Technische Universität Berlin
Electrical Engineering and Computer Science
Department of Telecommunication Systems
Distributed and Operating Systems (DOS)

Accelerating Breadth First traversals using AI accelerators

Masterarbeit

von Luk Burchard

zur Erlangung des Grades „Master of Science“ (M. Sc.)
im Studiengang Computer Science

Erstgutachter:	Prof. Dr. habil. Odej Kao
Zweitgutachter:	XXXXXXXXXXXXX
Erstbetreuer:	Daniel Thilo Schroeder
Zweitbetreuer:	Dr. Johannes Langguth

30 March

Erklärung der Urheberschaft

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Ort, Datum	Unterschrift

Zusammenfassung

Die Graphcore Intelligence Processing Unit (IPU) ist ein neu entwickelter Prozessortyp, dessen Architektur sich nicht auf die traditionellen Caching-Hierarchien stützt. Entwickelt, um den Bedarf an immer mehr datenzentrierten Anwendungen, wie z. B. maschinelles Lernen, zu decken, kombinieren IPUs einen dedizierten Teil des SRAM mit jedem ihrer zahlreichen Kerne, was zu einer hohen Speicherbandbreite, aber zu Kosten in Form der globalen Speicherkapazität führt. Die Nähe von Prozessorkernen und Speicher macht die IPU zu einem vielversprechenden Experimentierfeld für Graph Algorithmen, da es die unvorhersehbaren, unregelmäßigen Speicherzugriffe sind, die bei traditionellen Prozessoren mit Pre-Caching zu Leistungsverlusten führen.

Ziel dieser Arbeit ist es, die Eignung der IPU für Algorithmen mit schwer vorhersagbaren Speicherzugriffen zu testen, indem eine Breadth-First-Suche (BFS) implementiert wird, die den Spezifikationen des Graph 500 entspricht. Gerade wegen seiner scheinbaren Einfachheit ist BFS ein etablierter Benchmark, der nicht nur für eine Vielzahl komplexerer Graphenalgorithmen als Unterprogramm dient, sondern auch eine Vergleichbarkeit über eine Vielzahl von Architekturen ermöglicht.

Benchmarks der IPU-Codes auf einer Vielzahl von Instanzen und vergleichen seine Leistung mit modernsten CPU und GPU-Codes. Die Ergebnisse zeigen, dass die IPU einen Geschwindigkeitszuwachs von bis zu $4\times$ gegenüber dem schnellsten konkurrierenden Ergebnis auf einer NVIDIA V100 GPU liefert, mit typischen Geschwindigkeitssteigerungen von ca. $1.5\times$ auf den meisten Testinstanzen.

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 Graph 500 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.

Benchmarks of the IPU code on a wide range of instances 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.

Contents

List of Figures

1 Introduction

Moore's Law is probably the most famous theorem which dictates the market and development of microprocessors and software. From the inception of the first microcontrollers in 1970 to 2010, developers had a simple time improving their code by waiting for the next hardware generation. However, since 2010 the clock frequency of a single processor core is not increasing due to the available thermal capacity. All available transistors are placed into more complex logic to optimize the instructions a single core is executing. Some famous techniques are prefetching of data, deep cache hierarchies, or speculative execution. Even though all of these optimizations are transparent to the machine developers, they are struggling with optimizing their code to optimally utilize the processor's capabilities for a wide range of problems.

Instead of spending the transistor budget on increasingly complex cores, hardware architects shifted towards multicore architectures (Figure 1.1). Nonetheless, these multicore architectures still have most optimizations implemented to increase the effective operation throughput. Henceforth, traditional general-purpose processors operate on shared memory spaces and have a number of cores in the order of 4 to 32, still limited by the complexity of the cores.

As the big performance increases diminished, since the 2010s, new ways needed to be found to work with the growing demand of specialized problems such as AI/ML or graph analytics. Therefore, more specialized architectures become viable, which are not requiring the generalized processor overhead. A trend in hardware architecture is the development of domain-specific AI/ML processors tailored to make machine learning faster. Bespoke processors often work with

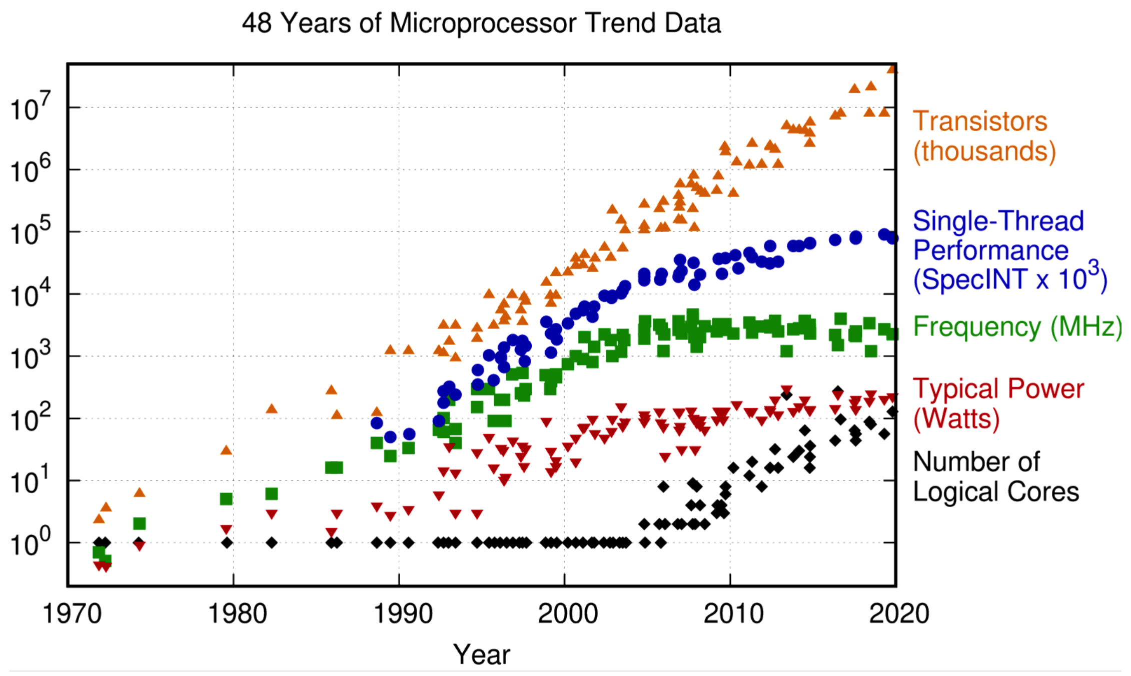

Figure 1.1: Development of processor frequeny while upholding Moore's Law. Credits and Copyright at M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, C. Battne, and K. Rupp.

different assumptions compared to CPUs and GPUs. AI/ML accelerators often provide many more simple cores with no complex optimizations and work with a shared noting memory model. These processors have numbers of cores in the order of 1000 to $350,000$ and up to $850,000$[Roc+20].

One of the companies developing novel accelerators is Graphcore, which 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 computing 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.

Graph algorithms still remain a challenging problem in the field of distributed computing and high performing computing due to their irregular structure, paired with the need to share pieces of information between every edge. This makes it even difficult[1] to efficiently solve graph algorithms in big distributed Massively Parallel Computation frameworks such as MapReduce.

We want to understand how useful IPUs are to solve challenging graph problems. As we are not able to implement all kinds of graph algorithms, we need to find a representative problem for general graph algorithms. Henceforth, we choose a breadth-first search as it contains irregular communication along the edges and the algorithm is a heavily used subroutine of more complex algorithms. Solving breadth-first search in a fast manner will likely be a good indicator for implementing efficient graph algorithms on top of our work and IPUs in general.

In theory, the manycore design makes the IPU uniquely suited for highly irregular workloads such as graph algorithms. The goal of this work is to test whether all of the IPUs architectural advantages result in measurable performance benefits.

1.1 Contributions

We implement an IPU-based breadth-first search (BFS), following the specifications of the Graph500 [Mur+10] 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 [CZF04]. Results are denoted in traversed edges per second (TEPS). In addition, we use a test set of Yang et al. [YBO20], which consists of matrices from the SuiteSparse [Kol+19] 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 algorithms used in the analysis of social networks, network optimization, cyber

[1]https://ai.googleblog.com/2021/03/massively-parallel-graph-computation.html

security, network centralities, graph matchings, bioinformatic and more domaines.

While the IPU's large number of independent compute cores, fast interconnect between these cores, and fast SRAM memory makes 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 get 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 compares favorably to all tested alternatives.

1.2 Thesis Outline

This thesis is organized as follows:

In Section 2 we introduce the IPU hardware architecture. Furthermore, we are introducing the programming model and provide examples of how to programmatically interact with Graphcore's programming framework.

In Section 3 we provide related work and discuss parallel BFS work on other architectures. Further, we show BFS optimizations and implementation background for general linear algebra and sparse matrices.

In Section 4 we present our IPU implementation and algorithmic adaptations for running BFS on the IPU hardware. We show and discuss challenges, workarounds, and optimizations techniques for efficiently interacting with the IPU.

In Sections 5 we describe our experiments and datasets used, and **in Section 6** we show our results and analyse the hardware utilizing.

In Section 7 we discuss the experience of developing and adapting algorithms for the IPU, together with tradeoff and limitation due to time and effort.

In Section 8 we discuss the implications of the results and conclude the thesis.

In Section 9 we outline the next research directions extending and building on top of this work.

This work has also been accepted to the International Supercomputing Conference on High Performance 2021 (ISC21)

2 IPU Hardware

2.1 Architecture

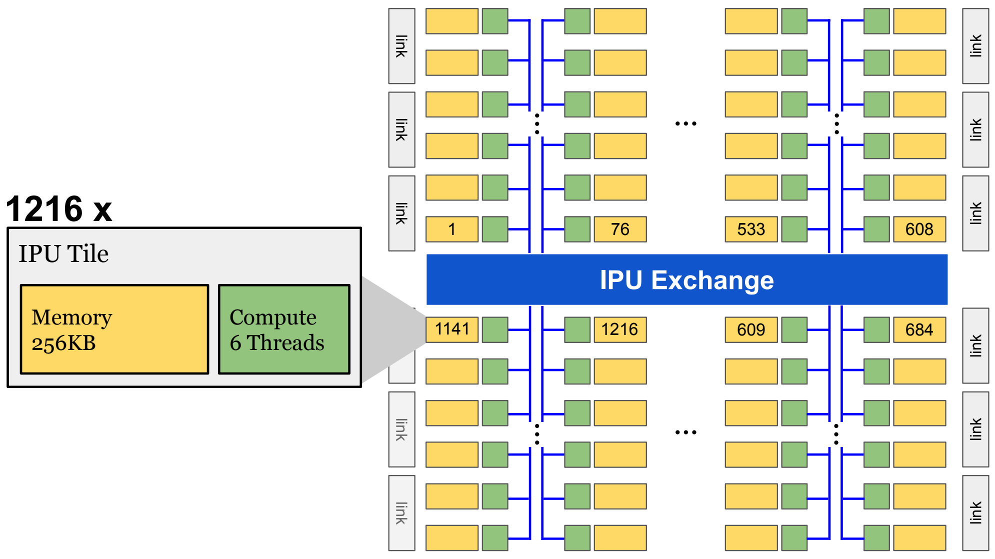

Figure 2.1: Tile layout on the IPU processor.

The Colossus GC2 IPU has 1216 *tiles*, each tile having a compute core and its own local memory of 256 KB. Thus, the IPU has a total of 304 MB of memory. The tile layout is illustrated in Figure 2.1. 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, and the GC2 IPU has 16 such columns

(Table 2.1). 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. [Jia+19]. In 2020, Graphcore presented the GC200 IPU with more tiles and more memory per tile, but the device was not available for development at the time of this writing.

Element	**Contents**
One Tile	Core and Local Memory (256 KB)
Island	4 Tiles
Column	76 Tiles (19 ”Islands”)
IPU	1216 Tiles (16 Columns)

Table 2.1: Hierarchy of components in the IPU

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 [Aba+16] 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 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) [Val90] superstep structure. Figure 2.2 depicts such a synchronized graph.The rationale for this structure is that due to bandwidth contention, overlapping memory-bound computation and communication is difficult and sometimes impossible [LCS18]. 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, that all communication must be planned at compile time. Therefore, this poses a challenge when communicating sparse data, which is necessary in graph algorithms such as BFS.

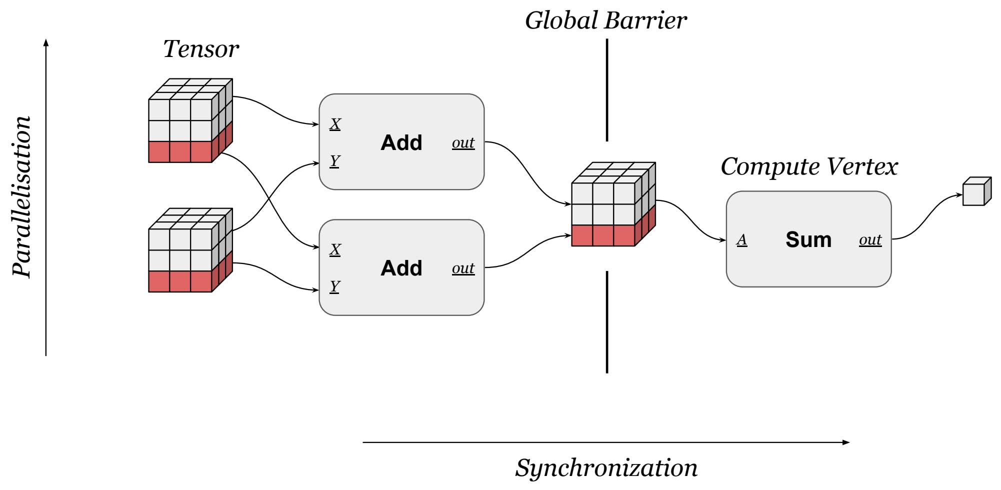

Figure 2.2: Poplar's data-flow graph represents data as *Tensors* and computation as multi input-output Verticies. Data from Tensors regions (gray,red) get copied to tiles running *Verticies* which output into Tensor regions. Parallelisation is done through subslicing Tensors and running multiple instances of Verticies on different tiles and threads. A *Global Barrier* is introduced to synchronize data writes before reads.

2.2.1 Programming Stack

The native interface to the IPU is given through *Poplar* the low-level programming interface, which does not abstract the use of the IPU on a higher level such as Tensorflow or PyTorch. Poplar is the lowest level access Graphcore provides to a user and offers the most control over the individual aspects of the hardware. Higher-level libraries build on top of the poplar functionality (Figure 2.3). The interface to the hardware is given through a data flow defining the interface in which a compute graph is structured and data regions (tensors) defined. The user has to assign each part of a tensor to an individual tile, and each compute-vertex to a tile. The graph compiler creates an executable for the IPU, which is runnable through Poplar. Poplar calls into `popc` the graph compiler responsible for compiling codelet files written in C++ to emit IPU for later use to construct the graph executable, and a CPU ELF for emulation and testing. Poplar exposes primitives to upload and download data to the IPU pre and during runtime. Graphcore provides the users of Poplar with a set of handcrafted and optimized vertices and mapping functions that reassemble a higher-level operation library. The collection of these is referred to as the Poplibs. Poplibs contain important functionality such as `poplin` for doing linear algebra functions such as dense matrix multiplication, `popops` general operations and functions on tensors such as the determinant or the dot product. `poputil` provides helper functions for data mapping and convenience methods for abstracting higher-level graph operations.

The Poplar libraries are written in C++ but offer no incentive for optimized codes as the main purpose of the APIs is to create a dataflow graph. The only incentive for high-performance programming arises from interacting with callbacks on IPU data streams.

Graphcore is selling the IPU for machine learning and thus offers higher-level access to their APIs. They support common machine learning frameworks such as Tensorflow which compiles kernels and dataflow execution graphs with the Accelerated Linear Algebra (XLA). XLA offers a common intermediate representation and module-wise compiler architecture, thus making it possible to connect Poplar as a backend target to XLA.

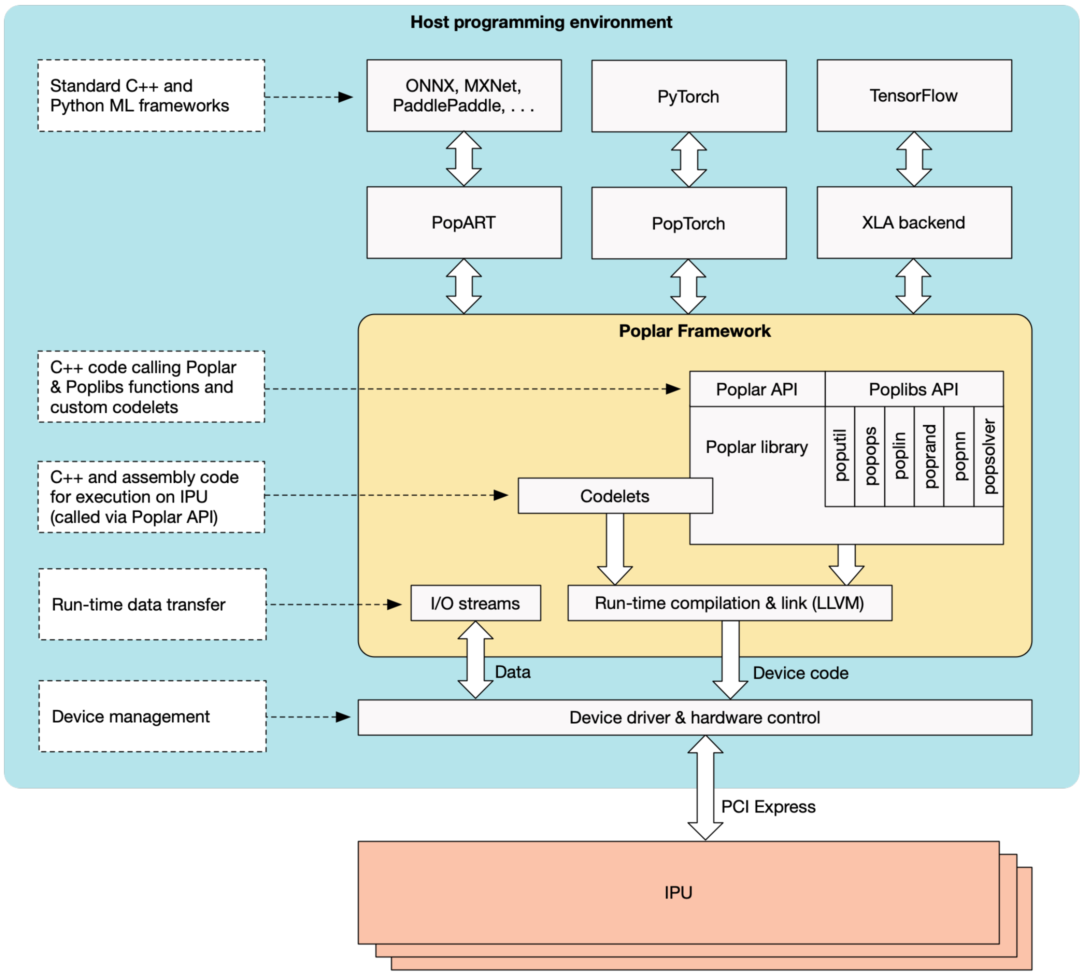
Host programming environment
Standard C++ and Python ML frameworks
ONNX, MXNet, PaddlePaddle, . . .
PyTorch
TensorFlow
PopART
PopTorch
XLA backend
Poplar Framework
C++ code calling Poplar & Poplibs functions and custom codelets
Poplar API
Poplibs API
Poplar library
poputil
popops
poplin
poprand
popnn
popsolver
C++ and assembly code for execution on IPU (called via Poplar API)
Codelets
Run-time data transfer
I/O streams
Run-time compilation & link (LLVM)
Data
Device code
Device management
Device driver & hardware control
PCI Express
IPU

Figure 2.3: Poplar's software stack has multiple abstraction levels over the IPU hardware. The lowest level is the Poplar framework, with capabilities to write user-defined vertices and construct data flow graphs. The higher levels are provided by common ML frameworks, which transparently compiles to the Poplar Framework. Copyright at Graphcore.

2.2.2 Codelets

When defining a compute graph, each codelet in a Vertex is a function that takes multiple input tensors and output tensors. When working with the Poplar API the user connects all tensor inputs and outputs with the vertex. During runtime the data regions get copied to and from the tile the vertex was placed on. The user does not have to manage the exchange code, as it is done automatically by the framework.

All user-defined codelets are written through a C++ templated interface and are provided to Poplar as vertices that take in tensors and output or modify tensors. These vertices can be seen analogous to functions operating on data via reference to copy conventions. Codelets are defined as classes that inherit from the `poplar::Vertex` type and overwrite the `compute()` method. The compute method is the entry point called after the vertex is fully set up and all data is available; the normal execution context is in the hardware worker mode, which is set up from the supervisor mode.

Listing 1 implements the operation $\vec{a} = \vec{x} \circ \vec{y}$, where $(\circ)$ represents the pointwise Hadamard-product and $\vec{a}, \vec{x}, \vec{y} \in \mathbb{Q}^n$. Each vertex defines its input and output fields by its class values that are connected to real tensors. The framework will copy said class variables before the entry function is run and write the output results back once it is finished. Class variables of type `poplar::Input<T typename>` are used for input data structures, such as `poplar::Vecotr<float>` tensor vector of type float. Outputs are defined with `poplar::Output<T typename>` analogously. Furthermore, scalars and multidimensional nested vectors are accepted. For convenience, the index and the dereference operators are implemented on top of poplar input and output data structure to iterate, read and modify over them in a common C/C++ syntax loop.

It is noteworthy that output data is undefined when read and input data emits undefined behavior when written.

```cpp
#include <poplar/Vertex.hpp>

class ProductVertex : public poplar::Vertex {
public:
  poplar::Input<poplar::Vecotr<float>> x;
  poplar::Input<poplar::Vector<float>> y;
  poplar::Output<poplar::Vector<float>> result;

  bool compute() {
    for (auto i = 0; i < result.size(); i++) {
        result[i] = x[i] * y[i];
    }
    return true;
  }
};
```

Listing 1: Vertex code used to programm on the IPU, all code running on the IPU is a contained within a C++ `Vertex` class.

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 [GB84], the topic gained widespread interest in the previous decade on distributed memory computers [GL05; Yoo+05], on shared memory [BM06; KS05], and on GPU systems [HN07]. The establishment of the Graph500 benchmark [Mur+10] 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 [BM11; Che+20; CP14; HOO11; YFG13]. Furthermore, BFS implementations for GPUs have also received considerable attention in the recent years [Gai+19; LH15; Wan+16; YBO20]. 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 [Bea+11]. 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 [AB16; Lan+14; LPM11].

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 [BG11] 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 (Section 3.5).

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.

3.3 Bulk Synchronus Parallel

For aiding the design and understanding costs of parallel algorithms with distributed memory Leslie Valiant[Val90] created the Bulk Synchronous Parallel (BSP) machine model. The BSP model does not abstract away communication and separates the synchronization and communication into different steps, other than the more common RAM/PRAM model, which focuses on a single global memory region.

The model uses a set of parallel processors, able to run arbitrary instructions, while each processor has only access to its local memory. Further, each processor is connected to a network and can exchange data with any other processor. The exchange between the processors is deterministic and happens in dedicated queues per processor pair.

Each execution in the BSP model consists of sequentially executed supersteps $S_1, \dots, S_n$, which are formed from three separate phases shown in figure 3.1, within a superstep exists no notion of synchronization or order between the processors:

1. **Computation:** In parallel, work on a local processor with input data from S_{i-1} and generate output data for use in S_{i+1}.

2. **Exchange:** Communicate the output data of the processors to input data fields of another processor. All communication between processors happen in this phase.

3. **Global Sync:** Globally synchronize each parallel processor and end a full superstep such that all input data for the next computation is locally available on each processor.

This strict model makes it easy to reason about different costs, possible deadlocks, and it inherently avoids race conditions. Making it suitable for complex, large-scale parallel algorithms on distributed hardware.

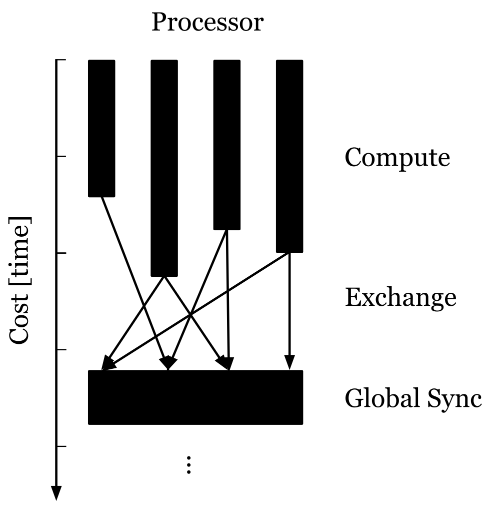

Figure 3.1: Visualization of a BSP superstep. Each superstep runs three phases (compute, exchange, global sync) in series parallel and synchronize at the end of the superstep. The costs of each step can the read from the time axis.

The BSP model gained interest in the last years, as it went from a theoretical bridging model to an implemented model used in libraries and frameworks. Notable implementations for scientific parallel programming are the BSPLib[Hil+98], however, more recently attention for the BSP model came from big-data frameworks utilizing MapReduce[DG08] like Hadoop[DQR12] or Pregel[Mal+10] which are building upon the BSP model. Further to these additions, the IPU builds its programming and hardware model around the BSP model.

3.4 Direction Optimizing BFS

The Graph500 benchmark uses scale-free Kronecker graphs that emit a small diameter and fall under the category of small-world graphs, often found in social media network structures to benchmark and compare HPC systems.

All graphs generated under these parameters omit an exponential exploration rate of the traversed edges, combined with an average high edge-factor this leads to multiple parents claiming a child in an *explosion level* of the tree traversal.

Beamer et al.[BAP12] used a *bottom-up* pass in combination with a common *top-donw* pass to reduce the amount of lookups and fights over children in the explosion levels. A bottom-up pass, opposite to the top-down pass, does not expand the children list outgoing from the parents but searches from every unvisited node in the parent list for one that is currently activated in the frontier. Further, they provide a heuristic to switch between the two phases reaching an on average 25% of a perfect oracle. Using the direction optimizing BFS (DOBFS) Beamer et al. archived a more than $3.3 - 7.8\times$ speedups on Kronecker and other synthetic graphs, entering the Graph500 leaderboard in November 2011 as the fastest single-core result.

The top-down pass is efficient when only a few nodes need to be expanded and the network is unexplored, as all operations are almost certain to only claim children that have not been seen before. The bigger the frontier gets and the more nodes have been visited, the more effective a search using the bottom-up pass from the fewer remaining nodes becomes.

When parallelizing BFS, the direction optimizing variant with the bottom-up pass is less trivial as their common top-down counterpart as in general $\mathcal{O}(1)$ lookups need to be possible, and the inverted graph needs to be available.

3.5 Sparse Matrix Representation

A matrix that contains mainly zeros and, in important places values, is referred to as a sparse matrix. We here assume that all of our input graph problems represented as an adjacency matrix are sparse or can be represented as sparse matrices due to the fact that graphs, in reality, are often not fully connected but span a bigger sparse network. Examples of these can be online social networks like Facebook, where not all people are befriended with all other people in the network graph. A sparse matrix needs to be stored differently from a dense matrix where almost every field is populated with an important value. The techniques to represent sparse matrices follow compression methods that encode all non-zero values (nnz) and their coordinates of a matrix. We define sparsity as $nnz/dim1(A) * dim2(A)$

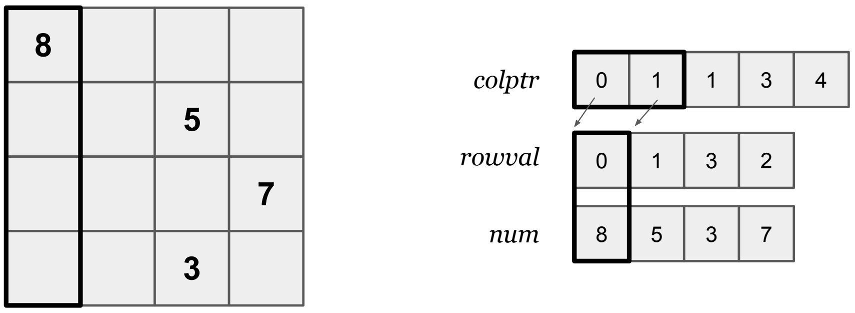

Figure 3.2: Compressed Sparse Column representation of the lefthand matrix though the `colptr`, `rowval`, and `num` array. The black boxes indicate the access of $A(0, 0)$, as seen the operation works columnwise. The `colptr` array bounds the range of values from position 0 to 1, which only contains the coordinate-value tuple 8 at row position 0.

Probably the simplest way to represent a sparse compressed matrix A is to encode it as triples of coordinates and their values. This format is commonly referred to as Coordinate Format (COO). However, the COO format is not efficient for lookups, $A(i, j)$, even if the coordinates are sorted, as the time to find an element would be $log(nnz)$ it is rather slow compared to directly accessing a dense matrix.

A more widely used data format is the compressed sparse row (CSR) or compressed sparse column (CSC) format [Gus72], which can be seen as an extension to the COO format but allows for better access times. We focus on CSC as CSR works analogously, CSC stores all values along the column direction.

CSC densely compresses the *nnz* values of each row and their position in two arrays, the *num* and the *rowval* array. Like in COO-format the `rowval` pointers indicate the row the value from `num` is residing in, both are of size *nnz*. The entry array, containing tuple sizes, pointers to the subarrays bounds in `colptr` of size $|V| + 1$, as the bounds are compactly stored head to tail. Accessing $A(i, j)$ is fast, as we can do a constant time lookup to get the range of the column containing all values by a lookup $colptr(i)$ to bounds $colptr(i+1)$, as seen in figure 3.2. Using the bounds, we can iterate through the `rowval` array looking for j. If j is not found the value of the field is zero. This results in a worst-case performance of $\mathcal{O}(|V|)$. However, for operations like BFS multiplication, we are interested in $A(i, :)$, we can iterate through the *rowval* bounds and immediately work on the data, making complexity to access useful data $\mathcal{O}(1)$.

Compared to pointer lists, CSC/CSR representations are more compact but can not be efficiently altered at runtime, thus should only work with static graphs or matrices.

Much research has been conducted on sparse formats other than COO and CSR/CSC representations. Some mentionable alternatives are (1) doubly compressed sparse columns (DCSC) which can compress highly sparse data better [BG08]. (2) Triply compressed sparse columns (TCSC), which improves upon DCSC [Mof+19]. (3) Block compressed sparse matrices (BSR) can adaptively exploit dense structures in matrices [VM05]. We did not continue to work further with these more advanced compression techniques due to missing hardware instructions, complexity, and the fact that CSC is at the current memory size the fastest while still compact. Using BSR would allow us to use IPU hardware to accelerate matrix operations. However, graphs do not, without costly preprocessing, emit easy and efficient to work with block structures.

3.6 Matrix partitioning

The performance is related to the balancing of computing resources over the whole operation space between different execution units, the better balanced compute resources get distributed, the better the performance gets, as utilization is maximized. Especially other domains that work with big dense and sparse matrices such as Partial Differential Element solvers use matrix/graph partitioners to create a matrix following performance maximizing properties. An often assumption is that the input matrix has a structural pattern. However, graph problems are not required to emit structural patterns. Therefore, we can assume to work with an unstructured matrix. Graph partitioners can optimize these unstructured matrices to certain degrees, but in comparison require a lot of preprocessing time. Wang et al.[Wan+16] have shown that using Metis [KK98], a graph partitioner, does not result in a big performance benefit. However, randomly permutating the matrix will in the expected case result in homogenous partitions. This is even true if the input was highly structural, which under our assumptions would lead to inhomogenous partitions under an even partition grid and therefore can be corrected.

4 BFS implementation on IPU

4.1 Sparse Data over Dense Communication

The IPU does not allow dynamically sized data transmissions, we always communicate a dense tensor. To allow for more efficient compute we define a sparse queue data structure on the dense tensors where a queue Q holds m values. Q is made from a scalar holding the allocated number of values and a pointer to the dense data structure containing the values. We need to allocate space in the dense tensor for the maximum amount of values that Q can hold in any possible scenario. We call the higher-level structures SpV respectively.

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.

4.2 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 $p_y \times p_x$ pattern. Thus, an adjacency matrix A is decomposed into p square partitions

$$A = \left(\begin{array}{c:c:c} A_{1,1} & \dots & A_{1,x} \\ \hdashline \vdots & \ddots & \vdots \\ \hdashline A_{y,1} & \dots & A_{y,x} \end{array}\right)$$

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 [Bul+17; Yoo+05], 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.

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

Algorithm 1: Topdown BFS algorithms, adopted from [Bul+17] and the linear algebraic version [BM11]

```
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
```

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 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 4.1. 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.

4.4 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

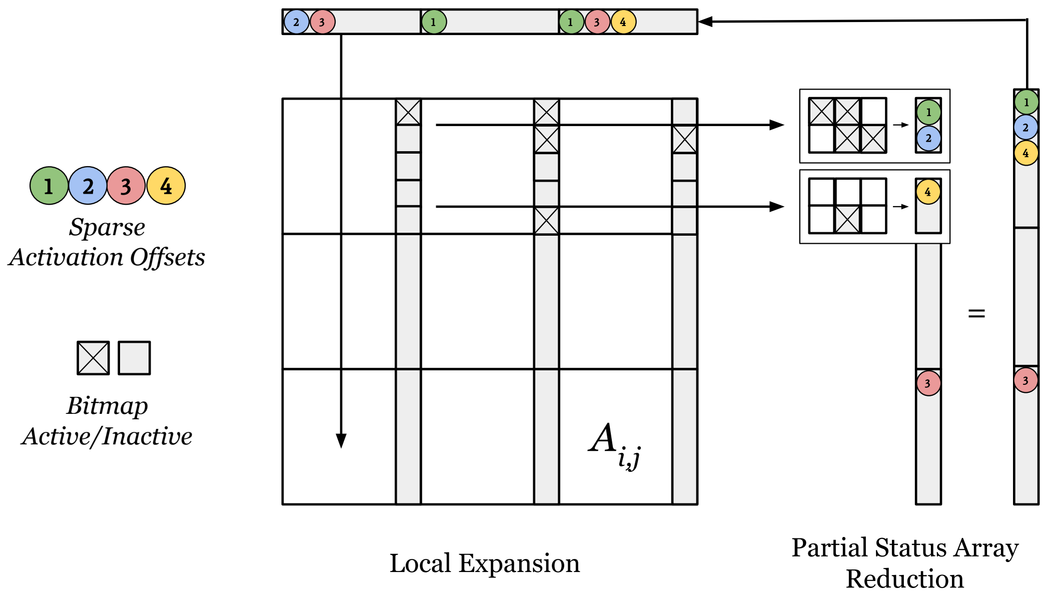

Figure 4.1: Layout of the 2D decomposition. We map each partition to a physical processor tile. Each tile also receives a copy of the sparse input frontier, along the first dimension, indicated with colored balls as activated vertices normalized to local offsets on each tile. In the reduction phase processor tiles receive the output status array of the local expansion and merge these into a new sparse frontier vector of the next BFS level.

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 a lack of memory.

4.5 Challenges of IPU Graph Implementations

4.5.1 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.

4.5.2 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 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, Grapcore's *popc* compiler has data-flow analysis [ASU86]. 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, and their space can be used by another live variable. For our program, the lifetime of variables connected to the expansion phase is related to the reduction phase and vice versa. Table 4.1 gives an overview of 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.

The variables from the expansion phase are required in the reduction phase and vice versa. Therefore, both variables need to allocate space in memory. As data is also moved within a phase, we use a multiplicator of two to factor in the double existence of data in the input field during a phase (Table 4.1. We define $p_x * p_y = p$, $n_p = n/p_x$.

Usage	**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 4.1: 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.

4.6 Optimizations

4.6.1 Removal of Isolated Vertices

The Kronecker graph generator used to generate the graphs for the Graph 500 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% [SPK13] 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.

4.6.2 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 iterations gets smaller the more vertices have already been flagged as found. Thus, when processing the first BFS level, this number is the 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.

4.6.3 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

```
__attribute__((target("supervisor"))) bool compute() {
  __asm__ volatile(
      "setzi   $m1, __runCodelet_WorkerEntry\n"
      "runall  $m1, $m0 , 0 \n"
      "sync    " STR(TEXCH_SYNCZONE_LOCAL) "\n");
  return true;
}
```

Listing 2: Vertex source code to launch worker contexts (threads), from a hadware supervisor context (root thread). We load the vertex base of the worker and start all six worker contexts, temporarily suspending the supervisor context until a worker returns. The sync instruction waits for all worker contexts to return back to the supervisor.

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.

In our first few implementation iterations, we used a hand-crafted version written in assembly. This had the advantage of using all six hardware execution contexts, offering full throughput. At the time of writing, the Poplar SDK does not allow for spawning and joining higher-level threads known from operating systems. Neither does the SDK give an API for managing execution contexts. Falling back to assembly was a convenient way of controlling the hardware. However, this resulted in long development times and difficulty fixing bugs, as Graphcore does not provide a proper debugger.

We worked around this issue by entering the code through the supervisor hardware context, which is capable of starting contexts with the same vertex layout as the worker and calling into the `bool compute()` function of the worker, emulating the supervisor setup that would usually take place. This was accomplished by using inline assembly in a supervisor vertex (Listing 2).

5 Experimental Setup

We have implemented our work 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 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 the 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 [LH15; YBO20], 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 connects 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 them in the input graph. We run the second kernel with all given keys and take the mean.

5.1 Test Instances

We use both Graph500 instances as well as graphs derived from SuiteSparse [Kol+19] matrices. The matrices were selected to match a published test set [YBO20] after removing all instances that are too large to run on the IPU. Table 5.1 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 $\mathbf{e} = 16$ 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 conforming 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.

5.2 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 [LH15] and **Gunrock** by Yangzihao Wang et al. [Che+20; Wan+16]. The Gunrock[1] and Enterprise[2] codes were both executed 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 [Mur+10] which relies on MPI, a sophisticated MPI/OpenMP implementation

[1]Git commit: 5ee3df5, Online: `https://github.com/gunrock/gunrock`
[2]Git commit: 426846f, Online: `https://github.com/iHeartGraph/Enterprise`

provided by Yasui et al. [YFG13] from Tokyo Institute of Technology (TITech), and the BFS implementation from the GAP benchmark suite [BAP15]. The latter has the advantage that it reads the Matrix Market format. We thus use it for comparison on SuiteSparse matrices outside of Graph500.

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[3][4] 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.

[3]https://en.wikichip.org/wiki/amd/epyc/7601
[4]https://en.wikichip.org/wiki/intel/xeon_gold/6130

SuiteSparse						
Name	**Diam**	**Vertices**	**Edges**	deg_{min}	deg_{max}	deg_{avg}
G43	4	1K	10K	7	36	19.98
coAuthorsDBLP	24	300K	978K	1	336	6.54
Journals	2	124	6K	19	124	97.32
coPapersDBLP	23	540K	15M	1	3299	56.41
loc-Gowalla	16	197K	950K	1	14730	9.67
ship_003	58	122K	4M	18	144	66.43
delaunay_n12	36	4K	12K	3	14	5.99
delaunay_n13	49	8K	25K	3	12	5.99
delaunay_n14	65	16K	49K	3	16	6.00
delaunay_n15	87	33K	98K	3	18	6.00
delaunay_n16	119	66K	197K	3	17	6.00
delaunay_n17	167	131K	393K	3	17	6.00
delaunay_n18	228	262K	786K	3	21	6.00
kron_g500-logn16	6	66K	2M	0	17998	74.96
kron_g500-logn17	6	118K	5M	0	29936	78.04
kron_g500-logn18	6	236K	11M	0	49163	80.74
kron_g500-logn19	7	432K	22M	0	80675	83.90

Generated						
Name	**Diam**	**Vertices**	**Edges**	deg_{min}	deg_{max}	deg_{avg}
kron19_16†	8	356K	8M	0	40329	29.53
kron19_16.2†	8	356K	8M	0	40389	29.53
kron19_16.3†	7	356K	8M	0	40326	29.53
kron18_16†	8	197K	4M	0	25336	29.04
kron17_16†	7	118K	2M	0	15759	28.45
kron16_16†	8	66K	1M	0	9763	27.76
kron15_16†	6	33K	524K	0	5925	26.95
kron19_48	7	432K	25M	0	78705	82.51
kron19_32	8	393K	16M	0	62468	56.71
kron18_128	6	236K	34M	0	77348	190.98
kron18_96	6	236K	25M	0	67737	148.63
kron18_64	7	236K	17M	0	55571	103.80
kron18_32	7	236K	8M	0	38505	55.35
kron16_32	7	66K	2M	0	14208	51.94
kron15_32	6	33K	1M	0	8604	49.84
kron17_32	7	118K	4M	0	23589	53.75

Table 5.1: 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.

6 Experimental Results

6.1 Performance Comparison Experiment

Our experimental results are collected in Figures 6.1 and 6.2. They show our performance 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. Our work shows the highest speedups for very small instances. 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×.

For the Suitesparse graphs, we observed 3× speedups for smaller and 1.5× speedups for the larger *DBLP* instances over Gunrock, which is the best alternative here. An exception is a larger and thus higher diameter *delaunay* graph which exhibits 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 the context of the scaling behavior of other BFS implementations. Results are shown in Figure

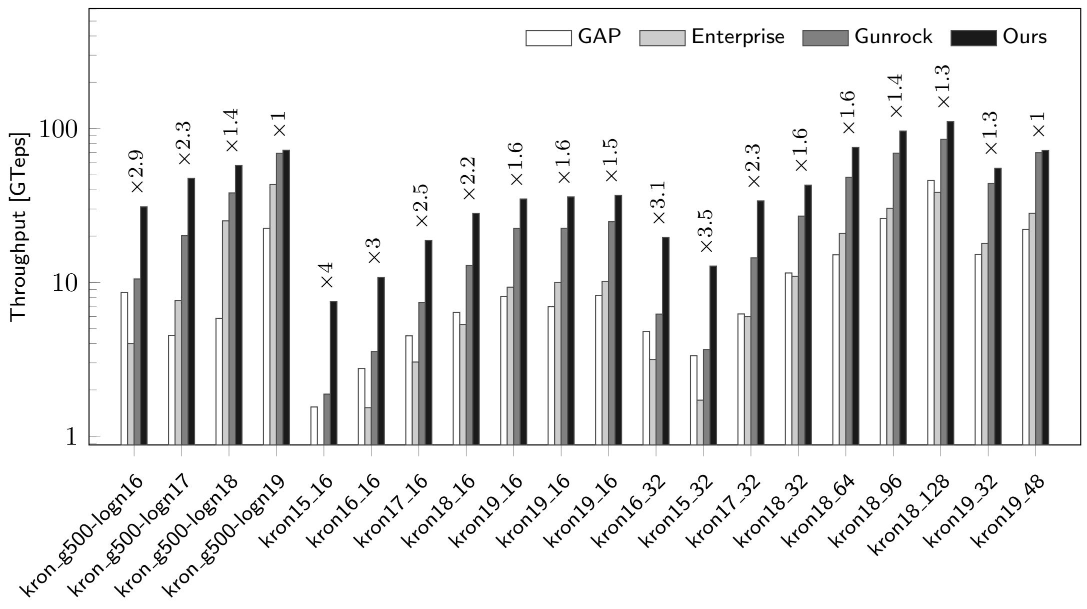

Figure 6.1: Performace of our code compared to CPU and GPU for the Kronecker graphs.

6.3. 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 lead over Enterprise. Furthermore, while ours starts with a large advantage at Scale 15, the gap closes to 1.5× at Scale 19. Thus, due to the limitation 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

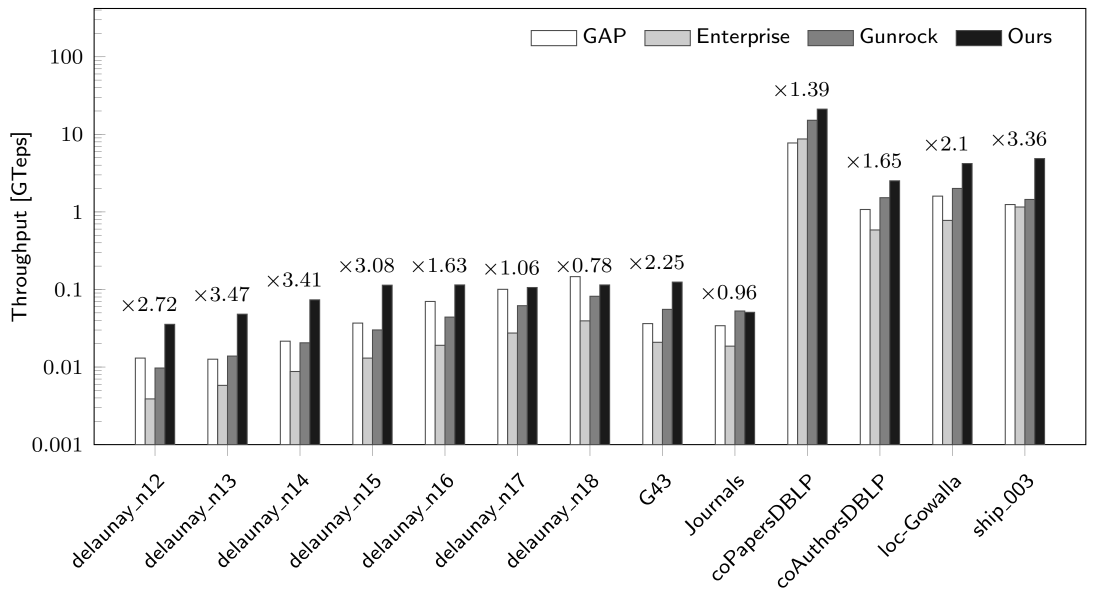

Figure 6.2: Performace of our code compared to CPU and GPU for the Suiteparse instances.

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 [Bea+11], sophisticated GPU codes explicitly cache the status of high degree vertices in shared memory during the backward search phase, as suggested for the Enterprise BFS code [LH15]. 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 that 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.

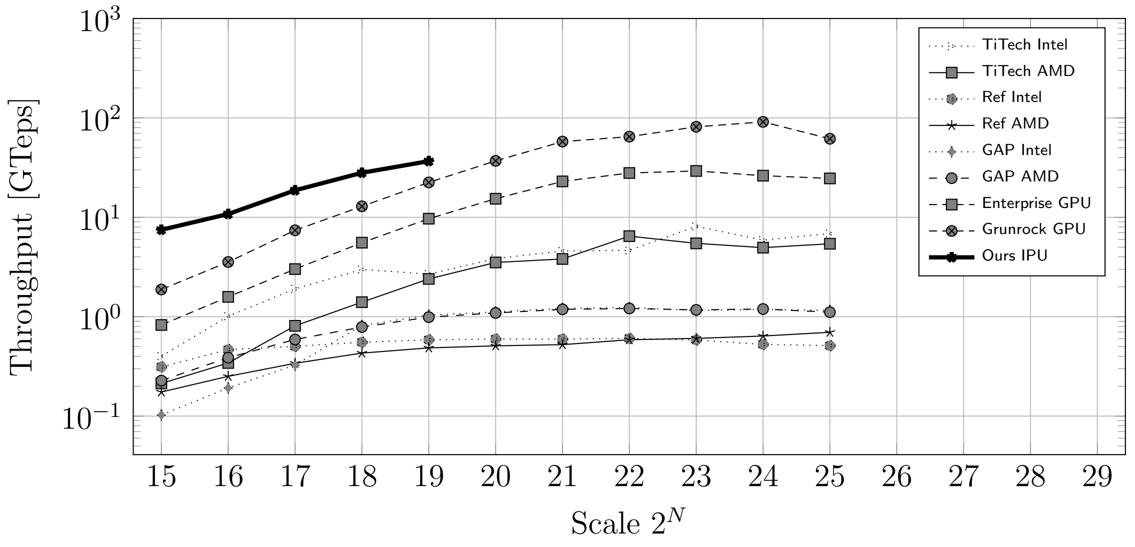

Figure 6.3: Performance of G500 Kron-N-16 graphs by scale on all tested codes and architectures.

6.3 Memory Usage

During our experiments and development, we also analyzed the total memory used per tile, as this is an essential part of programming the IPU. Small mistakes often resulted in a huge memory spike making compilation or running bigger instances impossible. Figure 6.4 and Figure 6.5 were generated through PopVision a tool dedicated to analyzing the IPU.

Figure 6.4 shows the biggest Kronecker graph that can fit on the IPU. Doubling the size would lead to overcommitted tiles, henceforth making it impossible to compile.

An insight from this is that data reduction by algorithms or using well aligned or compressed data types can result in a great benefit in runtime or instance size. During our development, we reduced the memory footprint by more than $2\times$ making it possible to fit a Kronecker-19-16 graph instead of only an Kronecker-17-16.

Furthermore, with more vertices, we saw a steep linear increase in occupied memory. This can be explained by the storage format (CSC/CSR) as the required

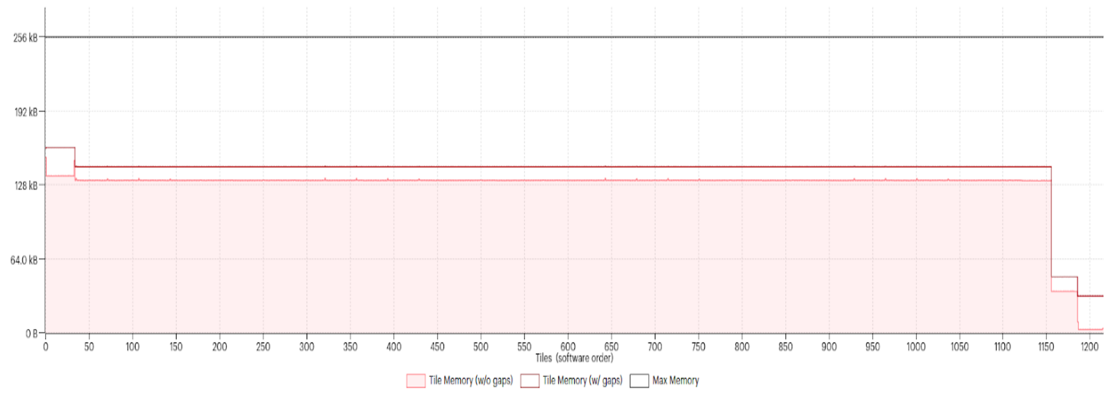

Figure 6.4: Maximum memory usage of the 1216 tiles on the IPU over the course of its execution for a Kron-19-16. The black line indicates the maximum available memory per tile.

space for storing the matrix is $2 * \sqrt{p} * |V|$ (Table 4.1), where most of the occupied space will be used to store the structure of the submatrix instead of the contained values.

6.4 GreenGraph500

With a peak performance of $38.58 Gteps$ on a Kronecker-19-16 input graph and a nominal power usage of $150W$, the IPU could be placed 10^{th} on the small graph input problem data leaderboards of November 2020. The theoretical performance per watts is equal to $2573 Mteps/W$. However, these results are theoretical as we do not adhere to the GreenGraph500 datacenter humidity, temperature, and power measurement specifications. Nonetheless, it can be assumed that the IPUs do not exceed their power limit constantly. Therefore leaving the possibility for a better efficiency rating.

6.5 Intra Code Performance Overview

PopVision allows visualizing the BSP phases per tile. We could observe that the majority of time is spent in computation, and the rest is similarly shared between synchronization and data exchange. While exchange servers the purpose of communication, synchronization is rather unwanted as it does represent idle tiles. Further, in the execution profile, the square nature of the algorithm is visible as the bottom tile row is idle during most of the execution time. Some Poplar of the shelf library function, which only run for a short time, are able to use 100% of the tiles and can be seen as small red lines spanning from the top to bottom of the profile.

The uneven spikes in the first iterations can be explained through the very sparse early activation and expansion of the BFS steps, as work initially is unevenly distributed, in the later execution phases (seen in the middle) most nodes get visited at once, creating an overall saturated and balanced computation phase.

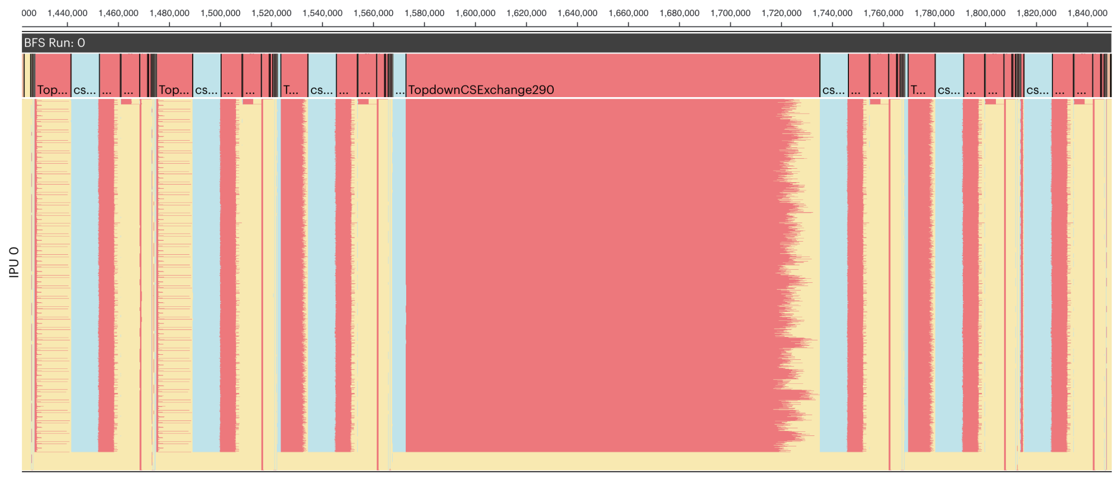

Figure 6.5: Execution profile for a Kron-19-16. Visualizes, how time and resources are spend. The red indicates computation, the blue data exchange, and the yellow synchronization phases.

7 Discussion

We have tested our BFS code on the IPU and achieved speedups between 0.96× and 4× over the fastest GPU and CPU codes, 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, the Kronecker graph generator used for the G500 benchmark, for bigger graph instances will result in a higher deg_{avg} possibly allowing the next generation of IPUs to get a performance increase by solving bigger instances in memory.

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 [Mur+10] performance results increased massively, but the 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 a 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 [HP19], which points out that programmer controlled scratchpad memory offers significant performance advantages compared to transparent general-purpose caches. In the case of the IPU, no additional programming complexity is incurred by this, since the memory hierarchy only has a single level.

During our development, we have observed that implementing irregular algorithms is challenging due to the static compute graph and the compile-time required communication patterns and transmission sizes. However linear algebra, even with the addition of sparsity, maps well to the concept of predefined communication flows, we have shown this in this work. However, algorithm designers and algorithm engineers will have to work around these constraints or reformulate algorithms which makes general-purpose programming or porting algorithms efficiently challenging.

Further, as all programs follow a bulk synchronous parallel pattern we need to take great care of unbalanced operations within a superstep as strugglers are stopping all other cores from continuing computing. Henceforth, we have also observed that interleaving inhomogeneous Verticies needs to be done carefully as the performance can drop because of a single long-running Vertex. Splitting long-running operations into smaller operations and distributing them throughout different supersteps is tedious but can improve the overall performance.

We found the hardware architecture adequately to understand. However, the compiler sometimes caused us performance-related problems i.e misaligned memory slicing of tensors can cause double alignment copy buffers, which created a 2x memory and compute footprint. Therefore, we heavily relied on the PopVision tool from Graphcore, which allows us to visually inspect the memory regions to uncover compiler behavior, hardware aspects, and exchange and data movement. Furthermore, the PopVision tool was used to visualize unbalanced computations and find bottlenecks in the sparse matrix computation and memory layout.

Graphcore intends to use threads for non-communicating context parallelization. We are working around the intended use-case by spawning worker contexts from the supervisor like one would create threads. We are using prefix sums, common from GPU algorithms, to dynamically distribute work between workers. However, the IPU GC2 is missing crucial instructions such as an integer divide or modulo to even more efficiently implement work splitting routines. The missing integer instructions are also limiting us in algorithmic design choices. As atomics are not implemented it is at the time of writing impossible to coordinate asynchronous memory operations. Atomic memory operations will allow for replacing the status arrays with dense bitmaps reducing the memory usage.

Most time developing the project was spend on preparing data for the IPU, and generating the compute graph. Only 150 of the 2000 lines of code are written to run on the IPU, the remaining lines are doing setup work. The IPU can profit from a language that allows for rapid prototyping and higher-level abstractions as the host code is often not performance-relevant.

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 [AK+17; Kay+20] 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.

9 Future Work

Graphic processing units (GPU) were developed for accelerating computer graphics, as general-purpose CPUs could not handle the amount of data and computations for efficiently producing good enough latency or resolution for the human eye. Later, GPUs were found to be an excellent addition to High-Performance Computation (HPC) clusters, as they allowed for highly predictable data-parallel computations such as dense linear algebra. Many algorithms outside of the graphical domain have been found to get accelerated through the use of GPUs such as machine learning, scientific computing, and accelerating databases. Every problem can be solved on CPUs, but some are up to 100x more effective to solve on GPUs, hence the wide adoption of GPUs in HPC clusters.

New accelerators for artificial intelligence, machine learning, and other domains are getting developed for speeding up problem-solving in respective fields, as GPUs are too general in their design or not applicable to specific algorithms. These accelerators all build on different assumptions and tradeoffs, which sets them apart from GPUs and CPUs.

We want to research hardware accelerators for their non-domain-specific use case to explore new possibilities to solve different problems in computer science efficiently, and other related fields. At the time of writing, these accelerators only serve a single specific purpose but researching their capabilities beyond the intended use-case can allow for a similar development to the use of GPUs, in the long term, accelerating a wide variety of HPC applications. The gap of hardware availability compared to their usage is caused by, among others, insufficient programmability, the need for specially-designed or adapted computational algorithms, and the lack of a quantifiable understanding of the realistically achievable performance. All these aspects require research effort, which will find the broad interest of HPC operators and enterprises that already see value in such hardware.

Some of these hardware accelerators have the potential of being used for other purposes as they build on different assumptions than the previous hardware generations. Most of the accelerators favor a shared nothing memory model instead of a global memory space, making it possible to fit orders of magnitude more cores onto a single chip than on a CPU. Cererbas Wafer Scale Engine (WSE), for example, has 850.000 cores and 40GB on-chip memory divided into 43KB chunks for each processor. All of these cores together have a memory bisection bandwidth of 20PB/s and an event-based mesh interconnect of 220PB/s in a single chip. The novel design and interconnect architecture opens up new possibilities, especially for irregular data-based due to the event-based design. Another accelerator, the Graphcores IPU, offers over 1472 cores with six threads and 900MB on-chip memory, again split between all cores. However, the communication capabilities allow for an all-to-all interconnect with 8TB/s in a Bulk Synchronous Parallel (BSP) manner. This allows for independent calculations and fast exchange of sub results throughout the whole chip, making the architecture promising for many application classes.

Future projects will improve the interface to the current programming model. C++ requires a constant recompilation of the project, and memory management is a tedious part of it. The pure efficiency promises of the language are great for using C++ for often executed applications which can take time to write when knowledge about features is known from the beginning. However, research is often changing requirements and is testing a lot of possibilities before reaching a stable point for publishing. Often these research projects are more written than used. Through interfacing with Julia, a Python-alike, on-the-fly compiled language, we hope to bring more expressiveness to researchers and first-time users. The Julia interface is more concerned with building compute graphs and preparing data, rather than generating the high-performance vertices as those are; out of our experience, the most performance-critical parts were not the ones that took the most effort to write. Through this Julia interface, we want to combine the performance of the IPU with the broad and growing scientific ecosystem of Julia.

Another field where GPUs are not highly more performant than CPUs is in bioinformatics, in the field of genome sequencing. The best CPU implementation can only achieve 25% of the GPU performance on a single thread, yet as multi-core/multithreaded processors are common, the GPU has no big advantage over a general CPU. The reasoning behind this is that the highly data-parallel the architecture of the GPU does not fit the problem domain well. Therefore, this indicates the potential for an IPU implementation to overtake GPU and CPU performance if a large number of irregular working tiles can be used.

We

Literature

[AB16] A. Azad and A. Buluç. „Distributed-memory algorithms for maximum cardinality matching in bipartite graphs“. In: *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2016, pp. 32–42 (cit. on p. 15).

[Aba+16] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. „Tensorflow: A system for large-scale machine learning“. In: *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 2016, pp. 265–283 (cit. on p. 8).

[AK+17] F. N. Abu-Khzam, R. L. Collins, M. R. Fellows, M. A. Langston, W. H. Suters, and C. T. Symons. „Kernelization algorithms for the vertex cover problem“. In: (2017) (cit. on p. 49).

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, principles, techniques, and tools*. Addison-Wesley Pub. Co, 1986 (cit. on p. 28).

[BAP12] S. Beamer, K. Asanovic, and D. Patterson. „Direction-optimizing breadth-first search“. In: *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE. 2012, pp. 1–10 (cit. on p. 19).

[BAP15] S. Beamer, K. Asanović, and D. Patterson. „The GAP benchmark suite“. In: *arXiv preprint arXiv:1508.03619* (2015) (cit. on p. 36).

[Bea+11] S. Beamer, K. Asanovic, D. Patterson, S. Beamer, and D. Patterson. „Searching for a parent instead of fighting over children: A fast breadth-first search implementation for graph500“. In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2011-117* (2011) (cit. on pp. 15, 41).

[BG08] A. Buluc and J. R. Gilbert. „On the representation and multiplication of hypersparse matrices“. In: *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE. 2008, pp. 1–11 (cit. on p. 21).

[BG11] A. Buluç and J. R. Gilbert. „The Combinatorial BLAS: Design, implementation, and applications“. In: *The International Journal of High Performance Computing Applications* 25.4 (2011), pp. 496–509 (cit. on p. 16).

[BM06] D. A. Bader and K. Madduri. „Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2“. In: *2006 International Conference on Parallel Processing (ICPP'06)*. IEEE. 2006, pp. 523–530 (cit. on p. 15).

[BM11] A. Buluç and K. Madduri. „Parallel breadth-first search on distributed memory systems“. In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 2011, pp. 1–12 (cit. on pp. 15, 25).

[Bul+17] A. Buluç, S. Beamer, K. Madduri, K. Asanovic, and D. Patterson. „Distributed-memory breadth-first search on massive graphs“. In: *arXiv preprint arXiv:1705.04590* (2017) (cit. on pp. 24, 25).

[Che+20] Z. Chenglong, C. Huawei, W. Guobo, H. Qinfen, Z. Yang, Y. Xiaochun, and F. Dongrui. „Efficient Optimization of Graph Computing on High-Throughput Computer“. In: *Journal of Computer Research and Development* 57.6 (2020), p. 1152 (cit. on pp. 15, 35).

[CP14] F. Checconi and F. Petrini. „Traversing trillions of edges in real time: Graph exploration on large-scale parallel machines“. In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE. 2014, pp. 425–434 (cit. on p. 15).

[CZF04] D. Chakrabarti, Y. Zhan, and C. Faloutsos. „R-MAT: A recursive model for graph mining“. In: *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM. 2004, pp. 442–446 (cit. on p. 3).

[DG08] J. Dean and S. Ghemawat. „MapReduce: simplified data processing on large clusters“. In: *Communications of the ACM* 51.1 (2008), pp. 107–113 (cit. on p. 18).

[DQR12] J. Dittrich and J.-A. Quiané-Ruiz. „Efficient big data processing in Hadoop MapReduce“. In: *Proceedings of the VLDB Endowment* 5.12 (2012), pp. 2014–2015 (cit. on p. 18).

[Gai+19] A. Gaihre, Z. Wu, F. Yao, and H. Liu. „Xbfs: exploring runtime optimizations for breadth-first search on gpus“. In: *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*. 2019, pp. 121–131 (cit. on p. 15).

[GB84] R. K. Ghosh and G. Bhattacharjee. „Parallel breadth-first search algorithms for trees and graphs“. In: *International Journal of Computer Mathematics* 15.1-4 (1984), pp. 255–268 (cit. on p. 15).

[GL05] D. Gregor and A. Lumsdaine. „Lifting sequential graph algorithms for distributed-memory parallel computation“. In: *ACM SIGPLAN Notices* 40.10 (2005), pp. 423–437 (cit. on p. 15).

[Gus72] F. G. Gustavson. „Some basic techniques for solving sparse systems of linear equations“. In: *Sparse matrices and their applications*. Springer, 1972, pp. 41–52 (cit. on p. 20).

[Hil+98] J. M. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling. „BSPlib: The BSP programming library“. In: *Parallel Computing* 24.14 (1998), pp. 1947–1980 (cit. on p. 18).

[HN07] P. Harish and P. J. Narayanan. „Accelerating large graph algorithms on the GPU using CUDA“. In: *International conference on high-performance computing*. Springer. 2007, pp. 197–208 (cit. on p. 15).

[HOO11] S. Hong, T. Oguntebi, and K. Olukotun. „Efficient parallel graph exploration on multi-core CPU and GPU“. In: *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE. 2011, pp. 78–88 (cit. on p. 15).

[HP19] J. L. Hennessy and D. A. Patterson. „A new golden age for computer architecture“. In: *Communications of the ACM* 62.2 (2019), pp. 48–60 (cit. on p. 46).

[Jia+19] Z. Jia, B. Tillman, M. Maggioni, and D. P. Scarpazza. „Dissecting the graphcore ipu architecture via microbenchmarking“. In: *arXiv preprint arXiv:1912.03413* (2019) (cit. on p. 8).

[Kay+20] K. Kaya, J. Langguth, I. Panagiotas, and B. Uçar. „Karp-Sipser based kernels for bipartite graph matching“. In: *2020 Proceedings of the Twenty-Second Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM. 2020, pp. 134–145 (cit. on p. 49).

[KK98] G. Karypis and V. Kumar. „A fast and high quality multilevel scheme for partitioning irregular graphs“. In: *SIAM Journal on scientific Computing* 20.1 (1998), pp. 359–392 (cit. on p. 22).

[Kol+19] S. P. Kolodziej, M. Aznaveh, M. Bullock, J. David, T. A. Davis, M. Henderson, Y. Hu, and R. Sandstrom. „The Suitesparse matrix collection website interface“. In: *Journal of Open Source Software* 4.35 (2019), p. 1244 (cit. on pp. 3, 34).

[KS05] R. E. Korf and P. Schultze. „Large-scale parallel breadth-first search“. In: *AAAI*. Vol. 5. 2005, pp. 1380–1385 (cit. on p. 15).

[Lan+14] J. Langguth, A. Azad, M. Halappanavar, and F. Manne. „On parallel push–relabel based algorithms for bipartite maximum matching“. In: *Parallel Computing* 40.7 (2014), pp. 289–308 (cit. on p. 15).

[LCS18] J. Langguth, X. Cai, and M. Sourouri. „Memory bandwidth contention: Communication vs computation tradeoffs in supercomputers with multicore architectures“. In: *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE. 2018, pp. 497–506 (cit. on p. 9).

[LH15] H. Liu and H. H. Huang. „Enterprise: breadth-first graph traversal on GPUs“. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2015, pp. 1–12 (cit. on pp. 15, 33, 35, 41).

[LPM11] J. Langguth, M. M. A. Patwary, and F. Manne. „Parallel algorithms for bipartite matching problems on distributed memory computers“. In: *Parallel Computing* 37.12 (2011), pp. 820–845 (cit. on p. 15).

[Mal+10] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. „Pregel: a system for large-scale graph processing“. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 2010, pp. 135–146 (cit. on p. 18).

[Mof+19] M. H. Mofrad, R. Melhem, Y. Ahmad, and M. Hammoud. „Efficient distributed graph analytics using triply compressed sparse format“. In: *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE. 2019, pp. 1–11 (cit. on p. 21).

[Mur+10] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang. „Introducing the graph 500“. In: *Cray Users Group (CUG)* 19 (2010), pp. 45–74 (cit. on pp. 3, 15, 35, 45).

[Roc+20] K. Rocki, D. V. Essendelft, I. Sharapov, R. Schreiber, M. Morrison, V. Kibardin, A. Portnoy, J. F. Dietiker, M. Syamlal, and M. James. *Fast Stencil-Code Computation on a Wafer-Scale Processor.* 2020. arXiv: `2010.03660 [cs.DC]` (cit. on p. 2).

[SPK13] C. Seshadhri, A. Pinar, and T. G. Kolda. „An in-depth analysis of stochastic Kronecker graphs“. In: *Journal of the ACM (JACM)* 60.2 (2013), pp. 1–32 (cit. on p. 29).

[Val90] L. G. Valiant. „A bridging model for parallel computation“. In: *Communications of the ACM* 33.8 (1990), pp. 103–111 (cit. on pp. 9, 17).

[VM05] R. W. Vuduc and H.-J. Moon. „Fast sparse matrix-vector multiplication by exploiting variable block structure“. In: *International Conference on High Performance Computing and Communications.* Springer. 2005, pp. 807–816 (cit. on p. 21).

[Wan+16] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. „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.* 2016, pp. 1–12 (cit. on pp. 15, 22, 35).

[YBO20] C. Yang, A. Buluc, and J. D. Owens. *GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU.* 2020 (cit. on pp. 3, 15, 33, 34).

[YFG13] Y. Yasui, K. Fujisawa, and K. Goto. „NUMA-optimized parallel breadth-first search on multicore single-node system“. In: *2013 IEEE International Conference on Big Data.* IEEE. 2013, pp. 394–402 (cit. on pp. 15, 36).

[Yoo+05] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek. „A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L“. In: *SC '05: Proceedings of the 2005*

ACM/IEEE Conference on Supercomputing. 2005, pp. 25–25 (cit. on pp. 15, 24).