Comparative Benchmark of Scaled-YOLOv4 on Graphcore IPU and NVIDIA GPU

Arjan Siddhpura[1], S.-Kazem Shekofteh[1], Holger Fröning[1]

Hardware and Artificial Intelligence (HAWAII) Lab, Heidelberg University
`arjan.siddhpura@stud.uni-heidelberg.de`, {`kazem.shekofteh`, `holger.froening`}`@ziti.uni-heidelberg.de`

Abstract. The performance of deep learning models is heavily coupled to hardware, with SIMT-based GPUs being the de facto standard. Novel architectures like the Graphcore Intelligent Processing Unit (IPU), with its Multiple Instruction, Multiple Data (MIMD) design and distributed on-chip SRAM, offer a different paradigm. However, direct performance comparisons for complex, state-of-the-art computer vision models are sparse. This paper presents a comprehensive performance benchmark of the Scaled-YOLOv4-P5 object detection model on a Graphcore GC200 IPU against a comparable NVIDIA A30 GPU. We investigate the performance trade-offs by analyzing inference latency and throughput while varying image size, batch size, and floating-point precision. Our findings reveal a stark performance trade-off. The IPU excels in low-latency scenarios, delivering a 6.56 ms inference time at batch=1 (896 px), nearly 4x faster than the GPU's 26.17 ms. Conversely, the GPU's SIMT architecture scales near-linearly for high-throughput, while the IPU is severely memory-constrained. The IPU failed to compile at batch=2 for the native 896px resolution, limited by its $\sim$900 MB on-chip SRAM. In contrast, the GPU's 24 GiB HBM2 memory handled batches of $\geq$64 at the same resolution. Furthermore, the IPU's Ahead-of-Time compilation incurs a major overhead: a full benchmark run at 896 px took 382.79 s on the IPU versus just 15.56 s on the GPU, with 75-88% of the IPU's time spent on compilation alone.

Keywords: Object Detection · Graphcore IPU · YOLOv4 · Performance Benchmarking · Hardware Accelerators

1 Introduction

The performance of deep learning is mutually tied to the evolution of specialized hardware, a dependence famously described as the "Hardware Lottery" [9]. For years, this field has been dominated by Graphics Processing Units (GPUs), whose Single Instruction, Multiple Threads (SIMT) architectures are exceptionally efficient at processing the dense matrix multiplications that form the backbone of modern computer vision models. This SIMT design thrives on massive data parallelism, which is achieved by processing large batches of data simultaneously. However, this is a disadvantage for applications like real-time object detection, which demand the lowest possible latency on a single input, i.e., a batch size of 1. This mismatch has spurred the development of novel AI accelerators built on different paradigms. One such alternative is the Graphcore Intelligent Processing Unit (IPU), a massively parallel processor built on a Multiple Instruction, Multiple Data (MIMD) design with ∼900 MB of distributed on-chip SRAM. This architecture is ideal for workloads with fine-grained, irregular parallelism and is designed to deliver low latency by co-locating memory and compute.

This architectural dichotomy however lacks a direct comparison for state-of-the-art (SOTA) vision models. While vendor-published benchmarks for the IPU claim significant inference speedups - including a 2.2x latency and 4.7x throughput improvement on Resnet-50 [8], a 15x latency and 14x throughput improvement on EfficientNet-B0 [17] and a 24x latency and 7x throughput on ResNeXt-101 [21], these claims [7] require independent validation and a thorough comparison. Other studies compare hardware from different generations (e.g., first-generation IPU MK1 (GC2) vs. NVIDIA V100) [1], different power envelopes, or different target workloads. These studies often focus on tasks such as matrix multiplication [10,14], or vision tasks like text detection [16] and cosmology image generation [1]. What is missing is an empirical benchmark that compares the current-generation, power-comparable data center accelerators, specifically the 150W Graphcore GC200 IPU and the 165W NVIDIA A30 GPU, on a large and complex object detection model like the 70.8-million-parameter Scaled-YOLOv4-P5 [19].

The objective of this work is to fill this specific gap. We present a rigorous, quantitative performance benchmark of the Scaled-YOLOv4-P5 model on a Graphcore GC200 IPU versus a power-comparable NVIDIA A30 GPU. We analyze the architectural trade-offs by measuring end-to-end inference latency and throughput while varying input image size, batch size, and floating-point precision.

Our primary contributions are:

1. Confirming the feasibility along with a benchmark of the ∼70.8M-parameter Scaled-YOLOv4-P5 model on a GC200 IPU and an A30 GPU, which quantifies the IPU's significant low-latency advantage in real-time scenarios: at a batch size of 1 (896px, FP16), the IPU delivers a 6.56 ms inference time, a nearly 4x speedup over the A30 GPU's 26.17 ms.
2. Identification of the IPU's critical performance bottlenecks: the ∼900 MB on-chip SRAM memory wall which causes compilation to fail at a batch size of 2 (896px), and the 24x time-to-result overhead (382.79 s vs. 15.56 s) incurred by its Ahead-of-Time (AOT) compilation model.

This work provides empirical data for system designers, demonstrating the fundamental performance trade-off between the MIMD/SRAM (specialization) and SIMT/HBM (generalization) architectural designs. We show how the IPU's low-latency advantage is a direct result of its specialized MIMD/SRAM design, which successfully maps the model into on-chip memory at batch=1. However, this specialization also creates an inflection point: the ∼900 MB on-chip "memory wall" is now insufficient for SOTA models, causing a hard compilation failure at batch=2. This work thus quantifies the practical trade-off: the IPU's 4x low-latency (batch=1) win is counter-balanced by its inflexibility and a severe 24x time-to-result compilation overhead, a practical cost of its AOT model.

The code for this benchmark is available at [GitHub link].

2 Background and Related Work

2.1 Object Detection Model

The YOLO family transformed object detection by treating the problem as a single regression task, enabling real-time performance. From its multiple iterations, we selected the Scaled-YOLOv4-P5 for our benchmark. While newer models exist, this model provides a representative benchmark for hardware. Its architecture is not just a simple, sequential CNN; it features a complex CSP-Darknet53 [20] backbone and a Path Aggregation Network (PANet) [12] neck. The backbone consists of deep stacks of dense convolutional layers, which aligns well with the high data parallelism of the GPU's SIMT architecture. Conversely, the PANet, with its top-down and bottom-up feature-fusion pathways, creates an irregular, graph-like computation flow favoring the IPU's MIMD flexibility. This model thus serves as a stress test for the competing SIMT and MIMD execution models.

Previous research highlights the performance of YOLOv4 variants on GPU hardware. For instance, the YOLOv4-tiny model achieved 371 FPS at 21.7% AP on an NVIDIA GTX 1080 Ti and 443 FPS on an NVIDIA RTX 2080 Ti [19]. The larger YOLOv4-P5 model, in turn, reached 41-43 FPS on an NVIDIA Tesla V100 GPU [19], similar to our results.

2.2 Hardware Accelerators

We chose the following two accelerators based on their similar power consumption levels and pricing. Table 1 provides an overview of both the hardware platforms used.

The NVIDIA A30 GPU is a data-center accelerator based on the Ampere architecture. Its execution model is Single Instruction, Multiple Threads (SIMT) and comprises 56 Streaming Multiprocessors (SMs), each executing the same instruction on many data points in parallel. This throughput-oriented design's performance relies on latency hiding. It uses a 24 GB pool of off-chip HMB2 memory, which is relatively slow to access (933 GB/s) compared to on-chip memory. Given the SIMT design, workloads characterized by irregular data access, control flow divergence or sparsity result in performance penalties on the GPU. Despite having a large memory capacity, the GPU is also limited by its memory bandwidth when the time spent transferring data outweighs the computational speedup.

The Graphcore GC200 IPU is a many-core MIMD processor which addresses such limitations. It is a latency-oriented design, which contains 1,472 independent "tiles", each a full processor core with 624 KiB of dedicated SRAM. This totals ∼900 MB of on-chip memory, accessible at an aggregated 47.5 TB/s. The Poplar SDK acts as an Ahead-of-Time (AOT) compiler. It analyzes the entire YOLOv4 compute graph, partitions, it, and maps each operation (e.g. a specific convolution) and its required data (weights, activations) to a specific tile before execution, including a static, deterministic allocation for the distributed SRAM to minimize latency from caching, swapping, or prefetching. However, a key limitation arises from the constrained 624 KiB per tile, leading to resource contention among instruction code, model weights and input data; for instance, the largest square matrix multiplication feasible on the GC200 IPU was limited to a side length of 3,584 elements (single-precision), utilizing only 17% of available ∼900 MB of In-Processor Memory [14]. This memory bottleneck extends beyond matrix operations, though subsequent research on butterfly factorization [13,15] has shown that structured sparsity can yield up to 98.5% compression, offering potential strategies to alleviate constraints for large-scale models like YOLOv4.

At runtime, the entire model is resident on-chip when using a single IPU and no streaming memory. As input images arrive, the chip executes in a Bulk Synchronous Parallel (BSP)-style [18]: (1) all 1,472 tiles compute in parallel, (2) they all synchronize, (3) they exchange necessary data with other tiles and then repeat. This architecture enables significant performance advantages over GPUs in certain tasks; for example, the IPU achieved up to 100x speedup in Breadth-First-Search (BFS) execution time over an NVIDIA A100 GPU for large graphs with up to 1 million edges [3].

Benchmarks specific to YOLOv4 on IPU systems further underscore these benefits: the model attained 924 FPS with a latency of 6.49 ms for 896 px images at a batch size of 4 (half-precision) on the Open Graph Benchmark (OGB) [4], while the Graphcore Bow-2000 system delivered 903 FPS compared to 122 FPS on an NVIDIA A100 under similar conditions, representing a throughput increase of more than 7 times [2].

3 Methodology

To provide a fair and direct comparison, we designed an experiment to measure the inference performance of the identical Scaled-YOLOv4-P5 model on both hardware platforms, varying key parameters.

3.1 Model and Dataset

The benchmarked model was Scaled-YOLOv4-P5, adapted from the Graphcore example repository [5]. The model has approximately 70.8×10^6 parameters and requires approximately 167×10^9 multiply-add operations (MACs) per inference [1]. [2] We used the COCO 2017 dataset's [11] 5,000-image validation set to verify

[1] as measured via Graphcore's tool `gc-hosttraffictest` for measuring the host-device bandwidth on the IPU.

[2] Estimates using `torchinfo`.

Chip	Graphcore GC200 IPU	NVIDIA A30 GPU
Number of Parallel Cores	1472	3584
Memory per Parallel Unit [KiB]	624	192
Number of Threads	8832	229 376
Total SRAM [MiB]	918	34.75 (L1 + L2)
Total DRAM [GiB]	256	24
DRAM Bandwidth [GB/s]	20	933
Clock Frequency [GHz]	1.33	1.44
FP32 Peak Compute [TFLOP/s]	62.5	10.3
Power Consumption [W]	150	165
Inter-Chip-Bandwidth [GB/s]	350	200
Host-Device Memory Bandwidth[1] [GB/s]	14.55	31.5
Device-Host Memory Bandwidth[1] [GB/s]	6.33	31.5
Software Stack	PopTorch 3.40.0	PyTorch 2.4.1
CPU Host Processor	2x AMD EPYC 7443 24-Core	2x AMD EPYC 7452 32-Core

Table 1. Specification comparison of the environment setup used for the benchmarking.

accuracy using pre-trained weights [6], achieving mAP$_{50-95}$ results of 49.1 and mAP$_{50}$ of 68.7, consistent with the official benchmarks on both platforms [5].

The GPU port was gradually adapted to ensure maximum overlap to the original implementation on the IPU. This included replacing PopTorch dependency with PyTorch for the inference, explicit device management to shift all relevant tensors to the GPU, using `cuDNN` to find the most optimal convolution algorithms applicable for the model and elimination of redundant data movement during post-processing steps such as non-maximum suppression (NMS).

3.2 Benchmarking Methodology

Two performance metrics were measured during the inference, including on-device pre-processing and post-processing:

1. Latency (ms): We measured full round-trip (end-to-end) latency. To ensure precision, we used device-specific timing primitives: `poptorch.PoplarExecutor.getLatency()` for the IPU and `torch.cuda.Event` for the GPU. This captures all host-to-device transfers, model execution, and device-to-host transfers. For the GPU, we used `torch.cuda.synchronize()` to account for CUDA's asynchronous nature.
2. Throughput (FPS): We measured the total number of images processed per second. The total wall-clock time was measured using standard Python primitives (`time.time()`) on the host and divided by the total number of images, calculated after performing 100 warm-up iterations.

We varied three experimental parameters:

1. Input Image Size: We fixed the batch size to 1 and varied the input image sizes from 64 to 1280 pixels. The baseline batch size of 1 ensures the maximum possible image size can be tested on both hardware, along with providing a benchmark for real-life scenarios where batching is not feasible.
2. Batch Size: We fixed the image sizes and varied the batch sizes in powers of two from 1 to 64. Image sizes of 128, 224 and 320 pixels are used to provide a comprehensive overview.
3. Floating-Point Precision: We repeated all the experiments using both half-precision (FP16) and single-precision (FP32) floating point parameters.

Lastly, to quantify the practical implications of each platform's development cycle, we measured the total time-to-result, defined as the sum of compilation and execution times.

All tests are performed using a single IPU and GPU to ensure a fair comparison. The results are reported as the average of 100 benchmark iterations.

4 Results

Once the accuracy of the model was verified on both platforms, we varied the image sizes and batch sizes. Preliminary tests revealed a failure on the IPU when compiling the model using a batch size of 2 using the model's default image size of 896 px due to the limited on-tile memory capacity of 624 KiB being exhausted. This memory limitation on the IPU constraints what image and batch sizes could be tested on both the hardware. Thus, the performance envelope of the IPU had to be determined before further tests could be performed, which is presented in Figure 1.

When varying the input image sizes, we find the IPU being 3.99x faster than the GPU on the model's default input image size of 896 px, with a round-trip latency of 6.56 ms compared to the GPU's 26.17 ms when using a batch size of 1 and half precision. At batch size of 1, the IPU is faster across all image sizes. The GPU maintains an almost constant latency until images of size 800 px, after which it trends upwards. At half precision, we observe a speedup ranging from 24.7x (image size = 128 px) till 3.60x (image size = 1088 px) on the IPU. The doubling of memory requirements when using a higher floating point precision level (FP32) results in only half the image sizes compiling on the IPU along with a higher rate of increase in latency when increasing the input image size.

When varying the batch size on the IPU, smaller image sizes of 128 px, 224 px and 320 px are used due to its limited memory. The GPU maintains a consistent latency (at $\sim$25 ms) from batch sizes of 1 to 8, confirming it is overhead-bound. The IPU's latency scales linearly with batch size (e.g. from 1.03 ms at batch 1 to 8.51 ms at batch 64 for 128 px images), confirming it is compute-bound. As shown in Figure 4, the GPU's throughput scales almost linearly with batch size, whereas the IPU's scaling is shown to be weaker. Figure 3 shows the IPU always maintaining a lower latency than the GPU.

Doubling the precision doubles the memory footprint for weights and activations. This leads to a cut in half for the IPU's maximum achievable batch size for most tests, as shown in Figure 2, 3 and 4 along with a greater increase in latency on the IPU compared to the GPU, for which the latency and throughput remains the same despite using FP32 until the batch size of 8.

Lastly, Figure 5 shows the Poplar compiler taking 382.79 seconds (over 6 minutes) to run the benchmark for the default image size of 896 px, compared to the GPU's 15.56 seconds.

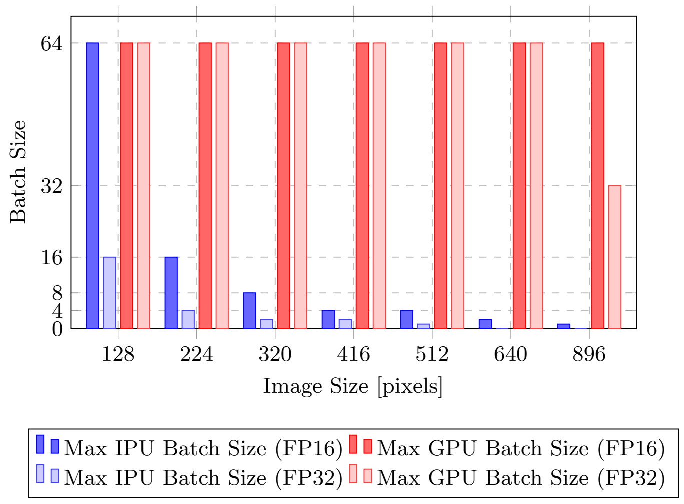

Fig. 1. Maximum achievable batch size for the Scaled-YOLOv4-P5 model on a set of pre-defined image sizes, contrasting the memory limitations of the Graphcore GC200 IPU to the Nvidia A30 GPU.

5 Discussion

As Figure 1 illustrates, the NVIDIA A30, with its 24 GB of HBM2, easily handled all tested batch sizes. The Graphcore GC200, however, is severely constrained by its ~900 MB of on-chip SRAM. At the model's native resolution, it can only compile for a batch size of 1. At 1120 px, it fails to compile entirely.

The performance difference when varying the input image size stems directly from the architectural trade-offs. The IPU's 6.56 ms is the true compute time, as the model and graph are resident in fast SRAM. Thus, the IPU's latency, governed purely by computation, scales quadratically with the range of image sizes throughout, conforming to theoretical relationship of quadratic scaling for increasing input image sizes on the YOLOv4 model [19]. The GPU's 26.17 ms is an overhead-bound time. Its powerful SMs are starved for data at a batch size of 1, and the total time is dominated by the fixed costs of CUDA kernel launches and the high latency of fetching data from off-chip HBM2. For real-time applications, the IPU's MIMD/SRAM architecture is demonstrably superior.

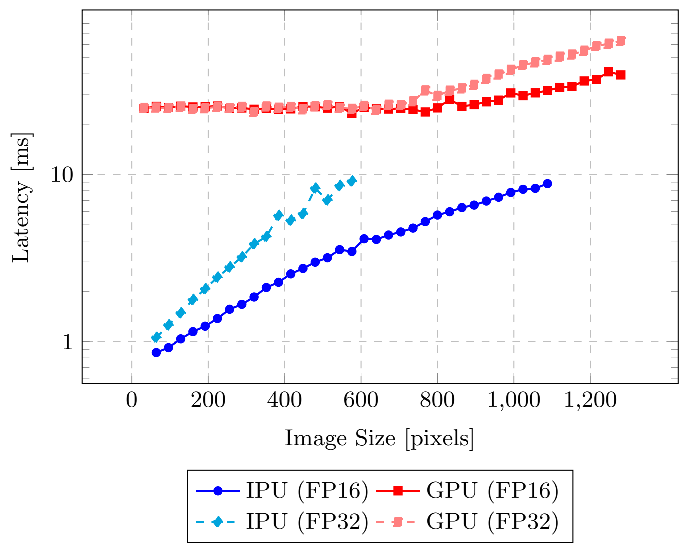

Fig. 2. Comparison of the effect of changing floating-point precision from FP16 to FP32 on the average single-batch inference latency as a function of the input image size. The throughput in this case is the inverse of latency given the fixed batch size of 1. The plot uses a log-linear scale.

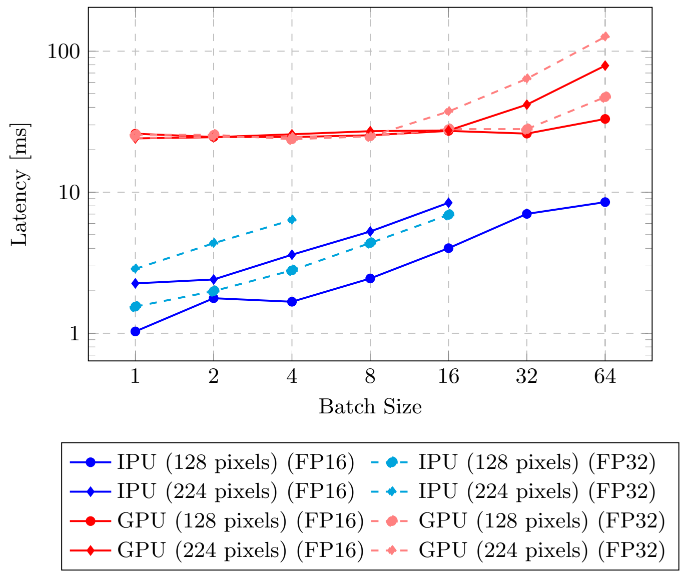

Fig. 3. Average inference latency as a function of batch size for both FP16 and FP32. This is shown using two image sizes to contrast not just the hardware, but also the effects of changing precision and image size on the same hardware. The plot uses a $\log_{10}$-$\log_2$ scale.

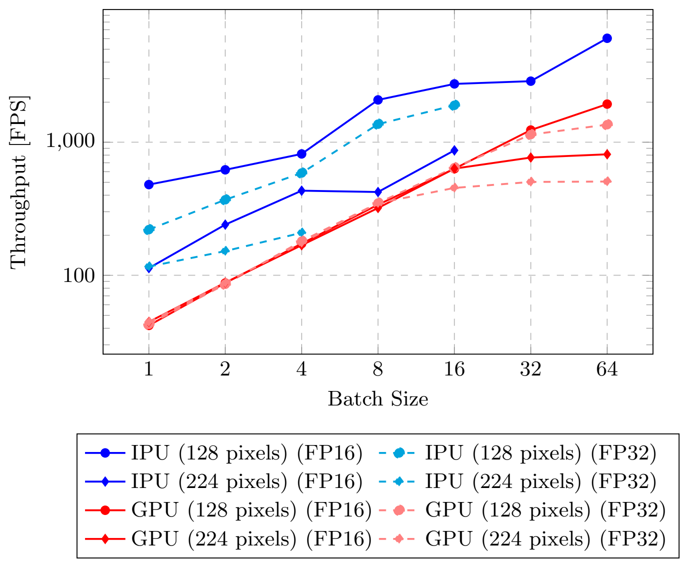

Fig. 4. Throughput vs. batch size across FP16 and FP32 precision for the image sizes of 128 and 224 pixels. The plot uses a $\log_{10}$-$\log_2$ scale.

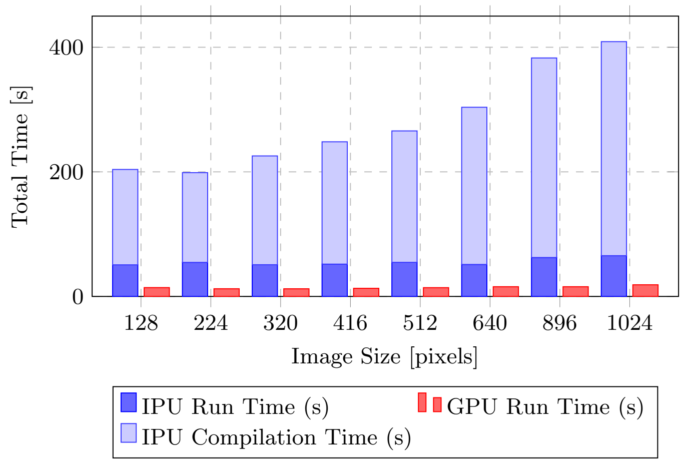

Fig. 5. Comparison of the total time-to-result on the IPU and the GPU. The IPU requires significant time to compile along with run-time overheads.

Figure 3 and 4 show how the GPU's SIMT architecture, designed for massive data parallelism, becomes fully saturated only at larger batch sizes, whereas the IPU's MIMD architecture and limited memory prevent it from scaling to the same batch sizes. In all the presented experiments, the IPU remains consistently bound more by compute than memory, unlike the GPU, which is bound by compute only after large enough input data is provided. This shows the GPU's advantage for applications where data can be batched and throughput is the primary metric to optimize.

Lastly, Figure 5 illustrates how the NVIDIA A30 GPU's eager execution model and a Just-In-Time (JIT) execution approach is 24.6x faster compared to the IPU on the default image size. Pre-optimized kernels are launched during execution, incurring minimal setup overhead. The Graphcore IPU, however, requires a complete, static computational graph before execution. The Poplar compiler performs an Ahead-of-Time (AOT) compilation to perform global optimizations and map the entire graph onto the IPU's 1,472 tiles, which incurs a substantial initial cost, although the resulting executable is cached for subsequent runs which share the same parameters.

6 Conclusion

After verifying the feasibility of running an object detection model on the IPU, we show how the IPU excels at minimizing latency when using smaller image and batch sizes in contrast to the GPU, which excels at maximizing throughput with bigger image and batch sizes. The primary performance differentiator is in single-batch inference, where the IPU exhibits a significant advantage for real-time applications; at input image sizes of 896 px using FP16 precision, the IPU delivers a latency of 6.56 ms, an almost 4x speedup over the GPU's 26.17 ms. This low-latency advantage presents a trade-off, as the GPU's larger memory capacity of 24 GB supports higher batch sizes (e.g., 32 vs. 8 at 320 px), enabling the GPU to achieve a greater peak throughput (426.17 FPS) compared to the IPU's memory-limited maximum (365.54 FPS) with $\sim$900MB of on-chip memory. Finally, due to the IPU's compilation cost, the total time to benchmark a new run was more than an order of magnitude greater than the GPU's. Nonetheless, for all independent metrics we benchmark on the IPU where it was not constrained by its memory, it outperformed the GPU.

These findings suggest several avenues for future research. A comprehensive roofline analysis, for instance, could quantify performance bottlenecks, while energy efficiency could be assessed using metrics like FLOPS/Watt. Furthermore, this evaluation could be extended to multi-IPU configurations and models with diverse computational and memory demands. The GC200 architecture also supports Streaming Memory, which was not utilized in this study. Future work should benchmark performance when streaming weights and activations from this larger memory pool to see if it provides a viable path for larger models or batches on the IPU. The results underscore the importance of aligning the architectural strengths of an accelerator with the specific requirements of the target application to maximize performance.

References

1. Arcelin, B.: Comparison of graphcore ipus and nvidia gpusfor cosmology applications (2021), https://arxiv.org/abs/2106.02465
2. Bohl, R.: Graphcore ipus: Accelerating argonne's ml/ai applications. https://extremecomputingtraining.anl.gov/wp-content/uploads/sites/96/2022/11/ATPESC-2022-Track-1-Talk-7-Bohl-Graphcore.pdf (Aug 2022), presented at ATPESC 2022, Track 1 (Hardware Architectures); accessed 2025-11-01
3. Gepner, P., Kocot, B., Paprzycki, M., Ganzha, M., Moroz, L., Olas, T.: Performance evaluation of parallel graphs algorithms utilizing graphcore ipu. Electronics **13**(11) (2024). https://doi.org/10.3390/electronics13112011, https://www.mdpi.com/2079-9292/13/11/2011
4. Graphcore: Performance results. https://www.graphcore.ai/performance-results (2023), web page; last updated July 4, 2023; accessed 2025-11-01
5. Graphcore: examples: vision/yolo_v4/pytorch (yolov4 pytorch example for graphcore ipus). https://github.com/graphcore/examples/tree/master/vision/yolo_v4/pytorch (2025), gitHub repository directory; accessed 2025-11-01
6. Graphcore: Yolov4 p5 reference weights. https://gc-demo-resources.s3.us-west-1.amazonaws.com/yolov4_p5_reference_weights.tar.gz (2025), archive of YOLOv4 P5 model weights; accessed 2025-11-01
7. Graphcore Limited: Benchmarks. Tech. rep., Graphcore Limited (Oct 2020), https://www.graphcore.ai/hubfs/assets/pdf/GC_Benchmarks_Oct_20.pdf?hsLang=en, pDF available at https://www.graphcore.ai/hubfs/assets/pdf/GC_Benchmarks_Oct_20.pdf?hsLang=en
8. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition (2015), https://arxiv.org/abs/1512.03385
9. Hooker, S.: The hardware lottery (2020), https://arxiv.org/abs/2009.06489
10. Jia, Z., Tillman, B., Maggioni, M., Scarpazza, D.P.: Dissecting the graphcore ipu architecture via microbenchmarking (2019), https://arxiv.org/abs/1912.03413
11. Lin, T.Y., Maire, M., Belongie, S., Bourdev, L., Girshick, R., Hays, J., Perona, P., Ramanan, D., Zitnick, C.L., Dollár, P.: Microsoft coco: Common objects in context (2015), https://arxiv.org/abs/1405.0312
12. Liu, S., Qi, L., Qin, H., Shi, J., Jia, J.: Path aggregation network for instance segmentation (2018), https://arxiv.org/abs/1803.01534
13. Shekofteh, S.K., Alles, C., Fröning, H.: Reducing memory requirements for the ipu using butterfly factorizations. In: Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis. p. 1255–1263. SC-W '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3624062.3624196, https://doi.org/10.1145/3624062.3624196
14. Shekofteh, S.K., Alles, C., Kochendörfer, N., Fröning, H.: On performance analysis of graphcore ipus: Analyzing squared and skewed matrix multiplication (2023), https://arxiv.org/abs/2310.00256
15. Shekofteh, S.K., Bogacz, D., Alles, C., Fröning, H.: Butterfly factorization for vision transformers on multi-ipu systems. Parallel Computing **VX**, XXX (2025)

16. Sumeet, N., Rawat, K., Nambiar, M.: Performance evaluation of graphcore ipu-m2000 accelerator for text detection application. In: Companion of the 2022 ACM/SPEC International Conference on Performance Engineering. p. 145–152. ICPE '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3491204.3527469, https://doi.org/10.1145/3491204.3527469
17. Tan, M., Le, Q.V.: Efficientnet: Rethinking model scaling for convolutional neural networks (2020), https://arxiv.org/abs/1905.11946
18. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33**, 103–111 (1990), https://api.semanticscholar.org/CorpusID:15655597
19. Wang, C.Y., Bochkovskiy, A., Liao, H.Y.M.: Scaled-yolov4: Scaling cross stage partial network (2021), https://arxiv.org/abs/2011.08036
20. Wang, C.Y., Liao, H.Y.M., Yeh, I.H., Wu, Y.H., Chen, P.Y., Hsieh, J.W.: Cspnet: A new backbone that can enhance learning capability of cnn (2019), https://arxiv.org/abs/1911.11929
21. Xie, S., Girshick, R., Dollár, P., Tu, Z., He, K.: Aggregated residual transformations for deep neural networks (2017), https://arxiv.org/abs/1611.05431