# Trainium Optimization for GNNs and Diffusion Speculative Decoding

**Sambhav Gupta** [1] **Joshua Delgadillo** [1]

## Abstract

Machine learning accelerators generally target standard transformer decoder workloads, leaving other machine learning architectures underoptimized in many cases. We run Graph Neural Network (GNN) and diffusion algorithms for speculative decoding on Trainium hardware, optimizing at the compiler level and distributed level respectively. We achieve up to 67% speedups on GNN algorithms with 67-fold reductions on HBM write pressure via the standard compiler flow, and optimize transformer decoding up to 33% faster by implementing DFlash speculative decoding for Trainium hardware.

## 1. Introduction and Problem Statement

The landscape of machine learning accelerators is rapidly advancing and widening, but the majority of workloads and overall model flops computed on these devices is transformer decoder inference. This leaves other workloads with differing compute and memory profiles underoptimized via standard least-resistance pipelines, like native PyTorch (13). Indeed, in the case of AWS Trainium, there are many optimizations, libraries, and kernels written for the LLM inference pipeline, but much fewer integrations with other types of models such as Graph Neural Networks (GNNs) and diffusion models.

Because many types of machine learning accelerators are optimized for transformer inference at the hardware level (for example, by concentrating model flops in systolic array-based tensor cores), there is a need for research into optimizing accelerator software for non-standard workloads. This improves the utility of accelerators and narrows the gap between raw hardware capability and realized performance on diverse workloads.

Upon exploring various workloads, we focus on graph neural networks (GNNs) and diffusion-based speculative decoding (DFlash) as non-standard Trainium2 workloads.

[1]Stanford University, Stanford, CA, USA. Correspondence to: Sambhav Gupta <samgupta@stanford.edu>.

## 2. Background and Related Work

### 2.1. Trainium Architecture and `NKI`

Trainium is an ML hardware accelerator developed by Amazon Web Services. We focus specifically on Trainium2. The chip contains 4 specialized processors and 3 levels of memory in its hierarchy. The Tensor Engine is a 128x128 systolic array meant primarily to accelerate matrix operations. The Vector Engine accelerates operations where each element of output may be dependent on more than one element of input (i.e., saxpy operations, layer norms, pooling). The Scalar Engine is meant for ops where each element of output only depends on one element of input. Finally, the General-Purpose SIMD Engine (GPSIMD) can execute arbitrary C++ kernels for any logic that does not map onto one of the specialized processors.

The Trainium memory hierarchy is organized into the High-Bandwidth Memory (HBM), which is the largest capacity but slowest "off-chip" memory; State Buffer (SBUF), scratchpad on-chip memory; and a very small Partial-Sum Buffer (PSUM) meant specifically for the Tensor Core to accumulate matmul results. A typical pattern is to load tiles of large tensors from HBM into SBUF, compute on them in SBUF/PSUM, then store back to HBM. Additionally, SBUF and PSUM are 2-dimensional, with 128 rows (the "partition" dimension) and a much larger number of columns (the "free" dimension). Most memory accesses exhibit a number of restrictions on the partition dimension (e.g., no strided accesses in most situations), because the partition dimension frequently enables parallelism. Data movement between HBM and SBUF is handled by the 16 DMA engines per Neuron core.

Developers can program this behavior through the Neuron Kernel Interface (`NKI`). `NKI` is an embedded Python DSL that will get lowered to `NeuronISA` (actual hardware instructions) by the closed-source Neuron compiler (`neuronx-cc`). `NKI` gives developers the ability to explicitly manage HBM $\leftrightarrow$ SBUF data movement, tiling strategies, loop ordering, etc. However, the compiler still has the final say over many scheduling elements such as pipelining and memory-compute synchronization.

### 2.2. PyTorch for AWS Trainium

AWS Trainium integrates with PyTorch by way of the existing XLA backend. XLA is an open-source ML compiler that translates code from frameworks such as PyTorch and JAX into hardware instructions. First, the ML framework must emit an HLO (High Level Operation) graph—this is XLA's IR. Then, the XLA compiler will make several optimization passes on this graph. Finally, the optimized graph will be lowered to hardware instructions such as C++/OpenMP for CPUs or PTX for GPUs. The case for AWS Trainium is similar: the `torch_xla` package traces the PyTorch graph into an HLO graph, which the XLA compiler will optimize. At this stage, AWS' `neuronx-cc` takes over and lowers the HLO graph into `NeuronISA` instructions. Additionally, developers can mix custom `NKI` kernels with PyTorch via the `@nki.jit` decorator. On the backend, these kernels are inserted as a `CustomCall` op within the HLO graph. Once `neuronx-cc` receives the HLO graph, it will lower the HLO ops and the `CustomCall` ops to `NeuronISA`.

### 2.3. Graph Neural Networks

Graph Neural Networks (GNNs) are a class of deep learning models that operate on the natural topology of a network, generalizing operations like linear layers and convolutions to arbitrary connectivities. This is especially useful in fields like social networks and molecular structures, which are inherently relational.

Most architectures involve a "message passing" step, wherein each node aggregates feature vectors from its neighboring nodes. Implementing this step is often a challenge, especially on custom hardware, as accessing the node's neighbors is an irregular "gather" operation, and batch writing the aggregated features for a potentially non-contiguous set of nodes is a "scatter" operation. These operations are especially perilous on Trainium, as the architecture is set up to maximize memory bandwidth for contiguous accesses in the partition dimension.

### 2.4. DFlash Background

Autoregressive transformer decoder-based LLM inference generally has two stages: prefill, where all tokens in the prompt are propagated to populate the KV cache, and decode, where single tokens are generated sequentially. At a batch size of one, the decode operation massively underutilizes most machine learning accelerators, as there is immense memory pressure to load the model weights from HBM for just one token. This pressure is mitigated for larger batch sizes since the model need only be loaded once for multiple tokens across the batch.

Speculative decoding attempts to mitigate this bottleneck by delegating the generation of future tokens to a smaller, faster draft model. The target LLM then verifies these draft tokens in parallel, allowing it to decode with width larger than 1. The longest speculated prefix that is consistent with the target model is then appended to the generation.

DFlash (Block Diffusion for Flash Speculative Decoding) (3) is a technique that uses a diffusion LLM to generate these speculated tokens in wide batches of 16, which is an advantage over autoregressive approaches like EAGLE-3 (1) that can have similar MFU issues as the target model due to single-token sequential decoding. By implementing and optimizing DFlash for Trainium hardware, we can significantly improve the MFU of Trainium hardware under this low-throughput, low-latency regime.

## 3. Implementation

### 3.1. Graph Generation and Preprocessing

To develop a baseline, we first implemented a standard GNN, GraphSAGE (7) in PyTorch. Our (randomly generated, in this case) graph is encoded by `src` and `dst` tensors of dimension `(num_edges,)` where the `k`th edge is `src[k]->dst[k]`. We also encode each node's feature vector in a tensor `x` of dimension `(num_nodes, feat_dim)`. Importantly, we perform 2 common CPU-side pre-processing steps:

1. In order to enable contiguous output writes, we sort `src` and `dst` by `dst`. This way, all edges with destination `n_d` will be contiguous.
2. In order to avoid maintaining a separate tensor of start and end indices inside `src` that correspond to the window of edges with destination `n_d`, we pad `src` so that each node has `max_deg` incoming edges and construct a mask `src_mask` that has a one where this edge actually exists and a 0 where it is just padding. This simplifies the indexing: `src` is now a `(num_nodes, max_degree, feat_dim)` tensor, and for destination node `n_d`, we can find the sources of all incoming edges from `src[n_d*max_deg:(n_d+1)*max_deg] * src_mask[n_d*max_deg:(n_d+1)*max_deg]`.

These pre-processing steps are valid because the graph structure is fixed, so this only needs to be done once before launching the model in its entirety.

### 3.2. Graph Profiling

In order to draw accurate comparisons between PyTorch and custom `NKI` implementations, we set up a profiling harness. For each of our experiments, we defined a `run_torch.py` with the PyTorch code we'd like to beat. Then, we defined a `run_nki.py` that was *the*

*same as the PyTorch code* except for the specific operation(s) that we were replacing with an `NKI` kernel. Since each script is wrapped by PyTorch, we use the `torch_xla.core.xla_model.mark_step()` function to produce a `.neff` binary (the result of the aforementioned XLA+`neuronx-cc` compilation flow).

We then created a profile script that would produce this artifact for both the pure PyTorch and `NKI` scripts in an experiment, and run `neuron-profile capture -n <neff> -s <ntff>` on each of them to obtain profile traces and execution times, averaging over 3 trials to compute a relative speedup.

### 3.3. DFlash Implementation

DFlash speculative decoding works by running a Qwen3 (15)-based diffusion model to transform noise tokens into predicted tokens using model hidden states as context.

The dataflow is as follows: in the starting state, the target model prefills some $n$ tokens. For some subset of the target model's layers, the hidden states are extracted across the entire sequence. At each position, these hidden states are concatenated and projected down to a smaller vector, forming that token position's "context vector". For each of the five layers of the DFlash model, these context vectors are projected to a key and value vector, forming the same shape of primitive as a KV cache. The Qwen3-based model then transforms a block of noise tokens through five regular Qwen3 layers, each time using a self attention layer on these keys and values; queries are derived from the noise vectors, and the noise vectors provide keys and values themselves which are concatenated onto the context KV cache.

The final noise states are projected down to token distributions by the target model's LM head, and these distributions form the speculated prediction. The target model prefills this predicted draft block, accepting the longest prefix with which it agrees, before returning to the draft stage again. The speedup comes from the ability to run the target model on more than one token at a time, increasing the MFU of the bandwidth-bound decode stage.

The noise-based KV vectors are evicted and replaced with verified token context-based vectors as soon as the corresponding token has been verified by the target model.

Our implementation must abide by strict rules for neuron compilation, namely that all shapes must be fixed; Neuronx tracing is much more challenging with dynamic shapes, as it can involve recompiling the entire model graph for every possible shape and combination of shapes. This presents a challenge for the KV cache, as it grows with sequence length. Rather than adapt the compilation graph with dynamic shapes, we simply adapt a sliding window attention-style optimization where we keep a fixed window size of $496$ (one block less than $512$) and instead of keeping all `sequence_length` tokens in the KV cache, we evict those farther than $496$ behind the current position and only attend among $496 + 16$ prefilled and drafted tokens.

This tends to work well in practice; we theorize this is because many of the drafted tokens are low-entropy, predictable tokens which do not require long-context recall to predict accurately.

## 4. Experimental Setup

### 4.1. GNN Optimizations

We followed a methodical process to optimize a variety of GNN architectures. We started by profiling the baseline PyTorch implementation, then optimizing the most minimal kernel of the model, and gradually began fusing more functionality into it. The kernels we optimized were:

#### 4.1.1. SCATTER-ADD

The most minimal kernel in the "message passing" layer common to many GNNs is the "scatter-add". This kernel comes after the feature vectors for source nodes have already been "gathered" and performs the reduction of feature vectors that correspond to the same destination node. Formally, given a tensor of shape `(num_nodes, max_deg, feat_dim)`, where the first dimension corresponds to the destination node, and the latter 2 dimensions correspond to the feature vector for each of the source nodes connected to that destination node, we sum over the `max_deg` dimension for a result tensor of shape `(num_nodes, feat_dim)`. Since this aggregation happens over a middle dimension, it is known as a **strided reduction**.

After profiling the pure PyTorch version, we saw an excessive amount of instructions issued to the Scalar Engine for control flow purposes, suggesting that the compiler was not properly batching the reduction. This motivated our approach to chunk the reduction over the degree dimension. We load a tile of `(tile_nodes=128, tile_degree=16, feat_dim)` into SBUF from HBM, and then reduce it in SBUF. The `tile_nodes=128` matches the SBUF partition dimension size and forces the compiler to utilize the Vector Engine's entire parallel capacity by computing 128 reductions in parallel. The `tile_degree=16` is a tuned value that appears to best allow the compiler to pipeline the computations for a single reduction.

#### 4.1.2. GATHER-SCATTER

Since `scatter-add` assumed the relevant node feature vectors had already been gathered, we next tried to optimize that gathering step into the `NKI` kernel. Formally, we'd like

to construct the `(num_nodes, max_deg, feat_dim)` tensor `messages = x[src]` where `x` is our tensor of node features and `src` is the tensor of node indices that begin an edge, sorted by their destination. This is the input to the `scatter-add` kernel.

After profiling the PyTorch implementation, we noticed an excessive quantity of reads/writes to/from HBM for the size of the tensor. This guided our approach to fuse both the gather and scatter together, to avoid the many round trips to HBM. Essentially, we keep the same tile dimensions of `(tile_nodes=128, tile_degree=16, feat_dim)` from `scatter-add`, and use the DMA's limited gather functionality to populate the tile. Then, we compute the `scatter-add` reduction on just that tile in the same way as before. This aims to significantly reduce the amount of HBM round-trips that are necessary.

#### 4.1.3. GNN Generalizability

The `gather-scatter` kernel is the core of the "message passing" layer fundamental to many GNN architectures. The two tunable parameters our kernel exposes are:

1. Reduction normalization constant — the divisor applied to the sum of neighboring nodes' feature vectors (e.g., degree for mean pooling).

2. Mask — implemented as part of the padding preprocessing for our `src` and `dst` tensors. By default, `mask[d, s]` is 1 if the edge exists, and 0 otherwise.

To confirm this generalizability, we inserted our `gather-scatter` kernel into the following GNN architectures:

- *GraphSAGE* - Reduces the feature vectors of neighbor nodes via the mean
  - Reduction Normalization Constant = $1/\deg(n)$, Mask = Default
- *Graph Convolution Network* (10) — Reduces the feature vectors of neighbor nodes via a sum normalized with $(\deg(i)\deg(j))^{-1/2}$
  - Reduction Normalization Constant = 1, Mask = $(\deg(i)\deg(j))^{-1/2}$
- *Graph Isomorphism Network* (16) — Reduces the feature vectors with a simple sum, later fed into an MLP
  - Reduction Normalization Constant = 1, Mask = Default

### 4.2. DFlash Sampling

Our experimental design is relatively simple, and mostly replicates the design of experiments conducted in the original DFlash formulation. We use Qwen3 4B as implemented in the NxD inference library on a trn2.3xlarge AWS EC2 instance. We evaluate tasks in three categories: Math: GSM8K (5), MATH-500 (11), AIME 2024 (12), and AIME 2025 (12); Code: HumanEval (4), MBPP (2), LBPP (6), SWE-bench (9), and LiveCodeBench (8); Chat: MT-Bench (17) and Alpaca (14). As DFlash did, for each task, we assess the performance of the draft models using average acceptance length ($\tau$) and end-to-end decoding speedup over the autoregressive baseline. We sample at temperature $1.0$ with top_p $0.9$ for $4096$ tokens.

## 5. Results and Evaluation

### 5.1. GNN Results

Table 1 shows the profiled execution times for each of these experiments, each of which saw a speedup from the `NKI` kernel. For `scatter_add`, these results show that our tiling strategy was an effective hint to the compiler for how to schedule and batch more effectively. For `gather_scatter`, there are more insights. Table 2 shows the memory traffic improvements from our fused gather-scatter.

The reduction by 67× in HBM writes suggests that doing both the gather and scatter inside one tile avoided materializing the massive intermediate tensor (`num_nodes, max_degree, feat_dim`) and making continual round trips to and from HBM. Since the kernel is highly memory bound (DMA time is over 98% of kernel time), this made a significant improvement in runtime.

Finally, our last 3 experiments on the generalizability of the kernel show that across GNN architectures, speedups are achievable. In fact, the speedups seen in full networks are even greater than those of the kernels themselves. This is likely due to the larger dimensions magnifying the effects of speedups compared to kernel overhead, as well as the compounding nature of kernel speedups when applied to multiple layers.

| Experiment | Torch (ms) | `NKI` (ms) | Speedup |
|---|---|---|---|
| `scatter_add` | 0.159 | **0.139** | 1.14× |
| `gather_scatter` | 1.900 | **1.737** | 1.09× |
| `graphsage` | 5.926 | **3.539** | 1.67× |
| `gcn_nki` | 5.187 | **3.510** | 1.48× |
| `gin_nki` | 5.223 | **3.525** | 1.48× |

*Table 1.* Average kernel execution time (3 trials) on Trainium2 (`NKI` vs. PyTorch baseline).

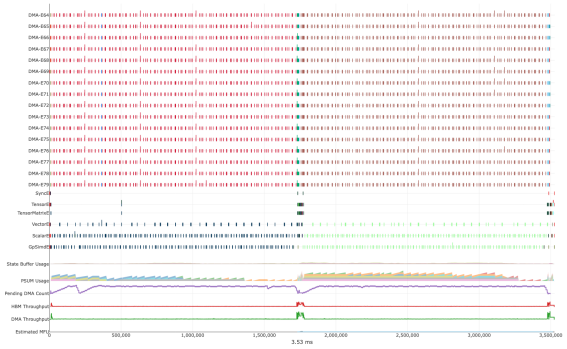

*Figure 1.* `neuron-profile` trace for 2-layer GraphSAGE (`NKI` implementation)

*Figure 2.* `neuron-profile` trace for 2-layer GraphSAGE (Torch implementation)

| Metric | Torch | `NKI` | Improvement |
|---|---|---|---|
| `total_time` | 1.897 ms | **1.736 ms** | 1.09× |
| `dma_active` | 98.25% | **98.91%** | +0.67 pp |
| `hbm_read_bytes` | 35.7 MB | **34.6 MB** | 1.03× |
| `hbm_write_bytes` | 70.3 MB | **1.05 MB** | 67× |
| `sw_dyn_dma_packets` | 52,128 | **33,344** | 1.56× |

*Table 2.* Neuron profiler metrics comparing PyTorch baseline and `NKI` kernel for `gather_scatter` on Trainium2.

### 5.2. DFlash Results

We observe a uniform speedup via DFlash speculative decoding, as seen in table 3. Though the speedup is not as large as seen on H100 GPUs, draft acceptance rates are similar, indicating that further optimization of the DFlash layers for Trainium (via tensor parallelism or custom `NKI` kernels for the self attention layer, for example) may yield speedups that approach those on GPUs. We note that this result is for a relatively small model (Qwen3-4B), and reevaluation on a larger model may yield an improvement.

| Benchmark | Baseline tok/s | DFlash tok/s | Speedup | $\tau$ |
|---|---|---|---|---|
| | | *Math* | | |
| GSM8K | 148 | **159** | 1.07x | 5.61 |
| MATH-500 | 148 | **298** | 2.01x | 6.77 |
| AIME 2024 | 150 | **336** | 2.24x | 6.99 |
| AIME 2025 | 147 | **304** | 2.07x | 6.38 |
| | | *Code* | | |
| HumanEval | 155 | **240** | 1.55x | 5.80 |
| MBPP | 154 | **170** | 1.10x | 6.75 |
| LBPP | 140 | **226** | 1.61x | 6.21 |
| SWE-bench | 130 | **264** | 2.03x | 7.35 |
| LiveCodeBench | 140 | **166** | 1.19x | 6.87 |
| | | *Chat* | | |
| MT-Bench | 144 | **298** | 2.07x | 7.97 |
| Alpaca | 143 | **158** | 1.10x | 7.53 |

*Table 3.* DFlash performance benchmarks on Trainium2 (trn2.3xlarge instance, Qwen3-4B). Speedup and average acceptance length ($\tau$) are compared against the autoregressive baseline. Computed using 10 randomly sampled problems from each benchmark.

## 6. Conclusion

Our experiments reveal that the Trainium PyTorch/XLA/`neuronx-cc` workflow leaves performance on the table on unusual workloads such as GNN-based algorithms. However, we also demonstrate the ability to generate performance improvements in a generalizable way, finding that memory optimizations via an optimized hardware-aware `gather-scatter` kernel uniformly improve performance for a variety of algorithms. We

additionally find that, with an efficient software and KV cache-aware implementation, DFlash can accelerate Trainium LLM inference.

### 6.1. Future Work

There are significant additional performance opportunities with respect to both workloads. With the GNN kernel, further optimization of the `gather-scatter` kernel which takes graph structure priors into account could be immensely valuable for performance. Though we test on random graphs, real world graphs can have significantly more structure that may yield useful conditions for such an operation.

The DFlash implementation remains to be tested on a wider variety of models and in the batched inference regime, and ideally should be integrated into a framework like vLLM to allow for continuous batched inference, which would make Trainium2 even more useful for inference in the real world. It also remains to test on larger models, which in theory should provide even larger speedups as the decode MFU gap grows with active parameters.

## References


[1] Anonymous. Eagle-3: Scaling up inference acceleration of large language models via training-time test. *OpenReview*, 2025.

[2] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[3] Jian Chen, Yesheng Liang, and Zhijian Liu. Dflash: Block diffusion for flash speculative decoding. *arXiv preprint arXiv:2602.06036*, 2026.

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[5] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[6] Cohere Labs. Lbpp dataset. https://huggingface.co/datasets/CohereLabs/lbpp, 2024.

[7] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

[8] Naman Jain, Aashish Chouhan, King Han, et al. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

[9] Carlos E Jimenez, John Yang Murphy, Arijit Saha, Karthik Wong, Shunyu Chen, et al. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

[10] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017.

[11] Hunter Lightman, Vineet Kosaraju, Yura Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. *arXiv preprint arXiv:2305.20050*, 2023.

[12] Mathematical Association of America. American invitational mathematics examination (aime). https://maa.org/math-competitions/aime, 2024.

[13] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

[14] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca, 2023.

[15] Qwen Team. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[16] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2019.

[17] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Hao, Zhanghao Wu, Joseph Ba, Eugene Zhuang, Zi Lin, Zhuohan Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *arXiv preprint arXiv:2306.05685*, 2023.