Extreme Acceleration of Prediction Models for Quantum Chemistry over Molecular Graph Databases

Anonymous Authors[1]

Abstract

Molecular property calculations are the bedrock of chemical physics. High-fidelity *ab initio* modeling techniques for computing the molecular properties can be prohibitively expensive, and motivate the development of machine-learning models that make the same predictions more efficiently. Training graph neural networks over large molecular databases introduces unique computational challenges such as the need to process millions of small graphs with variable size and support communication patterns that are distinct from learning over large graphs such as social networks. This paper demonstrates a novel hardware-software co-design approach to scale up the training of graph neural networks for molecular property prediction. We introduce an algorithm to coalesce the batches of molecular graphs into fixed size packs to eliminate redundant computation and memory associated with alternative padding techniques and improve throughput via minimizing communication. We demonstrate the effectiveness of our co-design approach by providing an implementation of a well-established molecular property prediction model on the Graphcore Intelligence Processing Units (IPU). We evaluate the training performance on multiple molecular graph databases with varying degrees of graph counts, sizes and sparsity. We demonstrate that such a co-design approach can reduce the training time of such molecular property prediction models from days to less than two hours, opening new possibilities for AI-driven scientific discovery.

1 Introduction

Recent advancements in the field of Artificial Intelligence (AI) have fueled notable progress in data-driven scientific discovery. Neural Networks (NNs) provide particular advantage as surrogate models for modeling quantum chemical properties (Schütt et al., 2017). High-accuracy *ab initio* methods, which contain no empirical fitting parameters, are prohibitively expensive, with computational costs that scale as high as $O(N^7)$, where N is the number of atoms (Kulichenko et al., 2021). NNs are able to attain the same level of accuracy as the *ab initio* technique on which the NN was trained but with roughly $O(N)$ scaling (Behler, 2017; Smith et al., 2017; Lubbers et al., 2018). Molecular structures are naturally represented as graphs, which makes Graph Neural Networks (GNNs) (Kipf & Welling, 2017; Veličković et al., 2018; Hamilton et al., 2017) compelling for building surrogate models for quantum chemistry. Initially formulated as a message passing neural network (Gilmer et al., 2017), adding support for both the bonding graph structure and atom-level attributes such as types and 3D coordinates became a key feature of such models (Schütt et al., 2018). The past few years witnessed further expansion of this class of models, henceforth referred to as *molecular GNNs* through sophisticated message passing (Klicpera et al., 2020), accounting for multi-scale structure in the network (Zhang et al., 2020), adoption of self-supervised learning (Wang et al., 2022; Schwaller et al., 2019) and modeling interactions between geometric tensors (Batzner et al., 2022).

Fast training of these models becomes an increasingly important issue as the scientific community embraces these models to train on massive molecular databases (Schmidt et al., 2019; Lim et al., 2019; Bilbrey et al., 2020). Evidence from the literature shows that these models are expensive to train, often spanning multiple days in multi-GPU or TPU settings (Bilbrey et al., 2020; Wang et al., 2022; Addanki et al., 2021). However, most of the research on scaling the training of GNN models have focused on learning over a single massive graph (Tripathy et al., 2020; Wang et al., 2021b; Kaler et al., 2022), which subsequently steers the design space of software and hardware solutions (Zheng et al., 2020b; Gandhi & Iyer, 2021; Zheng et al., 2022) to addressing traditional graph processing challenges such as graph partitioning (Karypis & Kumar, 1998), irregular memory access (Morari et al., 2014) and workload imbalance due to power-law distributions (Geng et al., 2020). In contrast to training a GNN model on a single large graph (see Figure 1), molecular GNNs are trained on *an ensemble of small individual graphs* with varying node counts. Molecular graph databases (Sussman et al., 1998; Rakshit et al., 2019; Pinheiro et al., 2020; Chanussot et al., 2021) are characterized by the comparatively small size (both in terms of the total

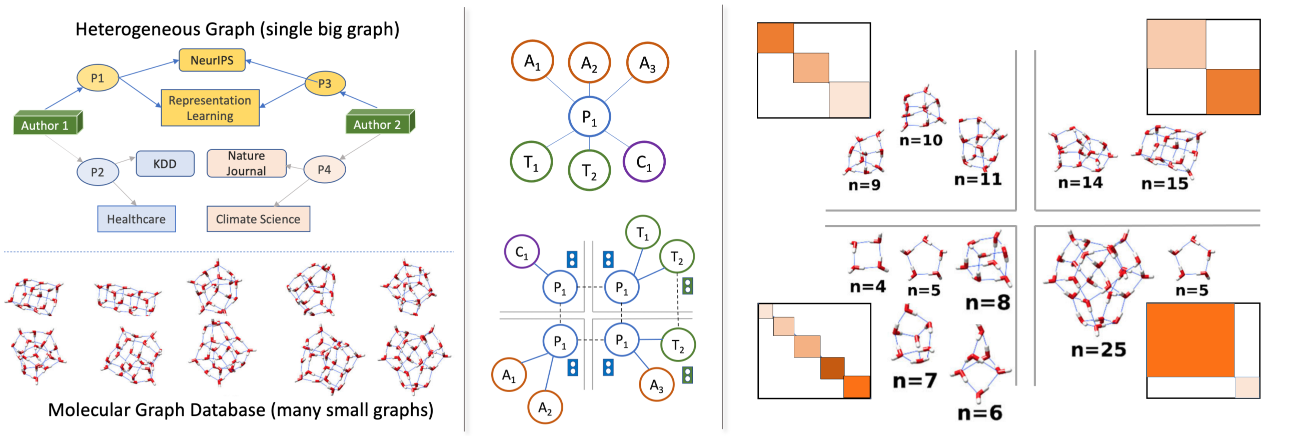


Figure 1. Column 1: Top row shows a knowledge graph or "big graph"; bottom row shows a molecular database. Column 2: Shows how big graphs are sampled (top row) and how nodes that falls across partition trigger communication during learning (bottom row). Column 3: Shows how molecular graphs are packed into nearly 30-node packs. Processing them requires support for efficient sparse block matrix operations. Communications are quite localized.

node count and the edge count) of individual graphs, and wide distribution of size and sparsity. Additionally, most of these datasets represents graphs where each node is also associated with a 3D coordinate representing the position of an atom, which subsequently requires support for geometric operators such as nearest-neighbor computation.

In a first work of its kind, we present a hardware-software co-design approach to accelerate the training of message passing graph neural networks that need to process millions of small graphs with diverse sizes. Recent years have also witnessed the emergence of machine-learning accelerators such as tensor processing units (Jouppi et al., 2017), reconfigurable dataflow (Emani et al., 2021) and intelligence processing units (IPUs) (Jia et al., 2019) etc. The large on-chip high-bandwidth memory, support for fine-grained parallelism, and faster all-to-all communication links make the IPUs attractive for molecular GNN architectures, and is the primary target of this work.

In addition to focusing on a specific architecture, we also identify a model to serve as a representative workload. We focus on the SchNet neural network architecture (Schütt et al., 2018; 2019) for this paper. Given a molecular structure, SchNet constructs a nearest-neighbor graph around each atom within a radial cutoff distance (Figure 2). Next, it learns the representation of each atom by modeling pairwise interactions between atoms based on a number of specified hops. Finally, a graph-level property such as the energy function is predicted by learning a composition function over each atom (or node) embedding. SchNet and its variants have been extensively used for molecular property prediction in diverse domains such as computational chemistry (Bilbrey et al., 2020), drug discovery (Joshi et al., 2021), materials science (Jha et al., 2021). Its pragmatic relevance and its capture of key kernels that are present in nearly all future model variants (Schwaller et al., 2019; Klicpera et al., 2020; Batzner et al., 2022) make SchNet a compelling representative for molecular GNN-driven co-design.

Our paper makes the following contributions.

- We present a comprehensive study of a recently released benchmark dataset (Choudhury et al., 2020a) that is among the largest 3D graph property prediction datasets available with highest degree of size and sparsity imbalance (Section 5.2).
- We propose a hardware-agnostic technique namely *batch packing* to coalesce small graphs while tracking their original connectivity structure for graph-level property prediction (Section 4.1). We show that *batch packing* significantly reduces the memory footprint and improves the performance of the SchNet for diverse molecular databases with varying sizes and sparsity.
- We implement a *scatter-gather planner* for efficient scheduling of the gather and scatter operators of the molecular GNN on the IPUs (Section 4.2.2). The planner finds a sweet spot between parallelism and device utilization. Additionally, we demonstrate the impact of other optimizations such as asynchronous, non-blocking I/O for batch loading, vectorization of scatter operation and pre-fetching for molecular GNN workloads (Section 4.2).
- As a demonstration of extreme acceleration, we train the SchNet model on the Hydronet dataset (Choudhury et al., 2020a) in 92 minutes using 64 IPUs, as compared to previously reported 2.7 days obtained with a single-GPU setting (Bilbrey et al., 2020). Rapid training of scientific ML models is essential to accelerate scientific discovery, and we hope that our work will stimulate further research in this direction.

2 Background: GNN-based Model for Quantum Chemistry

A molecule can be represented mathematically as a *graph*, denoted by $G = (V, E)$, with vertex set V and edge set E (Figure 2). Each vertex $v_i \in V$ represents an atom in the molecule and is associated with that atom's spatial coordinates $\mathbf{r}_i \in \mathbb{R}^3$, and atomic number $z_i \in \mathbb{N}$. The molecular properties calculated by *ab initio* methods such as density functional theory (DFT) are determined entirely these node properties and rely on mean-field approximations to model complex many-body interactions (Kohn & Sham, 1965). Explicit edges are introduced when applying GNNs to molecular property prediction tasks, where each edge $e_{ij} \in E$ reflects a pair-wise interaction between atoms v_i and v_j. The influence one atom has on another decreases non-linearly with the distance $d_{ij} = ||\mathbf{r}_i - \mathbf{r}_j||$ between the pair and beyond some critical distance r_{cut} is known to be negligible. In the context of electronic structure calculations this is often referred to as the *nearsightedness of matter* (Prodan & Kohn, 2005) where the specific value of r_{cut} varies depending on the composition of the molecule or material. We use this cutoff to define edges for the graph representation of a molecule as:

$$e_{ij} \in E \text{ iff } d_{ij} < r_{\text{cut}}. \tag{1}$$

In practice, a K-nearest neighbor (KNN) search is performed to return a fixed number of neighbors for each v. As a consequence, the number of edges in the graph representation will grow at most linearly alongside the number of atoms in the molecule.

The SchNet GNN architecture (Schütt et al., 2018) is trained to the learn mapping from this graph representation as SchNet : $G(V, E) \to \mathbb{R}$ where the prediction of the network is a property of the molecule. The computation graph in this architecture has two phases (Figure 2). The first phase focuses on atom-level representation learning, followed by an aggregation phase that learns a composition function of the set of atoms to predict a global property (such as the potential energy function). The entire computation process is visually described in Figure 2.

Embedding Layer: Each vertex $v_i \in V$ represents an atom, with initial state $\mathbf{h}_i = \text{Embedding}[z_i]$.

Interaction Block/GNN Layer: The *interaction block* layers in Figure 2 are used to learn the atom-level representations via message-passing over the graph. The message to be passed combines states $\mathbf{h}$, and edge attributes $e^{\mathbf{a}}{}_{ij}$, corresponding to a Gaussian radial basis function (RBF) expansion of d_{ij}:

$$e^{\mathbf{a}}{}_{ij} = \left[\exp\left(-\gamma(d_{ij} - k\Delta\mu)^2\right)\right]_{k=0}^{N_{\text{rbf}}} \tag{2}$$

where $\Delta\mu$ is the spacing of Gaussians with scale γ on a grid ranging from 0 to r_{cut}, so $N_{\text{rbf}} = r_{\text{cut}}/\Delta\mu$.

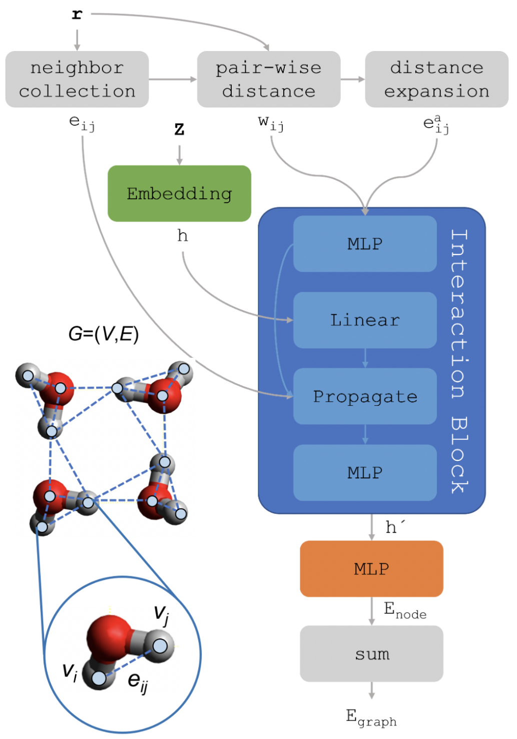


Figure 2. Illustration of the GNN-based implementation of SchNet.

The set of $e^{\mathbf{a}}{}_{ij}$ are sent through a Multi-Layer Perceptron (MLP) to learn the 'continuous filters', which are then weighted by a cosine function of d_{ij} to reflect the non-linear influence an atom has on its neighbor with respect to distance. Meanwhile, each node state $\mathbf{h}$ is passed through a linear layer, after which the states $\mathbf{h}$ and filters are propagated over edges e_{ij} and passed through a subsequent MLP to obtain the updated node state $\mathbf{h}'$:

$$\mathbf{h}'_i = \mathbf{h}_i + \sum_{j \neq i} f(\mathbf{h}_j, e^{\mathbf{a}}{}_{ij}) \tag{3}$$

MLP Layer: The updated state $\mathbf{h}'$ is reduced through an MLP to a scalar value for each node, representing the contribution of each atom to the prediction target.

Activation and Pooling Layers: After the embeddings of all vertices in the final layer are computed, the contribution from all atoms are aggregated to obtain the global property of the molecular graph.

3 IPU Hardware and Software

The recently developed Graphcore IPU combines a number of hardware features with software abstractions to create a platform that is exceptionally well suited to the unique computational demands of molecular GNNs.

Hardware Architecture. The main components of a Graphcore IPU are the processing elements called *tiles*. Each tile consists of one computing core and ≈ 625KB of local on-chip SRAM memory which is sufficient to store a large

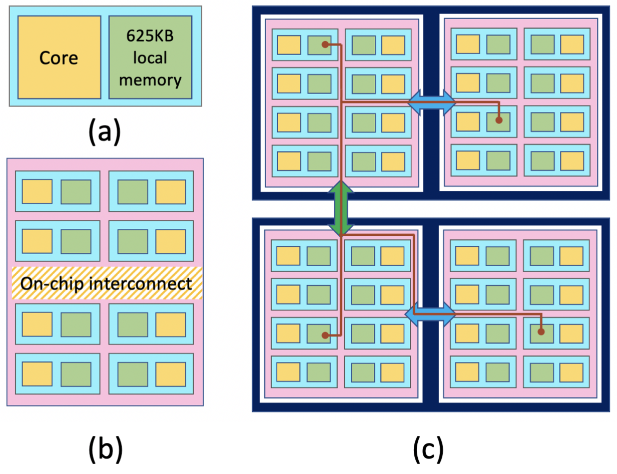


Figure 3. An illustration of tile-based IPU processor architecture and communication topologies used for inter-card and inter-IPU communication. Each card has two IPU processors.

number of (packed) molecular graphs (Figure 3a). The Bow IPU processor consists of a total of 1,472 tiles yielding a total of 900MB of high-bandwidth (65TB/s total) distributed SRAM memory available for machine learning workloads to exploit (Figure 3b). Each computing core has 6 hardware threads that are round-robin multiplexed in time.

An on-chip interconnect implements non-blocking, all-to-all pattern to enable high-bandwidth, low-latency communication among the tiles on an IPU. In addition, for inter-processor communication, low-latency, high-throughput *IPU link*s are used (Figure 3c). PCIe links are used to communicate with the host and in the Bow-2000 configuration, 4 IPUs are connected to a host CPU via PCIe connection (64GB/s) and form a IPU-POD system, and the communication between IPU are bypassing the high-throughput IPU-links (running at 320GB/s).

The IPUs support bulk synchronous parallel (BSP) execution model at the hardware level. This model includes three phases: *compute*: where each tile performs computation with the data available on its local SRAM memory, *sync*: where threads are synchronized across all IPUs, and *exchange*: where each thread exchanges data (located either on the same IPU but on different tile or on a tile of a remotely-located IPU).

Support for ML Operators The IPU is routinely programmed through high-level Python frameworks such as PyTorch and Tensorflow. We use the PopTorch framework (PopTorch, 2022) which provides a set of extensions to PyTorch to efficiently target our implementation of SchNet to the IPU architecture. We implement the interaction layer (section 2) using the PyTorch-Geometric (PyG) framework (Fey & Lenssen, 2019a). An order-invariant gather function is used to aggregate the neighbor's influence on a given node via messages. The updated node embedding information is then *scatter*ed back to the neighbors for the computation of the next set of embeddings in the next interaction layer. These gather-scatter operations for molecular graphs are localized on a single IPU so their execution is accelerated by utilizing the combined benefit of both the high-bandwidth local memory and the high-bandwidth communication over on-chip interconnect.

The PopTorch framework performs compiler optimizations by leveraging both the structure of the model computation graph and the size of input tensors. After capturing an initial intermediate representation (IR), a number of compiler transformation passes are performed over this IR with the aim of mapping high-level PyTorch modules to specific low-level instructions as well as the fusion of multiple constructs that can be performed more efficiently. An example of an operator fusion pass that is essential for the efficient evaluation of the scatter-reduce operation commonly seen in message-passing GNNs is discussed in greater detail in Section 4.2.1.

Our adoption of the PyG framework is motivated by extensibility. In principle, anyone should be able to extend our implementation of SchNet to more sophisticated models by altering the message-passing functions while still benefiting from the molecular GNN-targeted co-design features.

4 Co-Design Optimizations for Molecular GNNs

Effective memory management for the data structures and intermediate computation, optimal parallelism and leveraging efficient all-to-all collectives are the key factors that determine the performance of molecular GNNs. We review a set of optimizations that we have implemented for the SchNet model that can be categorized into three broad categories: input-specific optimization (packing), hardware-targeted optimizations (planning, pre-fetching and compiler passes), and model-specific optimizations (merging weight updates and optimized softplus activation). In the following, we discuss these optimization techniques to improve the overall performance of the molecular GNNs.

4.1 Input-Specific Optimization: Batch Packing

Molecular graph structures vary significantly in terms of the vertex counts and the edge counts. Efficiently targeting the high-bandwidth exchange fabric of the IPU is accomplished through static analysis of communication patterns between tiles for a given application. This ahead-of-time compilation of molecular GNNs requires knowing the shapes of the input tensors as a priori for preparing the batches. To keep the shape sizes like the number of vertices and edges fixed, the naive strategy is to just pad batches up to the maximum number of vertices or edges (Figure 4 (a)). This can lead to a very large proportion of padding and therefore waste com-

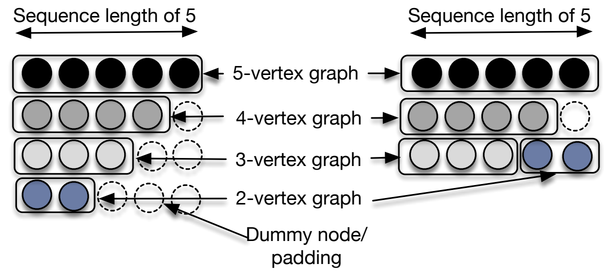


Figure 4. Batch packing

pute. GNN libraries such as PyG support the combination of multiple graphs into a single graph of multiple disconnected components. However, without fixing the sizes of graphs within batches no savings in padding can be made. In this paper, we are the first to apply an approach, previously introduced in natural language processing, where different sequences are concatenated (Krell et al., 2021). However, instead of concatenating sequences/sentences, we combine graphs and instead of the number of words as the sequence length, we measure length by the number of vertices, as proposed in the original paper. We call this strategy *batch packing*.

Packing has been known to be NP-complete problem (Korte & Vygen, 2012) for decades and is well established. It can be formulated as

$$\begin{aligned} \min_{y\in\{0,1\}^n, B\in\{0,1\}^{n\times n}} \quad & \sum_{j=1}^{n} y_j \\ \text{s.t.} \quad & \sum_{j=1} b_{ij} = 1 \quad \forall i \\ & \sum_{i=1}^{n} s(i) b_{ij} \leq s_m y_j \quad \forall j. \end{aligned} \tag{4}$$

We assume there are n graphs to be packed and the combination of graphs is called *pack*. There can be a maximum of n packs where $y_j \in \{0,1\}^n$ indicates which of the potential packs are actually used or not and is subject to minimization. The assignment variable $b_{ij} \in \{0,1\}^{n\times n}$ indicates if the i-th graph gets put into the j-th pack. The number of vertices in a graph is denoted by $s(i)$ and s_m is the maximum number of vertices in a pack. This can be the maximum number of nodes in a graph or a larger number as a trick to reduce padding.

In practice, numerous heuristic-based algorithm for packing exist (Johnson, 1973; Lee & Lee, 1985; Johnson & Garey, 1985; Yue & Zhang, 1995) that could be applied. All these approaches are in the range of $O(n \log n)$ time complexity, where n is the number of graphs/samples. Hence, we use the faster approach from (Krell et al., 2021) that is called *longest-pack-first histogram-packing* (LPFHP). The algorithm is derived from the best-fit method and is shown in Algorithm 1. Best-fit means that an item gets added to the bucket that will leave the least space after the item was

Algorithm 1 Simplified LPFHP

```
# Histogram counts
$h \leftarrow$ empty array
for $g_i \in Graphs$ do
    $h[s(i)] \leftarrow h[s(i)] + 1$
$S \leftarrow$ empty strategy dictionary of lists of pack counts
for $s \in [s_m, ..., 1]$ do
    $c \leftarrow h[s]$
    $R \leftarrow s_m - s$
    while $c > 0$ do
        # Get best fitting pack space and update strategy
        $i = \min_{S[j]\neq\emptyset, j\in\{s,...,s_m\}} j$
        if $i$ is NaN then
            $S[R] \leftarrow [(c, [s])]$
            $c \leftarrow 0$
        else
            $c, S \leftarrow$ update$(S, i, c, s)$
return S
def update(S, i, c, s):
$c_p, p = S[i].pop()$
if $c \geq c_p$ then
    $S[i-s].append([(c_p, p.append(s))])$
    $c \leftarrow c - c_p$
else
    $S[i].append((c_p - c, p))$
    $S[i-s].append([(c, p.append(s))])$
    $c \leftarrow 0$
return c, S
```

added. The algorithm first computes the histogram of graph sizes and respective sample counts. The algorithm also takes as input the potential maximum sequence lengths in a pack, s_m. Next, it iterates from the largest to the smallest graph and tries to add it to existing combined graphs/packs. The trick here is that the algorithm works on histogram bins and counts and not single graphs. If a graph can fit with multiple different already combined graphs, we chose the one with the least space (best-fit). For example, if we need to add a graph with size 10 and our maximum is 100, we would prefer to combine it with a graph of 90 nodes to get a perfect match instead of combining it with a pack or graph of size 11, which is the smallest setting that can occur.

4.2 Hardware-targeted Optimizations

For accelerating the training of molecular GNN models on the IPUs, we leverage fully static compilation and a gather-scatter planner (Section 4.2.2). With static compilation, access to the computation graph provides the opportunity to reduce maximum memory usage by changing the order of operations (called "rescheduling") in a full compute graph to reduce the maximum live memory at any given point in the program. This both may allow a full compute graph to fit into SRAM, and increase the amount of available memory at a given point in the program. Also, increased SRAM availability can allow some operations to make better use of high bandwidth local memory to accelerate operations and assist the planner to decide optimal data layout. At the compiler level, two IPU-specific features are implemented for optimizing the execution of the molecular GNNs: scatter vectorization, and the planning of scatter/gather operators for targeting the IPU.

4.2.1 *Vectorization of gather/scatter operations*

One of the benefits of the ahead-of-time compilation of a PyTorch model is the ability to write specialized passes that can perform operator fusion to efficiently target the IPU hardware. These passes can apply a simple pattern matching on the captured computation graph and replace or fuse multiple operations into new patterns that are known to be more efficient to evaluate. In the context of GNNs, we have developed a fusion pass that finds a sequence where a vector of indices is broadcasted ahead of aggregation. Our pass removes the redundant broadcasting step and ensures that the more efficient vectorized form of the scatter operation is targeted by the scatter/gather planner.

4.2.2 *The Scatter/Gather Planner*

The message passing layer that forms part of the Interaction Block/GNN layer discussed in Section 2 *gather*s the embeddings for atoms in the neighborhood $N(v)$ of each atom, $\mathbf{h}_j$, as well as the generated 'continuous filter' values corresponding to each edge in the graph, e_{ij}, for the aggregation phase of the GNNs.

Gathering yields the values of each row in a matrix A corresponding to each row index in a vector i yielding a matrix gather(A, i).

$$\begin{aligned} A &\in \mathbb{R}^{M\times N} \\ i &\in \mathbb{N}^{I} \\ \text{gather}(A,i) &\in \mathbb{R}^{I\times N} \\ \text{gather}(A,i) &= \begin{bmatrix} A_{i_0,0} & A_{i_0,1} & \cdots & A_{i_0,N} \\ A_{i_1,0} & A_{i_1,1} & \cdots & A_{i_1,N} \\ \vdots & \vdots & \ddots & \vdots \\ A_{i_I,0} & A_{i_I,1} & \cdots & A_{i_I,N} \end{bmatrix} \end{aligned} \tag{5}$$

Once messages have been calculated they are *scatter*ed so that messages for each destination atom v are aggregated together which gives the output embeddings h' for the layer shown in Equation 3.

Scattering takes as input a matrix A and a set U of sparse matrices which are all aggregated. The sparse matrices U are collectively represented as a list of non-zero row indices i and a list of non-zero row values V.

$$\begin{aligned} A &\in \mathbb{R}^{M\times N} \\ i &\in \mathbb{N}^{I} \\ V &\in \mathbb{R}^{I\times N} \\ \text{scatter}(A,U) &\in \mathbb{R}^{M\times N} \\ \text{scatter}(A,U) &= A + \sum_{B\in U} B \end{aligned} \tag{6}$$

Gather provides batch read and *scatter* provides batch read-modify-write of dynamically indexed chunks of memory. These operations' runtimes are dominated by memory access. Optimised implementations of *gather* and *scatter* for IPU hardware are used to coordinate dynamically indexed memory access across the distributed tile memories and maximise memory bandwidth utilization.

Planning. The IPU scatter/gather implementation is controlled by the scatter/gather planner. This is a host utility that minimizes a cost function for a scatter/gather operation by varying implementation parameters. Parameters of the implementation that may be varied control how an individual scatter or gather operation is parallelized across tiles in the IPU.

This implementation of both gather and scatter uses a divide and conquer strategy. The full gather or scatter operations are divided evenly into smaller gather or scatter operations that are locally executed on each tile. The results are then exchanged and optionally reduced to give the final result. The terms P_I, P_M, and P_N represent divisors for 3 dimensions of the full gather or scatter operation I, M, and N outlined in Equations 5 and 6. $I_t = \left\lceil \frac{I}{P_I} \right\rceil$, $M_t = \left\lceil \frac{M}{P_M} \right\rceil$, and $N_t = \left\lceil \frac{N}{P_N} \right\rceil$ give the size of the sub-problem, or partition, calculated on each tile:

$$\begin{aligned} A_t &\in \mathbb{R}^{M_t\times N_t} \\ i_t &\in \mathbb{N}^{I_t} \\ V_t &\in \mathbb{R}^{I_t\times N_t} \\ \text{gather}(A_t,i_t) &\in \mathbb{R}^{I_t\times N_t} \\ \text{scatter}(A_t,i_t,V_t) &\in \mathbb{R}^{M_t\times N_t} \end{aligned} \tag{7}$$

Cost Function for Planner The cost function used for planning (Section 4.2.2) is one that estimates maximum machine cycles required to complete the operation on any one tile and a minimum is found by exhaustive search of valid implementation parameter settings. Simplified equations for gather and scatter planner cost c_{gather} and c_{scatter} are listed in

Equations 8 and 9.

$$
\begin{aligned}
B_{\text{data}} &= \text{bytes per data element} \\
B_{\text{index}} &= \text{bytes per index element} \\
B_{\text{vwidth}} &= \text{tile load/store/acc bytes per cycle} \\
W &= \text{number of worker threads per tile} \\
I_t &= \left\lceil \frac{I}{P_I} \right\rceil \\
M_t &= \left\lceil \frac{M}{P_M} \right\rceil \\
N_t &= \left\lceil \frac{N}{P_N} \right\rceil \\
e(b) &= \frac{b}{\text{tile exchange send/receive bytes per cycle}} \\
g(i,m,n) &= W \left\lceil \frac{i}{W} \right\rceil \frac{nmB_{\text{data}}}{MB_{\text{vwidth}}} \\
c_{\text{partial gather}} &= e(M_t N_t B_{\text{data}}) + e(I_t B_{\text{index}}) + \\
&\quad g(I_t, M_t, N_t) \\
c_{\text{gather reduce}} &= e(I_t N_t B_{\text{data}}) + \frac{I_t N_t B_{\text{data}}}{B_{\text{vwidth}}} \\
c_{\text{gather}} &= c_{\text{partial gather}} + \begin{cases} c_{\text{gather reduce}}, & \text{if } P_M > 1 \\ 0, & \text{otherwise} \end{cases}
\end{aligned}
\tag{8}
$$

$$
\begin{aligned}
s(i,m,n) &= W \left\lceil \frac{m}{W} \right\rceil \frac{inB_{\text{data}}}{MB_{\text{vwidth}}} \\
c_{\text{partial scatter}} &= e(I_t N_t B_{\text{data}}) + \\
&\quad e(I_t B_{\text{index}}) + \\
&\quad u(I_t, M_t, N_t) \\
c_{\text{scatter reduce}} &= e(M_t N_t B_{\text{data}}) + \frac{M_t N_t B_{\text{data}}}{B_{\text{vwidth}}} \\
c_{\text{scatter}} &= c_{\text{partial scatter}} + \begin{cases} c_{\text{scatter reduce}}, & \text{if } P_I > 0 \\ 0, & \text{otherwise} \end{cases}
\end{aligned}
\tag{9}
$$

Here g represents a function estimating machine cycles taken to execute a gather on a single tile in terms of the SRAM load/store bandwidth B_{vwidth}. s represents a function estimating machine cycles taken to execute a scatter on a single tile in terms of the SRAM load/store bandwidth and accumulation vector width B_{vwidth}. Note that these equations omit many overheads in the implementation for simplicity and represent more of a theoretical minimum. The real cost function used by the planner accounts for more of these overheads and practical implementation details. The function e simply represents how many cycles a certain number of bytes would take to be sent or received on a tile.

There are different tradeoffs when varying P_I, P_M, and P_N captured by this cost function. Increasing one of these may increase the amount of data that must be communicated between tiles. Increasing another may require a more costly reduction after computing per-tile partition results. Note that when all data, indices, and results fit into SRAM and we can use this implementation of gather or scatter, data never need be read from/written off-chip. Inter-tile communication under this implementation is always balanced allowing every tile to fully utilise its receive bandwidth thanks to regular communication patterns between tiles, and balanced sent/received payloads on each tile. Under the assumption that row indices are i.i.d. along the indexable dimension M load/store/accumulate hardware is fully utilised during on-tile gather, scatter, or reduction steps.

For both gather and scatter operations, this results in an initial exchange of data between tiles to provide the input to each tile's partition, followed by on-tile computation to calculate the result for each tile's partition of the full problem. A reduction of the results of either gather(A_t, i_t) or scatter(A_t, i_t, V_t) for each tile will then require further reduction to get the final result when $P_M > 1$. This is necessary as the full range of the indexed dimension M is not available on each tile in this case. The additional reduction entails an all-to-all exchange of data between tiles to redistribute partial results (there will be P_M partial results spread over of P_M tiles), followed by on-tile computation to reduce the P_M partials to get the final result.

4.2.3 Host-Device I/O Optimizations

Asynchronous, non-blocking I/O The preparation of mini-batches can be expensive as it involves the random access of irregular sized molecular graphs followed by the collation process to combine multiple graphs into one large disconnected graph. We can improve the time to access a single graph by using a two-level caching strategy:

- molecular graphs are stored on disk in an efficient compressed serialized binary representation for multi-dimensional tensor data.
- the fully materialized graph data structure is cached in memory on first time access which helps reduce redundant disk I/O.

We apply this caching strategy within multiple asynchronous workers for preparing mini-batches on the host, which effectively overlaps the time for preparing mini-batches with the computation on the IPU.

Pre-fetching To reduce the waiting time between host-device transfers, a *pre-fetch depth* can be specified for the data stream transfer. The depth dictates the number of pre-fetched batches and enables the host-to-device transfer of the next mini-batch simultaneously while the device continues evaluation of the training loop on the current mini-batch.

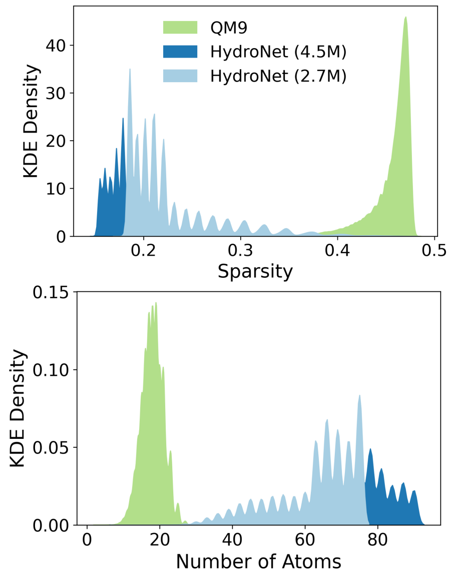


Figure 5. Characteristics of the HydroNet (Choudhury et al., 2020b) and QM9 (Ramakrishnan et al., 2014) benchmark datasets and example molecular graph G for a water cluster with 12 vertices. Here KDE stands for kernel density estimate. Smaller sparsity value implies sparser graph.

4.3 Model-specific Optimizations

Merged Communication Collectives. During data parallel training, each replica (IPU) computes a local gradient for the mini-batch. These gradients are then combined across all replicas ahead of performing the weight update. To reduce the latency in performing the weight update, all reduce collectives are merged together to communicate all variables at once. The effect of this optimization can be observed from the profiler output included in Figure 12 in the Appendix.

Optimized softplus. The default implementation of the softplus activation is defined in PyTorch as:

$$\text{softplus}(x) = \begin{cases} \frac{1}{\beta}\log(1+\exp(\beta x)), & \text{if } \beta x \leq \tau \\ x, & \text{if } \beta x > \tau, \end{cases} \tag{10}$$

where τ is a threshold value for which the activation saturates to the linear function. This conditional expression is used for numerical stability but for the default values of $\tau = 20$ and $\beta = 1$ we can more efficiently evaluate the stable softplus activation equivalently as follows:

$$\text{softplus}(x) = \log(1+\exp(-|x|)) + \max(x, 0), \tag{11}$$

which is used in our optimized implementation of the SchNet model as this simplifies expression compiles down to a more efficient compute vertex than the original formulation in Equation 10 and is numerically stable without additional parameters.

5 Evaluation

We performed a comprehensive set of experiments to evaluate the effect of our proposed optimizations for the SchNet model to predict the molecular properties on a large-scale IPU system. In addition, we conducted strong scaling experiments and benchmarked our holistic co-designed approach with an out-of-the-box SchNet implementation modified to run on multiple GPUs. These results are discussed in the following subsections.

5.1 Experimental Setup

5.1.1 Hardware Configuration

The GPU experiments were conducted on Ampere A100 GPUs. This system consists of 8 A100 GPUs with 40 GB main memory attached to each GPU. All IPU experiments were conducted on a Bow Pod64 IPU system. We used this system to study the scaling behavior for data parallel training from a single IPU up to 64 IPUs.

5.1.2 Hyperparameter setup

We used an up-to-date SchNet model implementation in PyTorch Geometric for all experiments reported here. Except where otherwise noted, we used 4 interaction blocks, a hidden feature size of 100, and a uniform grid of 25 Gaussians for the basis function expansion of the inter-atomic distances. This model was compiled for IPUs using PopTorch version 3.0.0 which uses PyTorch version 1.10.1 implicitly. We used the Adam optimizer with a learning rate of 0.001 and collected throughput and speedup numbers by executing twenty-five epochs of a standard training loop.

5.2 Characterization of the Molecular Graphs

In this work, we demonstrate the scalability of the accelerated SchNet model on the following two datasets with diverse characteristics.

HYDRONET: The Molecular Property Prediction task outlined in the HydroNet benchmark (Choudhury et al., 2020b) is as follows: given a molecule with specified spatial coordinate information, predict its energy. The HydroNet benchmark dataset is made up of 4.5M water cluster samples/molecules. Each sample molecule (graph) has between 9 and 90 atoms (vertices) and the task is to predict the energy E of the cluster. Of particular interest is the sparsity of the water cluster graphs, as demonstrated in Figure 5. Because edges are applied between nodes based on a limited spatial distance, physical constraints limit the number of atoms that can be packed into a region of space. Therefore, as the size of the cluster increases, so does the sparsity of the graph. We examine the full benchmark dataset, as well as a 2.7M subset containing clusters of size 9–75 nodes to examine the effect of reduced sparsity.

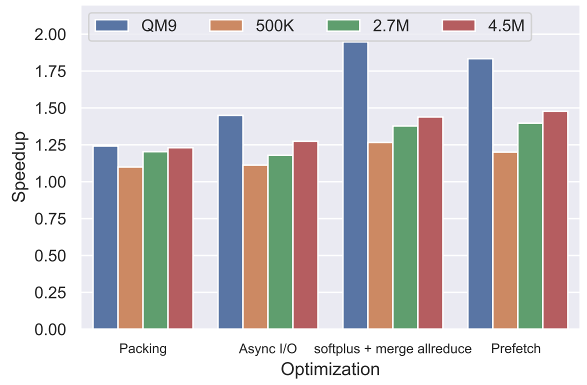
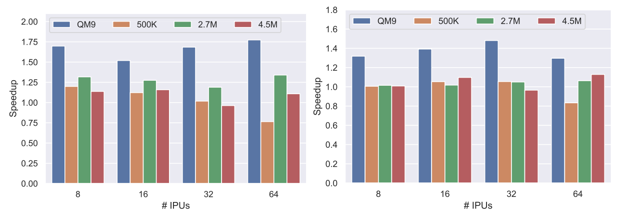


Figure 6. Speedup of the IPU SchNet model with different optimizations w.r.t. a baseline IPU implementation. The optimizations are added progressively from left to right. For example, the legend Async/IO implies packing with Async/IO. The experiments were conducted on 16 IPUs.



(a) Packing over padding. (b) Asynchronous I/O over the synchronous dataloader.

Figure 7. Effect of packing and asynchronous I/O at different scales. Here, all additional optimizations are enabled.

QM9: In contrast, the widely used QM9 molecular database (Ramakrishnan et al., 2014), which is limited to molecules with a maximum of 29 atoms, shows less sparsity in the corresponding molecular graphs (Figure 5). The limited number of atoms is the cause of the relatively dense molecular graph; however, increasing the number of atoms to examine, for example, liquid or condensed solid phases, would necessarily lead to increased sparsity similar to that of the HydroNet dataset. Because such small molecule datasets are commonly applied to train molecular NNs, we compare against the performance on the QM9 dataset.

These benchmarks are broadly representative for the current generation of electronic structure methods that are used to generate the training data for Molecular GNNs. One of the most widely used approximations in *ab initio* methods is density functional theory (DFT) which when applied to the modelling of molecules and materials is limited to systems of approximately one hundred atoms in routine calculations. This limit is not expected to be overcome any time soon despite decades of progress on linear scaling DFT methods that have yet to deliver the promise of quantum mechanical accuracy for molecules and materials composed of several thousands of atoms (Kohn, 1996; Prentice et al., 2020).

5.3 Evaluation of Different Optimization Techniques

5.3.1 *Evaluation of the Packing Technique*

We first evaluate the performance of the IPU SchNet model with both packing and padding. The experimental results are reported in Figure 6 and in Figure 7a. The speedup is calculated by taking the ratio of the average model training time with padding and packing. As can be observed from Figure 6, packing may improve the performance of the SchNet model by upto 25%, compared to the original padding. As the number of IPUs are increased (Figure 7a), packing technique becomes more impactful with the QM9 dataset due to its denser structure compared to the water cluster graphs as well as lower node counts (Figure 5). With larger 4.5M water cluster dataset and 64 IPUs, the packing technique outperforms the model with padding, as the compaction of the graphs enables processing more batches concurrently with more compute power.

We also evaluate the efficiency of our LPFHP batch packing algorithm compared to the padding technique (Figure 8). Efficiency is defined as the percentage of padding reduced by applying the LPFHP batch packing algorithm. With the QM9 dataset, padding may result in 38% wastage of memory where the maximum number of vertices in a graph is 29 (Figures 5 and 8). To determine the optimal maximum sequence length for packing, it is to be noted from the histogram of the number of nodes in graphs that sometimes the mode of the distribution is larger than half of the maximum number of nodes, as is the case for the QM9 as well as HydroNet dataset (Figure 5). In this case, if the maximum sequence length, i.e. maximum number of nodes in a pack, s_m is set to a smaller value, a substantial amount of padding would still be required (30% on QM9 compared to 38% with normal padding). Thus, it is beneficial to increase the allowed maximum number of nodes in a pack of graphs that gets processed jointly. For QM9, this can reduce the padding to less than 2% (Figure 8). Similar observation can be made for the two HydroNet datasets. In all cases, the curves are not smooth and contain several systematic spikes. This can be explained with the histograms, displayed in Figure 5. Only certain numbers of nodes occur in the dataset and there are several gaps and even for the existing numbers, the distributions are not smooth. This directly translates into the packing results.

5.3.2 *Impact of Asynchronous I/O*

In addition to applying packing technique, enabling asynchronous I/O improves the performance of the SchNet model, as reported in Figure 6 and Figure 7b. We observe that, with these two techniques, the QM9 dataset achieves better improvement in performance.

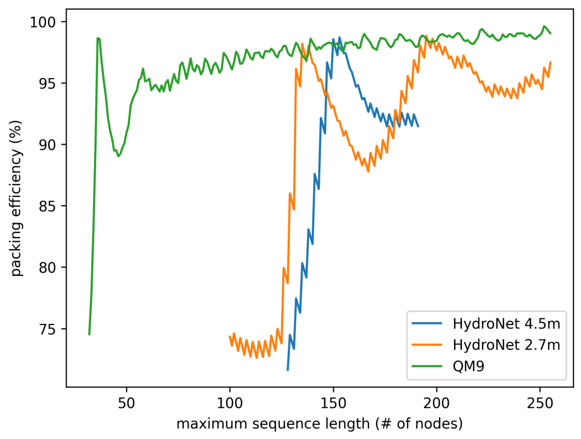


Figure 8. Effect of the maximum number of vertices in a pack (sequence length) on packing efficiency.

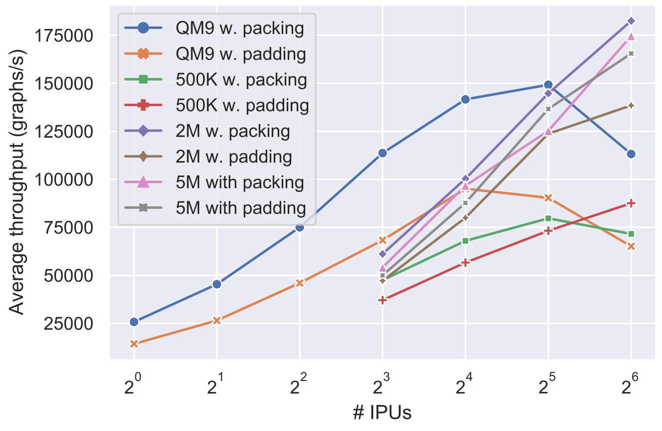


Figure 9. Strong scaling results of the optimized SchNet model on the IPUs with packing and padding. All additional optimizations are also applied in both cases.

5.3.3 *Impact of the optimized softplus, merging allreduce and prefetching*

Adding a cycle optimized implementation of the softplus activation and merging the all-reduce communincation collective improves the performance of training the SchNet model using IPUs for all the datasets (Figure 6). While prefetching improves performance with the 4.5M water cluster dataset, it negatively impacts the performance of the QM dataset. Here, the prefetch depth is set to 4.

5.4 Strong Scaling Results

We report the strong scaling training performance of the SchNet model on the IPUs in Figure 9. Here, we keep the dataset size constant while increasing the number of IPUs. The reported performance metric, average throughput, is calculated as the number of graphs processed per second (similar to the spirit of evaluating the number of sequences processed per second in training language models). As can be observed from the figure, in most cases, increasing the number of IPUs increases the average throughput. With smaller datasets such as QM9, however, the throughput is maximized with 32 IPUs (with packing). Beyond 32 IPUs, the performance deteriorates due to having not enough work to sustain higher throughput. With the 2.7M and 4.5M water cluster datasets and with packing technique, the average throughput continues to increase up through 64 IPUs, as there are sufficient available work to keep all the independent processors busy (We also report the per-epoch MSE loss with 2.7M water cluster dataset on 16 IPUs in Figure 11 of the Appendix due to space constraint).

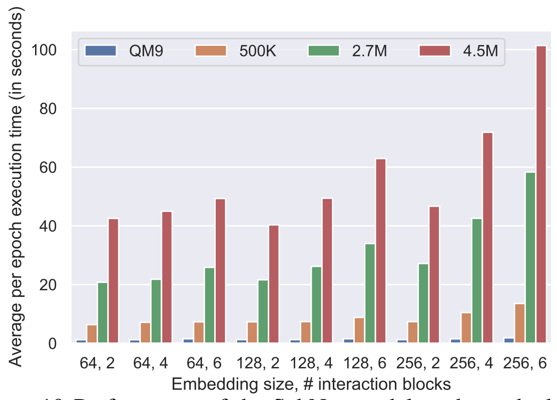


Figure 10. Performance of the SchNet model as the embedding size and the number of interaction blocks are varied.

5.5 Impact of Embedding size and the Number of Interaction layers

We also conducted experiments by varying the embedding size for each of the vertices of the input molecular graphs, as well as the number of interaction blocks in the SchNet model. We report the average per epoch execution time in Figure 10. Except for the (64, 2) and (128,2) embedding size and # interaction blocks combinations for the 4.5M dataset, as expected, the execution time increases with the increase of embedding size and the number of interaction blocks due to the increase in the total number of matrix multiplications performed in the SchNet model.

5.6 Average Per Epoch Execution Time with Different Number of IPUs

We also report the average per epoch execution time in seconds as we increase the number of IPUs in Table 1. With the QM9 dataset, the SchNet model achieves the best performance with 32 IPUs, closely followed by the performance while running on 16 IPUs. With QM9 dataset, on 64 IPUs, the overhead of communication starts to dominate the execution time instead of the available computation. With the 2.7M and 4.5M water cluster datasets, due to their larger sizes, increasing the number of IPUs help to achieve better performance as there are more local computation available.

Dataset	8 IPUs	16 IPUs	32 IPUs	64 IPUs	8GPUs
QM9	0.91	0.72	0.68	0.9	1.86
500K	8.39	5.36	5.0	5.57	6.87
2.7M	35.07	21.37	14.81	11.74	34.36
4.5M	62.56	35.0	27.03	19.38	60

Table 1. Average per epoch execution time in seconds with different number of IPUs and GPUs.

5.7 Hardware accelerator comparison

We compare the performance of training our optimized SchNet model for molecular GNNs on the IPUs with an existing out-of-the-box GPU implementation from the PyTorch Geometric (PyG) library. For running on multiple GPUs, we modified the existing single-GPU SchNet implementation in PyG to leverage the Distributed Data Parallel (DDP) model of the PyTorch library, without introducing any further optimizations. We conduct the experiments with different datasets for 25 epochs and report the result in Table 1. Here we report and compare the average per epoch execution time with 8 A100 GPUs and 16 IPUs because of their close equivalence in terms of compute capability, power and cost. As can be observed from Table 1, the SchNet model runs faster on the IPUs compared to the GPUs. In particular, with QM9, 500k water cluster, 2.7M water cluster and 4.5M water cluster datasets, the SchNet model running on the IPUs achieve a speedup of 2.58x, 1.28x, 1.6x and 1.71x respectively. The speedup is computed by taking the average of the execution time for 25 epochs for 16 IPUs and 8 GPUs then taking the ratio of the average runtime on the GPUs and on the IPUs.

6 Conclusion

The long history of science is full of examples where new technology has led to a deeper understanding of natural phenomena. More recently, progress in machine learning applied to computer vision and natural language processing has previously been driven by the re-purposing of GPU accelerators for the compute workloads of these deep neural networks (Krizhevsky et al., 2012; Vaswani et al., 2017). We present here the first application of IPUs to molecular GNNs and demonstrate extreme acceleration compared to a baseline GPU implementation. Beyond demonstrating this emerging capability, this paper proposes a set of optimization techniques for executing GNNs with molecular graphs: packing for efficient representation of the batches, planning for well-balanced work partitioning for optimized scatter-gather execution, overlapping computation and communication techniques for reducing the I/O bottleneck and model-specific optimizations. We demonstrate that altogether these optimizations can take advantage of the high-bandwidth local memory and faster interconnect of the IPU architecture to significantly improve the training time of the molecular GNNs. This has the potential to unlock the potential of machine learning applied to molecular-related scientific domains.

References

Addanki, R., Battaglia, P. W., Budden, D., Deac, A., Godwin, J., Keck, T., Li, W. L. S., Sanchez-Gonzalez, A., Stott, J., Thakoor, S., et al. Large-scale graph representation learning with very deep gnns and self-supervision. *arXiv preprint arXiv:2107.09422*, 2021.

Batzner, S., Musaelian, A., Sun, L., Geiger, M., Mailoa, J. P., Kornbluth, M., Molinari, N., Smidt, T. E., and Kozinsky, B. E (3)-equivariant graph neural networks for data-efficient and accurate interatomic potentials. *Nature communications*, 13(1):1–11, 2022.

Behler, J. First principles neural network potentials for reactive simulations of large molecular and condensed systems. *Angewandte Chemie International Edition*, 56 (42):12828–12840, 2017.

Bilbrey, J. A., Heindel, J. P., Schram, M., Bandyopadhyay, P., Xantheas, S. S., and Choudhury, S. A look inside the black box: Using graph-theoretical descriptors to interpret a Continuous-Filter Convolutional Neural Network (CF-CNN) trained on the global and local minimum energy structures of neutral water clusters. *The Journal of Chemical Physics*, 153(2):024302, 2020. doi: 10.1063/5.0009933.

Cai, Z., Yan, X., Wu, Y., Ma, K., Cheng, J., and Yu, F. Dgcl: An efficient communication library for distributed gnn training. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pp. 130–144, 2021.

Chanussot, L., Das, A., Goyal, S., Lavril, T., Shuaibi, M., Riviere, M., Tran, K., Heras-Domingo, J., Ho, C., Hu, W., et al. Open catalyst 2020 (oc20) dataset and community challenges. *ACS Catalysis*, 11(10):6059–6072, 2021.

Chiang, W.-L., Liu, X., Si, S., Li, Y., Bengio, S., and Hsieh, C.-J. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM International Conference on Knowledge Discovery & Data Mining*, 2019.

Choudhury, S., Bilbrey, J. A., Ward, L., Xantheas, S. S., Foster, I., Heindel, J. P., Blaiszik, B., and Schwarting, M. E. Hydronet: Benchmark tasks for preserving intermolecular interactions and structural motifs in predictive and generative models for molecular data. *NeurIPS Workshop on Physical Sciences*, 2020a.

Choudhury, S., Pope, J., Ward, L., Foster, I., Schwarting, M., Blaiszik, B., Heindel, J., and Xantheas, S. Hydronet: Benchmark tasks for preserving structural motifs and long-range interactions in predictive and generative models, 2020b. URL https://doi.org/10.18126/8PBB-YT6O.

Emani, M., Vishwanath, V., Adams, C., Papka, M. E., Stevens, R., Florescu, L., Jairath, S., Liu, W., Nama, T., and Sujeeth, A. Accelerating scientific applications with sambanova reconfigurable dataflow architecture. *Computing in Science & Engineering*, 23(2):114–119, 2021.

Fey, M. and Lenssen, J. E. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019a.

Fey, M. and Lenssen, J. E. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds (ICLR)*, 2019b.

Gandhi, S. and Iyer, A. P. P3: Distributed deep graph learning at scale. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pp. 551–568, 2021.

Geng, T., Li, A., Shi, R., Wu, C., Wang, T., Li, Y., Haghi, P., Tumeo, A., Che, S., Reinhardt, S., et al. Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 922–936. IEEE, 2020.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. Neural message passing for quantum chemistry. *arXiv preprint arXiv:1704.01212*, 2017.

Hamilton, W., Ying, Z., and Leskovec, J. Inductive representation learning on large graphs. In *Advances in neural information processing systems (NeurIPS)*, 2017.

Hosseini, R., Simini, F., and Vishwanath, V. Operation-level performance benchmarking of graph neural networks for scientific applications. *arXiv preprint arXiv:2207.09955*, 2022.

Jha, D., Gupta, V., Ward, L., Yang, Z., Wolverton, C., Foster, I., Liao, W.-k., Choudhary, A., and Agrawal, A. Enabling deeper learning on big data for materials informatics applications. *Scientific reports*, 11(1):1–12, 2021.

Jia, Z., Tillman, B., Maggioni, M., and Scarpazza, D. P. Dissecting the graphcore ipu architecture via microbenchmarking. *arXiv preprint arXiv:1912.03413*, 2019.

Jia, Z., Lin, S., Gao, M., Zaharia, M., and Aiken, A. Improving the accuracy, scalability, and performance of graph neural networks with roc. *Proceedings of Machine Learning and Systems*, 2:187–198, 2020.

Johnson, D. S. *Near-optimal bin packing algorithms*. PhD thesis, Massachusetts Institute of Technology, 1973.

Johnson, D. S. and Garey, M. R. A 7160 theorem for bin packing. *Journal of Complexity*, 1(1):65–106, oct 1985. ISSN 0885064X. doi: 10.1016/0885-064X(85)90022-6. URL https://linkinghub.elsevier.com/retrieve/pii/0885064X85900226.

Joshi, R. P., Gebauer, N. W., Bontha, M., Khazaieli, M., James, R. M., Brown, J. B., and Kumar, N. 3d-scaffold: A deep learning framework to generate 3d coordinates of drug-like molecules with desired scaffolds. *The Journal of Physical Chemistry B*, 125(44):12166–12176, 2021.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pp. 1–12, 2017.

Kaler, T., Stathas, N., Ouyang, A., Iliopoulos, A.-S., Schardl, T., Leiserson, C. E., and Chen, J. Accelerating training and inference of graph neural networks with fast sampling and pipelining. *Proceedings of Machine Learning and Systems*, 4:172–189, 2022.

Karypis, G. and Kumar, V. Multilevelk-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed computing*, 48(1):96–129, 1998.

Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. *International Conference on Learning Representations (ICLR)*, 2017.

Klicpera, J., Groß, J., and Günnemann, S. Directional message passing for molecular graphs. *arXiv preprint arXiv:2003.03123*, 2020.

Kohn, W. Density functional and density matrix method scaling linearly with the number of atoms. *Phys. Rev. Lett.*, 76:3168–3171, Apr 1996. doi: 10.1103/PhysRevLett.76.3168. URL https://link.aps.org/doi/10.1103/PhysRevLett.76.3168.

Kohn, W. and Sham, L. J. Self-consistent equations including exchange and correlation effects. *Phys. Rev.*, 140:A1133–A1138, Nov 1965. doi: 10.1103/PhysRev.140.A1133. URL https://link.aps.org/doi/10.1103/PhysRev.140.A1133.

Korte, B. and Vygen, J. *Combinatorial Optimization*, volume 21 of *Algorithms and Combinatorics*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-24487-2. doi: 10.1007/978-3-642-24488-9. URL http://link.springer.com/10.1007/978-3-642-24488-9.

Krell, M. M., Kosec, M., Perez, S. P., and Fitzgibbon, A. Efficient sequence packing without cross-contamination: Accelerating large language models without impacting performance, 2021. URL https://arxiv.org/abs/2107.02027.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In Pereira, F., Burges, C., Bottou, L., and Weinberger, K. (eds.), *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

Kulichenko, M., Smith, J. S., Nebgen, B., Li, Y. W., Fedik, N., Boldyrev, A. I., Lubbers, N., Barros, K., and Tretiak, S. The rise of neural networks for materials and chemical dynamics. *The Journal of Physical Chemistry Letters*, 12 (26):6227–6243, 2021.

Lee, C. C. and Lee, D. T. A Simple On-Line Bin-Packing Algorithm. *Journal of the ACM (JACM)*, 32 (3):562–572, jul 1985. ISSN 1557735X. doi: 10.1145/3828.3833. URL https://dl.acm.org/doi/10.1145/3828.3833.

Lim, J., Ryu, S., Park, K., Choe, Y. J., Ham, J., and Kim, W. Y. Predicting drug–target interaction using a novel graph neural network with 3d structure-embedded graph representation. *Journal of chemical information and modeling*, 59(9):3981–3988, 2019.

Lubbers, N., Smith, J. S., and Barros, K. Hierarchical modeling of molecular energies using a deep neural network. *The Journal of Chemical Physics*, 148 (24):241715, 2018. doi: 10.1063/1.5011181. URL https://doi.org/10.1063/1.5011181.

Ma, L., Yang, Z., Miao, Y., Xue, J., Wu, M., Zhou, L., and Dai, Y. Neugraph: parallel deep neural network computation on large graphs. In *USENIX Annual Technical Conference*, 2019.

Moe, J., Pogorelov, K., Schroeder, D. T., and Langguth, J. Implementating spatio-temporal graph convolutional networks on graphcore ipus. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 45–54. IEEE, 2022.

Morari, A., Tumeo, A., Chavarría-Miranda, D., Villa, O., and Valero, M. Scaling irregular applications through data aggregation and software multithreading. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pp. 1126–1135. IEEE, 2014.

Pinheiro, G. A., Mucelini, J., Soares, M. D., Prati, R. C., Da Silva, J. L., and Quiles, M. G. Machine learning prediction of nine molecular properties based on the smiles representation of the qm9 quantum-chemistry dataset. *The Journal of Physical Chemistry A*, 124(47):9854–9866, 2020.

PopTorch, G. Poptorch: Pytorch integration for the graphcore ipu, 2022. URL https://github.com/graphcore/poptorch.

Prentice, J. C. A., Aarons, J., and Womack, J. C. The ONETEP linear-scaling density functional theory program. *J. Chem. Phys*, 152:174111, 2020. doi: 10.1063/5.0004445. URL https://doi.org/10.1063/5.0004445.

Prodan, E. and Kohn, W. Nearsightedness of electronic matter. *Proceedings of the National Academy of Sciences*, 102(33):11635–11638, aug 2005. doi: 10.1073/pnas.0505436102. URL https://doi.org/10.1073%2Fpnas.0505436102.

Rakshit, A., Bandyopadhyay, P., Heindel, J. P., and Xantheas, S. S. Atlas of putative minima and low-lying energy networks of water clusters n= 3–25. *The Journal of chemical physics*, 151(21):214307, 2019.

Ramakrishnan, R., Dral, P. O., Rupp, M., and von Lilienfeld, O. A. Quantum chemistry structures and properties of 134 kilo molecules. *Scientific Data*, 1, 2014.

Schmidt, J., Marques, M. R., Botti, S., and Marques, M. A. Recent advances and applications of machine learning in solid-state materials science. *npj Computational Materials*, 5(1):1–36, 2019.

Schwaller, P., Laino, T., Gaudin, T., Bolgar, P., Hunter, C. A., Bekas, C., and Lee, A. A. Molecular transformer: a model for uncertainty-calibrated chemical reaction prediction. *ACS central science*, 5(9):1572–1583, 2019.

Schütt, K. T., Arbabzadah, F., Chmiela, S., Müller, K. R., and Tkatchenko, A. Quantum-chemical insights from deep tensor neural networks. *Nature Communications*, 8(1):13890, 2017. ISSN 2041-1723. doi: 10.1038/ncomms13890. URL https://doi.org/10.1038/ncomms13890.

Schütt, K. T., Sauceda, H. E., Kindermans, P.-J., Tkatchenko, A., and Müller, K.-R. Schnet – a deep learning architecture for molecules and materials. *The Journal of Chemical Physics*, 148(24):241722, 2018. doi: 10.1063/1.5019779.

Schütt, K. T., Kessel, P., Gastegger, M., Nicoli, K. A., Tkatchenko, A., and Müller, K.-R. Schnetpack: A deep learning toolbox for atomistic systems. *Journal of Chemical Theory and Computation*, 15(1):448–455, 2019. doi: 10.1021/acs.jctc.8b00908.

Smith, J. S., Isayev, O., and Roitberg, A. E. Ani-1: an extensible neural network potential with dft accuracy at force field computational cost. *Chem. Sci.*, 8:3192–3203, 2017. doi: 10.1039/C6SC05720A. URL http://dx.doi.org/10.1039/C6SC05720A.

Sussman, J. L., Lin, D., Jiang, J., Manning, N. O., Prilusky, J., Ritter, O., and Abola, E. E. Protein data bank (pdb): database of three-dimensional structural information of biological macromolecules. *Acta Crystallographica Section D: Biological Crystallography*, 54(6):1078–1084, 1998.

Tripathy, A., Yelick, K., and Buluç, A. Reducing communication in graph neural network training. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14. IEEE, 2020.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need, 2017. URL https://arxiv.org/abs/1706.03762.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., and Bengio, Y. Graph attention networks. In *International Conference on Learning Representations (ICLR)*, 2018.

Wan, C., Li, Y., Wolfe, C. R., Kyrillidis, A., Kim, N. S., and Lin, Y. Pipegcn: Efficient full-graph training of graph convolutional networks with pipelined feature communication. In *International Conference on Learning Representations*, 2022.

Wang, M., Yu, L., Zheng, D., Gan, Q., Gai, Y., Ye, Z., Li, M., Zhou, J., Huang, Q., Ma, C., Huang, Z., Guo, Q., Zhang, H., Lin, H., Zhao, J., Li, J., Smola, A. J., and Zhang, Z. Deep graph library: Towards efficient and scalable deep learning on graphs. *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

Wang, Y., Pan, Y., Davidson, A., Wu, Y., Yang, C., Wang, L., Osama, M., Yuan, C., Liu, W., Riffel, A. T., et al. Gunrock: Gpu graph analytics. *ACM Transactions on Parallel Computing (TOPC)*, 4(1):1–49, 2017.

Wang, Y., Feng, B., and Ding, Y. Tc-gnn: Accelerating sparse graph neural network computation via dense tensor core on gpus. *arXiv preprint arXiv:2112.02052*, 2021a.

Wang, Y., Feng, B., Li, G., Li, S., Deng, L., Xie, Y., and Ding, Y. Gnnadvisor: An efficient runtime system for gnn acceleration on gpus. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*, 2021b.

Wang, Y., Feng, B., Li, G., Li, S., Deng, L., Xie, Y., and Ding, Y. {GNNAdvisor}: An adaptive and efficient runtime system for {GNN} acceleration on {GPUs}. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pp. 515–531, 2021c.

Wang, Y., Wang, J., Cao, Z., and Barati Farimani, A. Molecular contrastive learning of representations via graph neural networks. *Nature Machine Intelligence*, 4(3):279–287, 2022.

Xu, K., Hu, W., Leskovec, J., and Jegelka, S. How powerful are graph neural networks? In *International Conference on Learning Representations (ICLR)*, 2019.

Yue, M. and Zhang, L. A simple proof of the inequality $MFFD(L) \leq 71/60 OPT(L) + 1, L$ for the MFFD bin-packing algorithm. *Acta Mathematicae Applicatae Sinica*, 11(3):318–330, jul 1995. ISSN 01689673. doi: 10.1007/BF02011198.

Zeng, H., Zhou, H., Srivastava, A., Kannan, R., and Prasanna, V. Graphsaint: Graph sampling based inductive learning method. In *International Conference on Learning Representations*, 2019.

Zhang, S., Liu, Y., and Xie, L. Molecular mechanics-driven graph neural network with multiplex graph for molecular structures. *arXiv preprint arXiv:2011.07457*, 2020.

Zheng, D., Ma, C., Wang, M., Zhou, J., Su, Q., Song, X., Gan, Q., Zhang, Z., and Karypis, G. Distdgl: distributed graph neural network training for billion-scale graphs. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pp. 36–44. IEEE, 2020a.

Zheng, D., Song, X., Ma, C., Tan, Z., Ye, Z., Dong, J., Xiong, H., Zhang, Z., and Karypis, G. Dgl-ke: Training knowledge graph embeddings at scale. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 739–748, 2020b.

Zheng, D., Song, X., Yang, C., LaSalle, D., and Karypis, G. Distributed hybrid cpu and gpu training for graph neural networks on billion-scale heterogeneous graphs. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 4582–4591, 2022.

A Appendix

A.1 Additional Experimental results

A.1.1 *Per Epoch MSE Loss for SchNet IPU*

We report per epoch MSE loss for the SchNet model with 2.7M water cluster dataset in Figure 11.

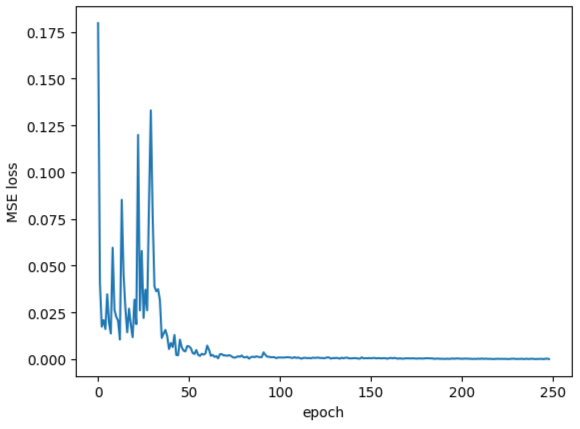


Figure 11. Per epoch MSE loss for the SchNet model with 2.7M water cluster dataset.

A.1.2 *Profiler Output to Demonstrate the Effect of Merging Communication Collectives*

We show the effect of merging communication collectives for weight updates (discussed in Section 4.3) by profiling with and without this optimization in Figure 12.

A.1.3 *Per Epoch Execution time With Different Number of IPUs*

We report per epoch execution time while varying the number of IPUs in Figure 13.

A.2 Extended Discussion on Related Work

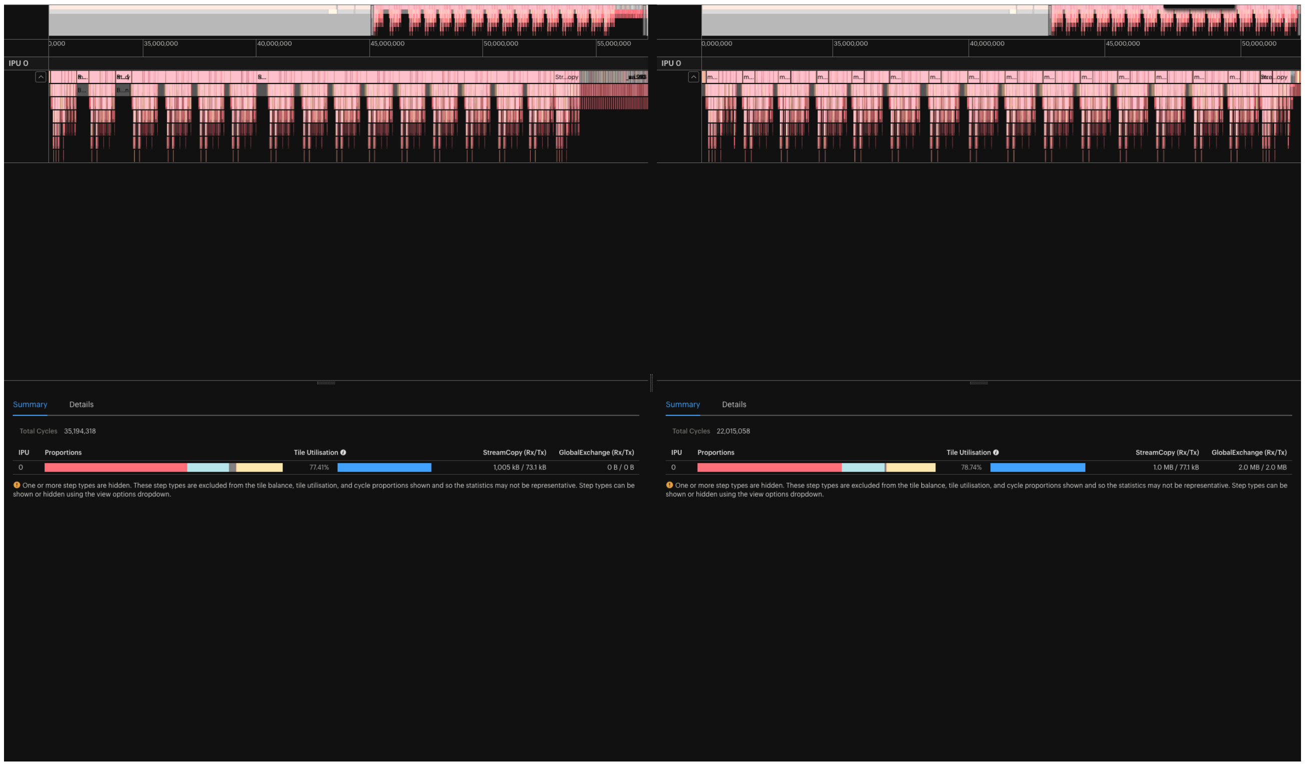
A plethora of frameworks has been proposed in the literature that seek to improve the performance of the GNNs (Jia et al., 2020; Zheng et al., 2020a; Fey & Lenssen, 2019b; Cai et al., 2021; Ma et al., 2019; Wang et al., 2021c). Among these frameworks, Pytorch Geometric (PyG) (Fey & Lenssen, 2019b) is the most well-known library for GNNs and leverages the torch-scatter library for graph aggregation operation. A recent work by Hosseini et al.(Hosseini et al., 2022) reported profiling results for the low-level operations in PyG and reported that, on the GPUs, memory is the main bottleneck in GNN execution. Deep Graph Library (DGL) (Wang et al., 2019) is another popular GNN library that uses the Sparse Matrix multiplication (SpMM) kernels from CuSparse for sum-reduction aggregation. NeuGraph (Ma et al., 2019) introduced the "Scatter-ApplyEdge-Gather-ApplyVertex" abstraction for GNN computation. Kaler et al. (Kaler et al., 2022) proposed customized neighborhood sampler, shared-memory parallelization and pipelining of batch transfer with GPU computation for improving GPU utilization. Other pipelining techniques for overlapping computation with communication have also been suggested (Wan et al., 2022).

A.2.1 *GPU-related Optimizations for GNNs*

To date, both algorithmic and system-level optimizations for GNNs have been mostly geared towards Graphic Processing Units (GPUs) (Kipf & Welling, 2017; Xu et al., 2019; Ma et al., 2019; Wang et al., 2019; Fey & Lenssen, 2019b) due to the availability of the GPUs (Nvidia A100, AMD MI200, etc.) on many current HPC systems. At the algorithm level, neighborhood sampling and mini batch processing have been proposed for reducing communication (Chiang et al., 2019; Zeng et al., 2019; Gandhi & Iyer, 2021). Alternatively, techniques including pipelined batch transfer and computation (Kaler et al., 2022), specialized runtime and communication libraries (Cai et al., 2021; Ma et al., 2019; Wang et al., 2021c) seek to improve the performance of the GNN libraries at the system level. Moreover, specialized hardware, such as tensor cores inside the GPUs have been adapted to accelerate the sparse-dense matrix multiplication step in the aggregation phase of the GNNs (Wang et al., 2021a). GPUs primarily rely on the oversubscription of threads to hide memory latency as well as coalesced memory access within each warp to achieve peak memory bandwidth. For these reasons, GPUs perform well when the computation is regular and dense. However, fine-grained, irregular workloads such as GNNs require significant engineering to obtain optimal performance on the GPUs (Wang et al., 2017).

A.2.2 *IPUs and ML Workload*

For and overview of the IPU hardware and various benchmarking results on these systems, please see the white-paper (Jia et al., 2019). Discussion about an implementation of Spatio-Temporal Graph Convolutional Networks on Graphcore IPUs can be found in (Moe et al., 2022).



Total Cycles 35,194,318

IPU	Proportions	Tile Utilisation	StreamCopy (Rx/Tx)	GlobalExchange (Rx/Tx)
0	🟥🟦🟨	77.41%	1,005 kB / 73.1 kB	0 B / 0 B

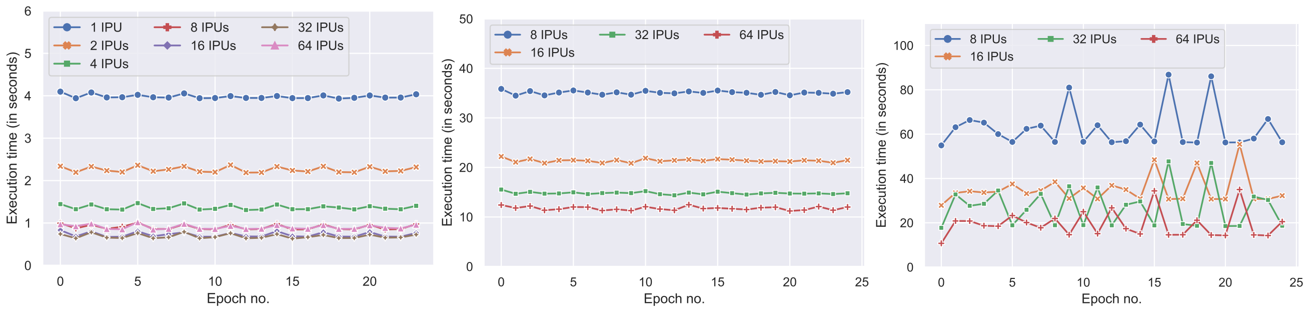
One or more step types are hidden. These step types are excluded from the tile balance, tile utilisation, and cycle proportions shown and so the statistics may not be representative. Step types can be shown or hidden using the view options dropdown.

Total Cycles 22,015,058

IPU	Proportions	Tile Utilisation	StreamCopy (Rx/Tx)	GlobalExchange (Rx/Tx)
0	🟥🟦🟨	78.74%	1.0 MB / 77.1 kB	2.0 MB / 2.0 MB

One or more step types are hidden. These step types are excluded from the tile balance, tile utilisation, and cycle proportions shown and so the statistics may not be representative. Step types can be shown or hidden using the view options dropdown.

Figure 12. Effect of merging of all the allreduce collectives at the end of the backward pass to reduce the tail latency of the execution. On the left profile, from at around 52k time unit mark till the end, only some of the tiles are busy while other tiles are waiting. In contrast, on the right, till the very end of the execution, most of the tiles are engaged reasonably in computation and communication without long waiting period, thanks to the merging of the all reductions.



(a) QM9 (b) 2.7M water cluster (c) 4.5M water cluster

Figure 13. Per epoch performance of the SchNet model with different number of IPUs and with packing (25 epochs).