k$ partitions. In this balanced case, the buckets mapped to tiles assigned each $q^m \cdot q^k$ partition should have space enough to hold all non-zero values falling within that partition. Then we can complete the operation in the distribution phase and no further propagation phases are needed. At the same time, each tile does an equal share of the total FLOPs. The worst case scenario is when all non-zero values in the sparse operand lie within a single $q^m \cdot q^k$ partition. The first tile in which all the non-zero values lie must do all the FLOPs for the operation, and all the other tiles do no useful work. Moreover, the bucket size could only take $1/(q^m \cdot q^k)$ of the non-zero values. As a result, all the non-zero elements must be spread between all the buckets. This means we must exchange + compute up to $q^m \cdot q^k$ times to complete the operation.

To give an example, in Figure 6a, the best scenario, there are two non-zero values for each partition (in total four partitions). Each bucket contains two $\mathrm{nzValues}$ and no further exchange is needed. With the worst scenario, as denoted in Figure 6b, all the non-zero values lie within the first tile and all the rest three tiles do no useful work.



(a) Balanced distribution

(b) Unbalanced distribution

Figure 6: Examples of the best (left) and worst (right) case scenarios for distribution and propagation phases. Cells marked black denote non-zero values and white denote zeros.

Somewhere in-between the two extremes described, there is a choice about how to spill non-zero elements from buckets which become too full (due to non-uniform distribution of sparsity pattern) to other buckets. When choosing a bucket to spill elements to, we can define a concept of distance based on the nested iteration (from innermost to outermost around the partitions of n, k and m) and attempt to minimise this distance for all elements.

B CUDA API THAT WERE NOT CONSIDERED IN THE CURRENT WORK

We note that the CUDA libraries offer support for additional, more restrictive sparsity patterns. We did not benchmark these, as there are no equivalent implementations on the IPU, and it is hard to quantify the modelling cost of a stricter sparsity constraint. Therefore, we only seek to compare identical computations on IPU and GPU. The CUDA APIs that we discarded due to this constraint were:

cusparseSpmm with cuSPARSE blocked ELL format. This format is a block-sparse matrix format with a maximum non-zero blocks per row. It stores an array of block column indices with a marker (-1) for missing blocks. Note that this is one of the few CUDA SpMM formats that supports FP16 compute. It would be possible to use this API to implement a static block-sparse pattern by padding rows to the maximum number of non-zero blocks, but this would presumably have a performance impact.

cuSPARSELt™ with N:M structured sparsity. This enforces a regular sparsity rule, for example 2:4, which specifies that exactly 2 of every chunk of 4 contiguous weights are non-zero. This is a strong regularity constraint on sparsity patterns and is hard to compare to block sparsity. This method has explicit hardware support in NVIDIA A100 GPUs.

C OVERVIEW OF THE STATIC SPARSE AND DENSE RESULTS ON IPU

Figure 7 shows a more complete view of the speedup ratio of static sparse to dense, as density d, block size b, feature size $m = k$ and batch size n all vary. Dark grey blocks indicate missing data (could not fit on single IPU memory). Same conclusions can be drawn as those stated in Section 5.3. As GPU dense performs similarly to IPU dense (see Figure 2), this figure can also be used as an approximate guide to compare IPU sparsity with GPU.

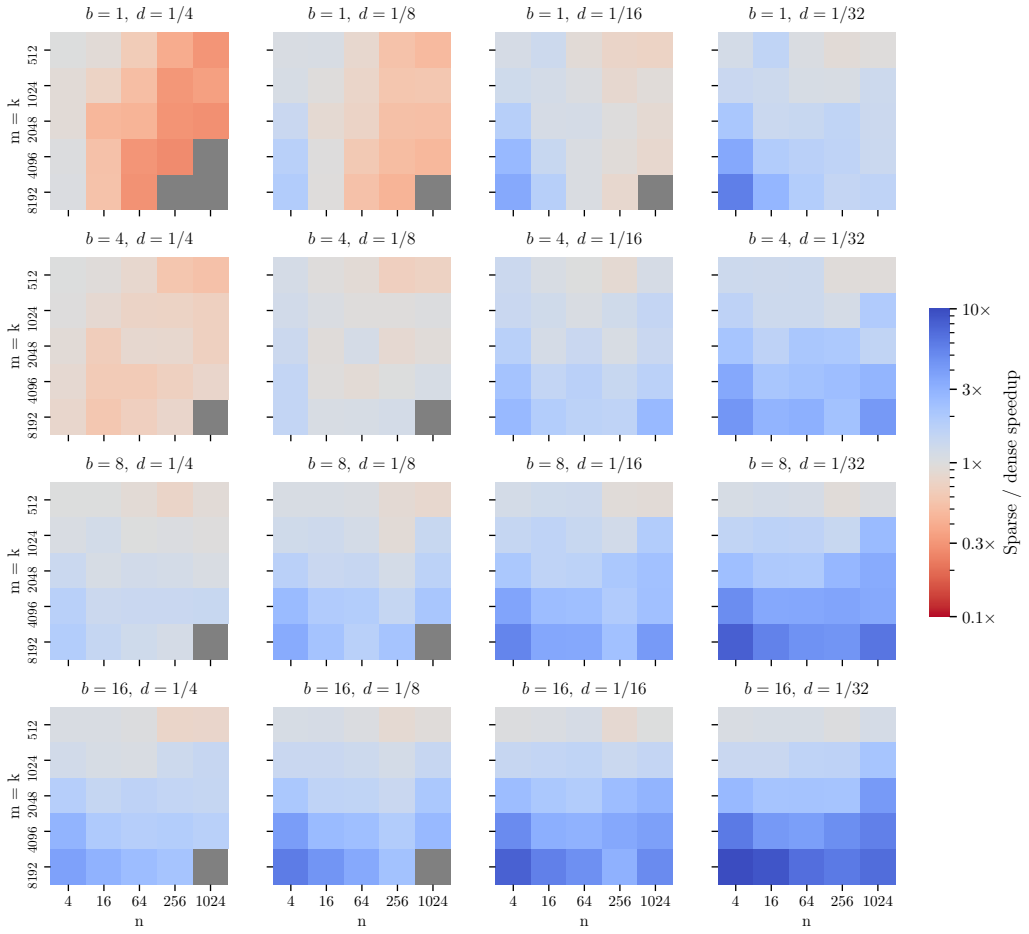


Figure 7: Grid showing the speedup ratio of static sparse to dense.