# iPUG: Accelerating Breadth-First Graph Traversals using Manycore Graphcore IPUs

Luk Burchard[1,3][0000−0002−8019−7047], Johannes Moe[1,2][0000−0001−7664−9517], Daniel Thilo Schroeder[3,4][0000−0003−0125−5243], Konstantin Pogorelov[1][0000−0002−7993−1769], and Johannes Langguth[1,5][0000−0003−4200−511X]

[1] Simula Research Laboratory, Norway {konstantin,langguth}@simula.no
[2] University of Oslo, Norway johanom@ifi.uio.no
[3] Technical University Berlin, Germany l.burchard@campus.tu-berlin.de
[4] Simula Metropolitan Center for Digital Engineering, Norway daniels@simula.no
[5] BI Norwegian Business School

**Abstract.** The Graphcore Intelligence Processing Unit (IPU) is a newly developed processor type whose architecture does not rely on the traditional caching hierarchies. Developed to meet the need for more and more data-centric applications, such as machine learning, IPUs combine a dedicated portion of SRAM with each of its numerous cores, resulting in high memory bandwidth at the price of capacity. The proximity of processor cores and memory makes the IPU a promising field of experimentation for graph algorithms since it is the unpredictable, irregular memory accesses that lead to performance losses in traditional processors with pre-caching.
This paper aims to test the IPU's suitability for algorithms with hard-to-predict memory accesses by implementing a breadth-first search (BFS) that complies with the Graph500 specifications. Precisely because of its apparent simplicity, BFS is an established benchmark that is not only subroutine for a variety of more complex graph algorithms, but also allows comparability across a wide range of architectures.
We benchmark our IPU code on a wide range of instances and compare its performance to state-of-the-art CPU and GPU codes. The results indicate that the IPU delivers speedups of up to 4× over the fastest competing result on an NVIDIA V100 GPU, with typical speedups of about 1.5× on most test instances.

**Keywords:** IPU · Graph500 · BFS · performance optimization

## 1 Introduction

In their Turing lecture 2018, John Hennessy and David Patterson announced a *"new golden age for computer architecture"* [18]. They based the statement on the fact that due to the slower performance gains made by general purpose processors today, domain-specific architectures becomes more and more viable. Indeed, a large number of startups that develop specialized processors, usually for AI applications, have appeared in the recent years.

One of these companies is Graphcore, who presented their first processor, called the Colossus GC2, in 2018. It is targeted at machine intelligence applications and referred to as an intelligence processing unit (IPU). Similar to GPUs, the IPU offers a high number of low precision FLOPS that come from a large number of compute cores. However, unlike the GPU, which focuses on single instruction multiple data (SIMD) processing, the IPU offers true multiple instruction multiple data (MIMD). Furthermore, instead of DRAM with a cache hierarchy, it uses SRAM as its main memory. In theory, this design makes the IPU uniquely suited for highly irregular workloads such as graph algorithms. The goal of this paper is to test whether these architectural advantages result in measurable performance benefits.

To this end, we implement an IPU-based breadth-first search (BFS), following the specifications of the Graph500 [28] benchmark. Introduced in 2010, Graph500 collects BFS performance results for a wide range of hardware platforms and instance sizes making it by far the most studied parallel graph problem, which gives us a wide range of meaningful comparison points. The Graph500 uses a Kronecker graphs generator similar to R-MAT[11]. Results are denoted in traversed edges per second (TEPS). In addition, we use a test set of Yang et al. [32], which consists of matrices from the SuiteSparse [22] matrix collection.

We consider our work primarily as a building block for multi-IPU BFS and other, more sophisticated graph algorithms that use BFS repeatedly. These include graph centralities and other algorithms used in the analysis of social networks, graph matchings, and similar algorithms.

While the IPU's large number of independent compute cores, fast interconnect between these cores, and fast SRAM memory make it a very attractive platform for graph algorithms, we face two major challenges when using the IPU in this manner. First, the device was designed for machine intelligence applications, and the provided data structures and architecture design reflect that. While the cores are MIMD capable, there is no special support for irregular data structures such as graphs. Furthermore, all communication between the IPU cores must be declared at compile time. Naturally, this is a major challenge for computations such as BFS or other graph algorithms that determine communication patterns based on decisions done at run-time. Second, since the main IPU memory is SRAM, it is very limited, which puts a strict limit on the size of the graphs that can be processed by a single IPU.

To tackle the former limitation, the code creates its own mapping of the graph to the compute cores. We also control memory alignment explicitly, as well as the spawning of worker threads on the compute cores. Via temporal multithreading, the memory access latency can be hidden such that the individual threads do not experience latency. Naturally, the latter cannot be overcome via software. Thus, our paper makes the following contributions:

1. We present the first implementation of a graph algorithm on the new Graphcore IPU architecture whose features promise outstanding performance for such problems.

2. We give a detailed discussion of the challenges that need to be overcome to run efficient graph algorithms on the IPU. We expect that these techniques are applicable to a wide range of other graph algorithms as well.
3. We present performance comparison experiments using state-of-the-art CPU and GPU codes and hardware. The results show that our IPU implementation (which we refer to as iPUG) compares favorably to all tested alternatives.

The remainder of the paper is organized as follows: we introduce the IPU in Section 2 and discuss related BFS work on other architectures in Section 3. We present our IPU implementation in Section 4 and our experiments in Sections 5 and 6. In Sections 7 and 8 we discuss the implications of these results and conclude the paper.

## 2 IPU Hardware

### 2.1 Architecture

The Colossus GC2 IPU has 1216 *tiles*, each tile having a compute core and its own local memory of 256 KiB. Thus, the IPU has a total of 304 MiB of memory. The tile layout is illustrated in Figure 1. The memory of the tiles is implemented in SRAM and is thus part of the chip. Naturally, this offers a far higher bandwidth (45 TB/s, aggregate) and lower latency (6 clock cycles) than DRAM. The tiles themselves are organized into *islands* consisting of four tiles, and the islands are grouped into *columns* of 19 islands each. The GC2 IPU has 16 such columns. The cores run at a default frequency of 1.6 GHz, but they can be clocked down to 1.3 GHz for thermal or electrical reasons, such as the PCIe slot not being able to provide the required power. A single GC2 IPU has 150W TDP. The PCIe version hosts two IPUs per card for a total of 300W TDP, which requires additional cables, similar to powerful GPUs. A rack-mounted IPU-POD with four socketed GC2 IPUs is also available. In this version all IPUs run at 1.6 GHz. For an in-depth discussion of the architectural details including microbenchmarks, we refer the reader to Jia et al. [20]. In 2020, Graphcore presented an GC200 IPU with more tiles and more memory per tile, but the device was not available for development at the time of this writing.

### 2.2 Programming Model

IPU programming follows the classical dataflow model. Programs are assembled by composing a logical execution graph at compile time. It consists of alternating layers of state and computation vertices. The state is exclusively organized in multidimensional arrays called tensors, which are symbolically represented at compile time and have pre-determined dimensions. Such a structure makes it ideally suited for Tensorflow [1] applications.

Each computation vertex is associated with a *codelet*, i.e. a piece of code that prescribes the computation to occur in the vertex. Multiple codelets at the same

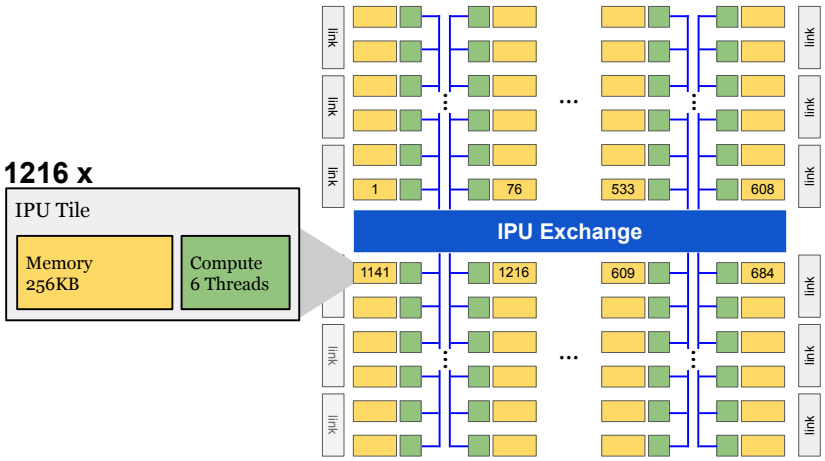


Fig. 1: Tile layout on the IPU processor.

layer of the graph can be executed in parallel as long as they do not write to the same part of a tensor, and all codelets must be executed before progressing to the next layer of the computation graph. At the end of such a compute step, data is exchanged among the cores to ensure a consistent state, thereby creating a bulk-synchronous parallel (BSP) [30] superstep structure. The rationale for this structure is that due to bandwidth contention, overlapping memory-bound computation and communication is difficult and sometimes impossible [25]. Furthermore, it provides a clear computation structure and obviates the need for message buffers and thus additional memory on the chip, making communication very efficient. On the other hand, this brings about that all communication must be planned at compile time. This poses a challenge when communicating sparse data, which is necessary in graph algorithms such as BFS.

## 3 Background

### 3.1 Related Work

BFS and DFS are the most fundamental ways of traversing graphs. For sequential execution, the BFS algorithm is essentially defined by the data structure used to store the graph, as its fundamental operation is to iterate over the edges of a given vertex. However, parallel implementation of BFS, particularly on distributed memory systems, is far more complicated. Consequently, there are far more possibilities for algorithm design and performance optimization.

While parallel BFS has been studied earlier [15], the topic gained widespread interest in the previous decade on distributed memory computers [16, 35], on shared memory [5, 23], and on GPU systems [17]. The establishment of the

Graph500 benchmark [28] in 2010 marks a turning point, since it encouraged direct comparability of results. This increased activity on the topic further, resulting in a large number of publications on that topic [10, 12, 13, 19, 33]. Furthermore, BFS implementations for GPUs have also received considerable attention in the recent years [14, 27, 31, 32]. In addition to the parallel implementation, algorithmic improvements have been presented in the last decade. Possibly the most important among those was the introduction of direction optimizing searches [7]. At the same time, efficient parallel algorithms for BFS and DFS were also developed in the context of other graph problems, such as parallel matching algorithms [4, 24, 26].

### 3.2 Graph Algorithms in the Language of Linear Algebra

Among the approaches developed for parallel graph processing, we focus on the linear algebra based formulation [9] of BFS. This is a natural fit since the IPU is designed for machine learning applications, and is thus geared towards linear algebra.

A graph $G = (V, E)$, $|V| = n$, $|E| = m$ can be represented as an adjacency matrix $A \in R^{n \times n}$ with $a_{ij} = 1$ if $(i, j) \in E$ and 0 if $(i, j) \notin E$. Each row in the adjacency matrix encodes the outgoing edges of a vertex. In practice the input graphs are always sparse. We can use the sparsity, and only store the non-zero values of the matrix in a compressed format. For our implementation, we choose the CSC format where the number of values non-zero and their positions are stored for each column. This encoding allows for fast iteration through the column but prohibits quick $A_{ij}$ lookups as we may need to scan through a whole column.

We can formulate a BFS search step by performing a multiplication of an adjacency matrix $A$ with a vector $x$. We initialize the frontier vector $x$ with the index of the source node $s$ with $x(s) = 1$. We can perform a step $A^T x_1 = x_2$ which yields the next frontier. Further we can union all previous frontiers into an array to mark the already visited nodes $v_k = x_1 + \ldots + x_k$, where $v_k(i) \neq 0$ if node $i$ was visited during step $k$. We can choose $A$ and a further $x$ to be represented by an efficient sparse data structure.

The advantage of this representation is that it allows the use of highly optimized sparse linear algebra primitives to accelerate graph algorithms. It provides a high level view for understanding and comparing communication patterns. It is important to note that most applications in scientific computing and machine learning exhibit sparse matrix dense vector (SpMV) communication, which means that the same communication pattern repeats over multiple rounds. On the other hand, graph algorithms such as BFS exhibit sparse matrix sparse vector (SpMSpV) communication where only some of the vertices or matrix rows/columns are active in each round, thus creating a new communication pattern each time.

## 4 BFS implementation on IPU

The distributed memory model of the Graphcore IPU forces us to partition our input problem beforehand; to do so, we divide the input graph and assign one part to each tile. During the following BFS steps, new tile memory needs to be allocated in order to store the previous step's output. Thus, the decomposition of the graphs for the IPU is similar to BFS implementations for distributed memory systems rather than GPUs. The graph decomposition remains static during the algorithm and no additional data is loaded during the entire BFS kernel.

### 4.1 Parallel BFS

Splitting a subset of vertices with their outgoing edges is called 1D partitioning because of the row-wise split in the adjacency matrix. Since input, output, and vertex data must be stored in the tile memory, load-balancing becomes challenging, especially in the case of graphs with vertices of high degree. Furthermore, 1D partitioning requires allocating $\mathcal{O}(n)$ bytes for input and output on each tile, making it an inappropriate partitioning strategy, even for small graphs.

In contrast to 1D, the 2D decomposition splits the adjacency matrix into a chessboard-like $p_x \times p_y$ pattern. Thus, an adjacency matrix $A$ is decomposed into $p$ square partitions $A_{1,1} \dots A_{x,y}$. Each such partition is mapped to an individual physical tile on the IPU. In this scenario, each partition is only responsible for a subset of the outgoing edges of each vertex. Therefore, no single partition has the global state of their vertices and thus the partitions that own a vertex need to communicate their partial results to arrive at a single global state. Our 2D data decomposition is very similar to that used for distributed memory systems [8, 34], and we also permute the vertices randomly. Unlike the 1D partitioning, in 2D we need to allocate only $\mathcal{O}(n/\sqrt{p})$ bytes for communication with other tiles.

### 4.2 Parallel Top-Down

Algorithm 1 shows the parallel top-down, 2D, bulk synchronous parallel (BSP) algorithm. As writing IPU does not require explicit declaration of communication between the tiles, we describe it as a mapping of input and output tensor data regions. In the current implementation all partitions are square, and thus the notation $v_c$ represents an $n/p_x$ sized vector with starting offset $p_x * (1 - c)$. A processor $P_{i,j}$ receives inputs from the frontier queue $Q_j$ and produces the new partial outputs represented by a bitmap matrix $SA_{i,j}$ working on the partition $A_{i,j}$. $SA$ is called the intermediate status array. In order to process one BFS

**Algorithm 1:** Topdown BFS algorithms, adopted from [8] and the linear algebraic version [10]

**input :** A 2D partitioned adjacency sparse matrix $A$, a source vertex $s$, vertex count $n$, partition count $p$
**output:** A vector $b$ containing the parent for each explored $i$ as $b(i)$.

$p_x = p_y \leftarrow \sqrt{p}$
$Q \leftarrow \{s\}, SA(:,:) \leftarrow 0, b(:) \leftarrow 0$
**for** *all processors* $P_{i,j}$ *in parallel* **do**
  **while** $Q \neq \emptyset$ **do**
    $frontier \leftarrow Q_i$ ▷ `Done through mapping and exchange`
    **for** $vertex \in frontier$ **do**
      **for** $neighbour \in adj(A_{i,j}, vertex)$ **do**
        $SA(i : neighbour) \leftarrow true$
      **end**
    **end**
    Global BSP Barrier ▷ `End ComputeSet`
    $Q \leftarrow \emptyset$
    $activations \leftarrow SA(i * p_y + j : i * p_y + j + 1, :)$ ▷ `Like AllGather`
    **for** $v \in b$ **do**
      **if** $v \neq visited$ **then**
        **for** $incoming \in activations(row, :)$ **do**
          **if** $any(incoming)$ **then**
            $b(row) \leftarrow visited$
            $Q \leftarrow Q \cup \{row\}$
          **end**
        **end**
      **end**
    **end**
    Global BSP Barrier ▷ `End ComputeSet`
  **end**
**end**

level, our algorithm requires two separate communication steps, each of which requires a synchronization barrier before proceeding to the next step.

1. Local Expansion: Each processor $P_{i,j}$ receives a $Q_j$ part of the frontier queue and uses it to create a new intermediate status array $SA_{i,j}$
2. Intermediate Status Array Reduction: A reduction that uses the parent array (i.e., Algorithm 1) to check all partial results of a vertex to determine if a new parent was found. This step uses all partitions along the row $j$ of size $n/p$ to reduce into the new frontier queue $Q_j$.

All communication during the local expansion happens column-wise, where the input frontier $Q$ is sent to all rows in their respective parts, as shown in Figure 2. During the reduction phase all communication happens row-wise as all data comes from the partial results of the row to be reduced. In general,

the communication before the local discovery is simpler, since we have a one-to-many communication in contrast to the reduction phase where a many-to-many communication pattern is required.

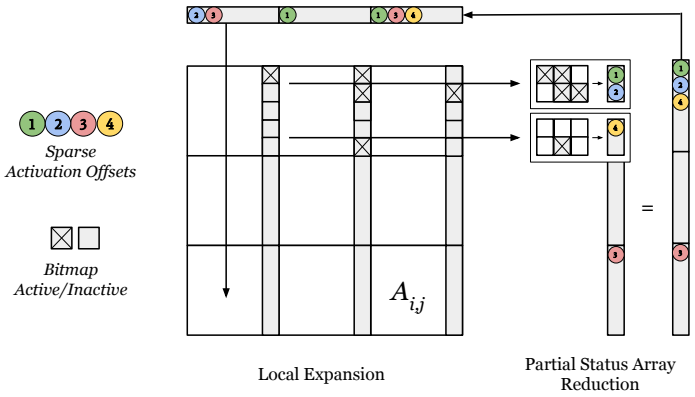


Fig. 2: Layout of the 2D decomposition. We map each partition to a physical processor tile. Each tile also receives a copy of the sparse input frontier, along the first dimension, indicated with colored balls as activated vertices normalized to local offsets on each tile. In the reduction phase processor tiles receive the output status array of the local expansion and merge these into a new sparse frontier vector of the next BFS level.

### 4.3 Mapping Data and Compute

Mapping and allocating data is an important part of the implementation as the compiler does not automate or abstract data and operation placement away from the developer. Thus it is necessary to specify a complete mapping of each tensor partition to each target tile. The same applies when placing vertices of the compute graph on the IPU: each vertex is assigned to a tile. If the necessary data is already present on a tile, then no additional overhead is introduced. However, due to the fast communication between the tiles, this overhead is relatively small when mapping data and compute on a single IPU. Moreover, any unnecessary communication leads to additional allocations of landing zones for data that is transferred between the tiles. This is crucial due to the limited memory on the IPU, which means that suboptimal allocations can cause a computation to fail due to lack of memory.

### 4.4 Challenges of IPU Graph Implementations

**Memory Alignment** Traditionally memory alignment is done by the compiler via padding. Such padding can align the values on cache line boundaries, which ensures that they can be accessed or written efficiently. However, when working with *Poplar* compute graphs, aligning data is not trivial and needs to be done explicitly through the size and splits of a data section. Without manual data alignment, the *popc* IPU compiler allocates rearrangement buffers on the tiles, which costs additional memory. When working with large tensors, the rearrangement buffers tend to grow quickly, thus rendering feasible graph instances infeasible.

**Memory Management** Each compute-and-data section of the compute graph is statically mapped to a tile during compile time. It is not possible to change the location of a data regions to a different tile during runtime and the compute graph does not allow for recursion. Thus, memory space and offsets needed to receive, transfer, and compute vertices can be determined during compilation. Therefore, allocating more memory than available on a single tile leads to an *out of memory* error during compile time. With 256KB of addressable space the per-tile memory is very small compared to traditional memory systems, making memory management a primary concern.

Like traditional compilers, Graphcore's *popc* compiler has dead code elimination [3]. Hence, we call tensors that will not be eliminated *Always Live* variables. These variables need to be allocated during the whole lifetime of the program. Variables that are not *Always live* may get optimized away at some point in the program. For our program, the lifetime of variables connected to the expansion phase is related to the reduction phase and vice versa. Table 1 gives an overview over the variables allocated by our algorithm. The factor of two for the input data is due to the fact that we also need to store the input of the previous round.

| Use | Type | Size | Always Live |
|---|---|---|---|
| Expansion Input | `int16` | $2n/p_x$ | False |
| Expansion Output | `int32` | $n/p_y$ | False |
| Matrix | `int16` | $(n/p+1)+nz_{max}$ | True |
| Backpointers | `int32` | $n/p$ | True |
| Reduction Input | `int32` | $2n/p_x$ | False |
| Reduction Output | `int16` | $n/p_y$ | False |

Table 1: Per tile memory allocated by the BFS algorithm. $nz_{max}$ represents the largest number of nonzeroes among all partitions. If a variable is always live it can not be optimized away by the compiler and is always present in an allocation.

### 4.5 Optimizations

**Removal of Isolated Vertices** The Kronecker graph generator used to generate the graphs for the Graph500 benchmark produces isolated vertices. The greater the generated graph's scale, the larger the ratio of isolated vertices in the generated graph. For our input sizes, we observe 26% isolated vertices at scale 15, which increase to 36% at scale 19. Other papers report a ratio of up to 74% [29] for scale 42 graphs.

For BFS, as well as many other graph algorithms, isolate vertices are completely irrelevant. By filtering these vertices while reading the graph we can reduce the dimension of the generated matrix by 1.6× in linear time, accessing every vertex exactly once. This makes it almost possible to run a scale 20 Kronecker graph on the IPU and further reduces the space needed to store the CSC matrix. By reducing the dimension of the matrix the status array and frontier are also reduced by an additional factor of 2×, thus saving communication and computation time.

**First Reduction Optimization** Our algorithm is required to iterate over all partitions in a row to find an activation if the parent for this row has not been found at the current level. The number of these iterations gets smaller the more vertices have already been flagged as found. Thus, when processing the first BFS level, this number is highest. For a single GC2 IPU we are required to check 34 partition outputs. However, in the first pass, we know that no vertices have been flagged as visited yet and that all possible activations can only come from partitions that get the frontier input section containing the single source vertex. Therefore, we can replace the first reduction with an algorithm saving $\mathcal{O}((p_x - 1)/p_x)$ time which is equivalent to skipping 97% of the instructions at the first level. Thus, instead of first checking the visited array and iterating over all incoming partitions we directly iterate over the incoming intermediate frontier from the partition responsible for handling the source vertex. If an activation was found we can simply insert it without the possibility of overwriting any information as we are in the first reduction phase.

**Utilizing Threads** Similar to GPUs, the IPU allows scheduling multiple threads per core on a tile to hide latencies and fill the processor's pipeline more efficiently. Unlike modern CPUs, which use *simultaneous multithreading*, the IPU architecture leverages a barrel processor design with *temporal multithreading* of up to six hardware threads. A feature of barrel processors is that each execution context has a constant instruction scheduling time as it alternates between active threads in a round-robin fashion. When six threads are executed in this manner, the memory access latency of six cycles can be hidden effectively. The *Poplar* SDK allows us to spawn a compute vertex into a supervisor mode, which is a restricted administrative context thought to be the entry point for starting and orchestrating the six worker contexts. The supervisor can further synchronize context flows into a single sequential point.

Our algorithm utilizes a sparse frontier vector generated in the reduction phase. We cannot write an interleaved value into the frontier immediately after finding it during the reduction, as no atomic instructions are available. To synchronize an unknown amount of value insertions we leverage a prefix sum often found in parallel algorithms on GPUs. Instead of computing and immediately inserting vertices into the output frontier queue, we split the algorithm into three parts: parallel flagging of frontier vertices in a temporary bitmap vector, synchronized prefix-sum calculation for the worker contexts, and parallel writes from the bitmap into the output queue vector adhering to worker regions using the prefix-sum.

## 5 Experimental Setup

We have implemented iPUG in under 2000 lines of code, including the code required to read and process Matrix Market files. We compile our project with the *Poplar* 1.3.6 SDK and *popc* running on a single GC2 IPU.

Based on the guidelines of the the Graph 500 benchmark, we split our measurements into two kernels: (1) the reading, preparing, and loading of the graph onto the device, and (2) the BFS graph traversal itself. Since our goal is to evaluate BFS performance on the IPU architecture, we concentrate on the second kernel. We begin measuring time of the second kernel $t$ when the search key is loaded onto the device. We stop measuring when the final BFS round terminates.

Following the codes we aim to compare our results with [27, 32], we count TEPS from both sides for undirected edges. As per Graph 500 specification, we ignore isolated search keys. Thus, since all our test instances are connected with the exception of isolated vertices, we always report TEPS:= $m/t$ where $m$ is the number of non-zero entries in the adjacency matrix that connect visited edges. Due to limitations in some of the codes, we report the arithmetic rather than the harmonic or geometric mean over the prescribed 64 searches.

We do not perform any special operations in the first kernel such as sorting vertices or finding vertices with special properties. However, we are filtering self-loops and vertices of degree zero from the graph while converting it into the CSC format required by our 2D decomposition algorithm. In the 2D decomposition algorithm we are splitting the matrix into square $n/p_x$ by $n/p_y$ sized parts. We always use a square processor grid, i.e. $p_x = p_y$. Since the number of cores on the GC2 IPU is 1216, the largest smaller square number is 1156, and thus $p_x = p_y = 34$. The remaining 60 cores do not take part in the computation.

To measure the runtime of the second kernel executed on the IPU, we measure the start and end cycle counter of the IPU and divide the difference by the tile frequency returned by the Poplar SDK. We run our experiments on an IPU-POD system. It does not have the power limitations of the PCIe version and is thus running at the full 1.6 GHz. As most runs only take microseconds, thermal throttling is no concern either. For each run we randomly generate 64 keys that have at least one edge connected to it in the input graph. We run the second kernel with all given keys and take the mean.

**Test Instances** We use both Graph500 instances as well as graphs derived from SuiteSparse [22] matrices. The matrices were selected to match a published test set [32] after removing all instances that are too large to run on the IPU. Table 2 lists all the instances along with their size and diameter. The sources of the graph come from the following groups:

- **kron_(n)_(e)** are Kronecker graphs with $2^{\mathbf{n}}$ vertices and edge factors **e**. The edge factor is the average number of edges per vertex. Graphs with larger values of **e** typically show higher TEPS as work is being amortized over a larger number of edges. Graphs generated by the Graph500 benchmark specification have **e** = 16 and can be used to compare implementations to other published Graph500 results. All graphs were generated with R-MAT parameters $A = 0.57$, $B = 0.19$, $C = 0.19$, and $D = 0.05$. Note that we filter isolated vertices. Thus, the number of vertices in the BFS is always lower than $2^{\mathbf{n}}$.
- **kron_g500-logn(n)** are Kronecker graphs from the 10th DIMACS implementation challenge. Despite the SuiteSparse name these graphs are not conform to the Graph500 benchmark, as they have an edge factor of 48, but they use the same R-MAT parameters as the Graph500 instances.
- **G43** represents a 1% sparse uniformly random matrix.
- **coAuthorsDBLP** and **coPapersDBLP** are academic research interaction and cooperation networks.
- **Journals** represent co-readerships in magazines.
- **delaunay_(n)** are planar graphs from the 10th DIMACS implementation challenge. They are generated by the triangulation of points in a flat area, with size $2^n$.
- **loc-Gowalla** represents friendships of a social network based on location data retrieved from the SNAP suite.
- **ship_003** represents a 3D mesh of a structural problem by the DNVS group.

**Comparison Platforms** As the Graphcore IPU is a completely new architecture, it is crucial to assess its performance in comparison to established processors. For comparison with the GPU we use two state of the art codes: **Enterprise** created by Hang Liu and H. Howie Huang [27] and **Gunrock** by Yangzihao Wang et al. [13, 31]. The Gunrock[6] and Enterprise[7] code were both run on an NVIDIA Tesla V100-SXM3 with 32GB of memory compiled with nvcc 10.1 and clang 11.0.0. Like the IPU, the V100 runs at 1.6 GHz.

As the performance benefits of the GPU over the CPU are well established, we consider this the primary point of comparison. However, we also study CPU performance. For that purpose, we use the Graph 500 BFS reference (Ref) implementation [28] which relies on MPI, a sophisticated MPI/OpenMP implementation provided by Yasui et al. [33] from Tokyo Institute of Technology (TITech), and the BFS implementation from the GAP benchmark suite [6]. The latter

[6] Git commit: 5ee3df5, Online: https://github.com/gunrock/gunrock
[7] Git commit: 426846f, Online: https://github.com/iHeartGraph/Enterprise

| SuiteSparse | | | | Generated | | | |
|---|---|---|---|---|---|---|---|
| **Name** | **Diam** | **Vertices** | **Edges** | **Name** | **Diam** | **Vertices** | **Edges** |
| G43 | 4 | 1K | 10K | kron19_16† | 8 | 356K | 8M |
| coAuthorsDBLP | 24 | 300K | 978K | kron19_16.2† | 8 | 356K | 8M |
| Journals | 2 | 124 | 6K | kron19_16.3† | 7 | 356K | 8M |
| coPapersDBLP | 23 | 540K | 15M | kron18_16† | 8 | 197K | 4M |
| loc-Gowalla | 16 | 197K | 950K | kron17_16† | 7 | 118K | 2M |
| ship_003 | 58 | 122K | 4M | kron16_16† | 8 | 66K | 1M |
| delaunay_n12 | 36 | 4K | 12K | kron15_16† | 6 | 33K | 524K |
| delaunay_n13 | 49 | 8K | 25K | kron19_48 | 7 | 432K | 25M |
| delaunay_n14 | 65 | 16K | 49K | kron19_32 | 8 | 393K | 16M |
| delaunay_n15 | 87 | 33K | 98K | kron18_128 | 6 | 236K | 34M |
| delaunay_n16 | 119 | 66K | 197K | kron18_96 | 6 | 236K | 25M |
| delaunay_n17 | 167 | 131K | 393K | kron18_64 | 7 | 236K | 17M |
| delaunay_n18 | 228 | 262K | 786K | kron18_32 | 7 | 236K | 8M |
| kron_g500-logn16 | 6 | 66K | 2M | kron16_32 | 7 | 66K | 2M |
| kron_g500-logn17 | 6 | 118K | 5M | kron15_32 | 6 | 33K | 1M |
| kron_g500-logn18 | 6 | 236K | 11M | kron17_32 | 7 | 118K | 4M |
| kron_g500-logn19 | 7 | 432K | 22M | | | | |

Table 2: Overview of the test instances. All graphs are undirected. Thus their adjacency lists contain twice as many entries as the number of edges. The diameter represents the longest path found during the BFS runs. Datasets marked with (†) conform to the Graph500 benchmark specification.

has the advantage that it reads the Matrix Market format. We thus use it for comparison on SuiteSparse matrices outside of the Graph 500.

We run all three codes on two dual-socket CPU platforms, an AMD Epyc 7601 with 64 total cores and an Intel Xeon Gold 6130 with 32 total cores. Since the CPUs are not the focus of this paper, we refer the reader to online resources[8,9] or the manufacturer's documentation for more information about their technical specifications. The codes are compiled with gcc 6.1.2 and run with MPICH 3.3.

## 6 Experimental Results

### 6.1 Performance Comparison Experiment

Our experimental results are collected in Figures 3 and 4. They show the performance of iPUG on the IPU compared to the GPU codes on the V100 and GAP on the Intel Xeon, along with the speedup of the IPU compared to the fastest alternative. iPUG shows the highest speedups for very small instances.

[8] https://en.wikichip.org/wiki/amd/epyc/7302p
[9] https://en.wikichip.org/wiki/intel/xeon_gold/6130

This is understandable since the CPU and GPU codes are not designed for such instances. However, on the largest and thus most relevant Kronecker instances that fit in IPU memory, we still observe a speedup of about 1.5×.

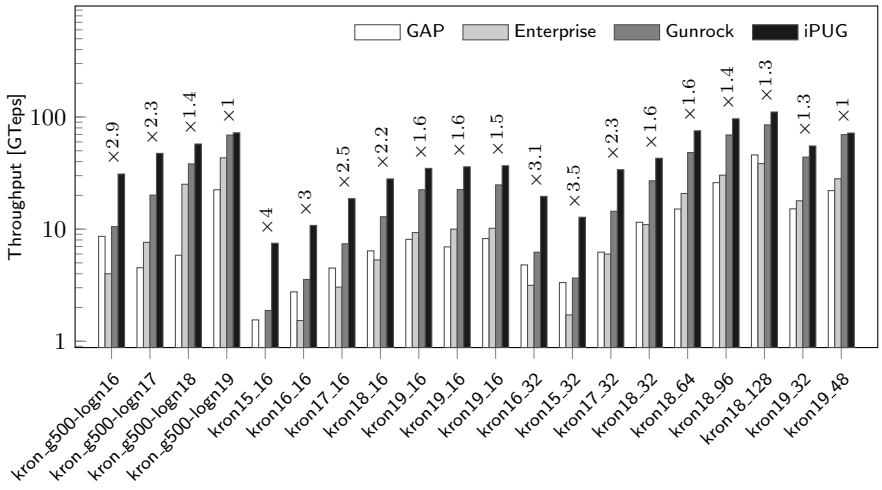


Fig. 3: Performace of iPUG compared to CPU and GPU for the Kronecker graphs.

For the Suitesparse graphs, we observe 3× speedups for smaller and 1.5× speedups for the larger *DBLP* instances over Gunrock, which is the best alternative here. An exception are the larger and thus higher diameter *delaunay* graphs which exhibit little parallelism. On average there are far fewer vertices in the frontier each round than the IPU has threads, thus making the wide parallelism inefficient. As a result, the CPU performs better than both IPU and GPU, although the difference between CPU and IPU is small. The only instance where the GPU exceeds IPU performance is the very small and dense *Journals*, and even there the difference is very small.

### 6.2 Graph 500 Scaling Experiment

In an additional experiment, we show the performance of the IPU in context of the scaling behaviour of other BFS implementations. Results are shown in Figure 5. We observe that the CPU type has little influence for all three codes. On the other hand, the TiTech code is almost an order of magnitude faster than GAP and the reference code, reaching almost 10 GTEPS. The CPU codes seem to reach maximum performance at Scale 22.

The GPU implementations are consistently faster, with Gunrock reaching almost 100 GTEPS at Scale 24. It also maintains a consistent and substantial

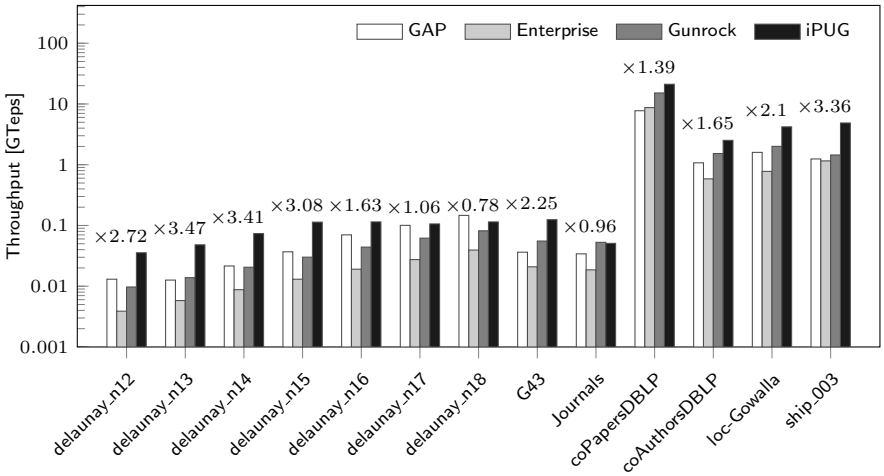


Fig. 4: Performace of iPUG compared to CPU and GPU for the Suiteparse instances.

lead over Enterprise. Furthermore, while iPUG starts with a large advantage at Scale 15, the gap closes to 1.5× at Scale 19. Thus, due to the limitations in IPU memory, it is not possible to say at which scale maximum IPU performance will be attained, and whether it would be faster than Gunrock on the V100. Since the larger instances have a higher fraction of isolated vertices, and removing such vertices has a substantial effect on IPU performance, it is possible that the IPU would maintain its lead if it had more memory.

An important insight from these results is that implementations may affect performance more than the hardware platform. This effect is certainly visible for the CPUs. Furthermore, GPUs were initially not widely considered a suitable architecture for BFS, but steady algorithmic advances have made GPUs highly competitive for the specific problem of BFS on Kronecker graphs.

In addition to direction optimization [7], sophisticated GPU codes explicitly cache the status of high degree vertices in shared memory during the backwards search phase, as suggested for the Enterprise BFS code [27]. This obviates the need for about 80% of all status queries, thereby improving performance dramatically. However, the technique is far less effective for other types of graphs. Furthermore, it creates a point of performance which depends on the size of the programmer-controlled shared memory. For both GPU codes, performance seems to decrease when going towards Scale 25. Naturally, the IPU cannot replicate this technique since it lacks a memory hierarchy in which such caching could take place.

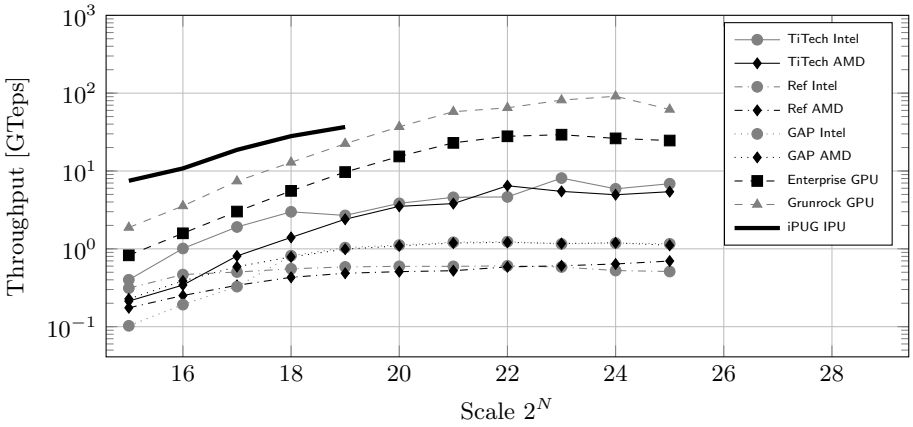


Fig. 5: Performance of Graph500 Kron-$N$-16 graphs by scale on all tested codes and architectures.

## 7 Discussion

We have tested our BFS code on the IPU and achieved speedups between 0.96× and 4× over the fastest GPU code, with a typical speedup of 1.5× for the largest feasible Kronecker graphs. The GPU results could certainly be improved by running on an NVIDIA Ampere A100 or AMD Instinct MI100 GPU, while the IPU results will benefit from the larger memory and increased core count of the M2000 IPU once it becomes widely available. However, the M2000 IPU does not provide a large increase in memory bandwidth or clock frequency, which means that the latest hardware generation could close the current gap between GPU and IPU to some extent. Even so, we expect that the IPU will maintain a lead for most instances.

Furthermore, based on the memory bandwidth of the IPU, it is conceivable that a far higher performance is possible. During the first few years after its inception, the Graph500 [28] performance results increased massively, but improvements have slowed down substantially thereafter. While we have considered several optimizations on the IPU, we are far from having exhausted its possibilities. We were not able to show performance improvements via direction optimizing search, although in principle such algorithmic improvements can be applied on the IPU. Thus, it is likely that faster Graph500 results will appear in the future.

Naturally, the small memory of the IPU limits its application to real-world problems. Furthermore, it is debatable whether it is fair to compare an SRAM based device to a DRAM based processor since the IPU is essentially running out of what would be cache on a CPU. However, our results indicate that the CPU does not experience a similar speedup when running on the smallest instances which certainly fit inside the L3 cache of the Intel Xeon or AMD Epyc. This is consistent with an observation from the 2018 Turing lecture [18], which

points out that programmer controlled scratchpad memory offers significant performance advantages compared to transparent general-purpose caches. In case of the IPU, no additional programming complexity is incurred by this, since the memory hierarchy only has a single level.

## 8 Conclusion

We have implemented the first BFS code on the Graphcore IPU and thus presented the first benchmark results of a graph algorithm on that platform. The results typically show 1.5× speedups over the fastest competing GPU and CPU codes, thus demonstrating the potential of this new architecture for graph algorithms. The main limitation to its usefulness is the small memory of the IPU. This means that it is more suited to algorithms with higher time complexities such as matching, betweenness centrality, or even NP-hard optimization problems. Furthermore, kernelization techniques [2, 21] will become even more valuable if they allow shrinking problems to the point of fitting into IPU memory. However, the main challenge in future work will be to scale graph problems to multiple IPUs in order to overcome the memory limitations. While the IPU programming model extends transparently to multiple IPUs, it is likely that substantial optimizations will be needed to scale up its performance. Consequently, future work will focus on scaling BFS to multiple IPUs, as well as use the current code as a basis to implement more sophisticated graph algorithms.

## References

1. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al.: Tensorflow: A system for large-scale machine learning. In: 12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16). pp. 265–283 (2016)
2. Abu-Khzam, F.N., Collins, R.L., Fellows, M.R., Langston, M.A., Suters, W.H., Symons, C.T.: Kernelization algorithms for the vertex cover problem (2017)
3. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers, principles, techniques, and tools. Addison-Wesley Pub. Co (1986)
4. Azad, A., Buluç, A.: Distributed-memory algorithms for maximum cardinality matching in bipartite graphs. In: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 32–42. IEEE (2016)
5. Bader, D.A., Madduri, K.: Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2. In: 2006 International Conference on Parallel Processing (ICPP'06). pp. 523–530. IEEE (2006)
6. Beamer, S., Asanović, K., Patterson, D.: The gap benchmark suite. arXiv preprint arXiv:1508.03619 (2015)
7. Beamer, S., Asanovic, K., Patterson, D., Beamer, S., Patterson, D.: Searching for a parent instead of fighting over children: A fast breadth-first search implementation for graph500. EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2011-117 (2011)

8. Buluç, A., Beamer, S., Madduri, K., Asanovic, K., Patterson, D.: Distributed-memory breadth-first search on massive graphs. arXiv preprint arXiv:1705.04590 (2017)
9. Buluç, A., Gilbert, J.R.: The combinatorial blas: Design, implementation, and applications. The International Journal of High Performance Computing Applications **25**(4), 496–509 (2011)
10. Buluç, A., Madduri, K.: Parallel breadth-first search on distributed memory systems. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–12 (2011)
11. Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-mat: A recursive model for graph mining. In: Proceedings of the 2004 SIAM International Conference on Data Mining. pp. 442–446. SIAM (2004)
12. Checconi, F., Petrini, F.: Traversing trillions of edges in real time: Graph exploration on large-scale parallel machines. In: 2014 IEEE 28th International Parallel and Distributed Processing Symposium. pp. 425–434. IEEE (2014)
13. Chenglong, Z., Huawei, C., Guobo, W., Qinfen, H., Yang, Z., Xiaochun, Y., Dongrui, F.: Efficient optimization of graph computing on high-throughput computer. Journal of Computer Research and Development **57**(6), 1152 (2020)
14. Gaihre, A., Wu, Z., Yao, F., Liu, H.: Xbfs: exploring runtime optimizations for breadth-first search on gpus. In: Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing. pp. 121–131 (2019)
15. Ghosh, R.K., Bhattacharjee, G.: Parallel breadth-first search algorithms for trees and graphs. International Journal of Computer Mathematics **15**(1-4), 255–268 (1984)
16. Gregor, D., Lumsdaine, A.: Lifting sequential graph algorithms for distributed-memory parallel computation. ACM SIGPLAN Notices **40**(10), 423–437 (2005)
17. Harish, P., Narayanan, P.J.: Accelerating large graph algorithms on the gpu using cuda. In: International conference on high-performance computing. pp. 197–208. Springer (2007)
18. Hennessy, J.L., Patterson, D.A.: A new golden age for computer architecture. Communications of the ACM **62**(2), 48–60 (2019)
19. Hong, S., Oguntebi, T., Olukotun, K.: Efficient parallel graph exploration on multi-core cpu and gpu. In: 2011 International Conference on Parallel Architectures and Compilation Techniques. pp. 78–88. IEEE (2011)
20. Jia, Z., Tillman, B., Maggioni, M., Scarpazza, D.P.: Dissecting the graphcore ipu architecture via microbenchmarking. arXiv preprint arXiv:1912.03413 (2019)
21. Kaya, K., Langguth, J., Panagiotas, I., Uçar, B.: Karp-sipser based kernels for bipartite graph matching. In: 2020 Proceedings of the Twenty-Second Workshop on Algorithm Engineering and Experiments (ALENEX). pp. 134–145. SIAM (2020)
22. Kolodziej, S.P., Aznaveh, M., Bullock, M., David, J., Davis, T.A., Henderson, M., Hu, Y., Sandstrom, R.: The suitesparse matrix collection website interface. Journal of Open Source Software **4**(35), 1244 (2019)
23. Korf, R.E., Schultze, P.: Large-scale parallel breadth-first search. In: AAAI. vol. 5, pp. 1380–1385 (2005)
24. Langguth, J., Azad, A., Halappanavar, M., Manne, F.: On parallel push–relabel based algorithms for bipartite maximum matching. Parallel Computing **40**(7), 289–308 (2014)
25. Langguth, J., Cai, X., Sourouri, M.: Memory bandwidth contention: Communication vs computation tradeoffs in supercomputers with multicore architectures. In: 2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS). pp. 497–506. IEEE (2018)

26. Langguth, J., Patwary, M.M.A., Manne, F.: Parallel algorithms for bipartite matching problems on distributed memory computers. Parallel Computing **37**(12), 820–845 (2011)
27. Liu, H., Huang, H.H.: Enterprise: breadth-first graph traversal on gpus. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–12 (2015)
28. Murphy, R.C., Wheeler, K.B., Barrett, B.W., Ang, J.A.: Introducing the graph 500. Cray Users Group (CUG) **19**, 45–74 (2010)
29. Seshadhri, C., Pinar, A., Kolda, T.G.: An in-depth analysis of stochastic kronecker graphs. Journal of the ACM (JACM) **60**(2), 1–32 (2013)
30. Valiant, L.G.: A bridging model for parallel computation. Communications of the ACM **33**(8), 103–111 (1990)
31. Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A., Owens, J.D.: Gunrock: A high-performance graph processing library on the gpu. In: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 1–12 (2016)
32. Yang, C., Buluc, A., Owens, J.D.: Graphblast: A high-performance linear algebra-based graph framework on the gpu (2020)
33. Yasui, Y., Fujisawa, K., Goto, K.: Numa-optimized parallel breadth-first search on multicore single-node system. In: 2013 IEEE International Conference on Big Data. pp. 394–402. IEEE (2013)
34. Yoo, A., Chow, E., Henderson, K., McLendon, W., Hendrickson, B., Catalyurek, U.: A scalable distributed parallel breadth-first search algorithm on bluegene/l. In: SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing. pp. 25–25 (Nov 2005). https://doi.org/10.1109/SC.2005.4
35. Yoo, A., Chow, E., Henderson, K., McLendon, W., Hendrickson, B., Catalyurek, U.: A scalable distributed parallel breadth-first search algorithm on bluegene/l. In: SC'05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing. pp. 25–25. IEEE (2005)