UNIVERSITY
OF OSLO

Luk Bjarne Burchard

Repurposing Domain-specific Hardware Accelerators for Sparse and Irregular High-Performance General-Purpose Computation

Thesis submitted for the degree of Philosophiae Doctor

Department of Informatics
The Faculty of Mathematics and Natural Sciences

High Performance Computing Department
Simula Research Laboratory

2023

Series of dissertations submitted to the
Faculty of Mathematics and Natural Sciences, University of Oslo
No. 2695

ISSN 1501-7710

Cover: UiO.
Print production: Graphic center, University of Oslo.

Abstract

In recent years deep learning workloads have driven the demand for conventional hardware accelerators like GPUs and the development of novel hardware accelerators. While GPUs have also greatly improved HPC with their data-parallel computation capabilities, there is a need for specialized accelerators that can solve specialized problems like deep learning, graph processing, or bioinformatics more effectively. We expect more novel hardware accelerators to be present in future HPC clusters. To better prepare the scientific community for embracing such non-conventional hardware, we are interested in assessing their capabilities for other HPC workloads than deep learning.

This thesis focuses on the Graphcore IPU architecture, a novel SRAM-based architecture that integrates massive amounts of SRAM into the processing chip. The IPU offers a significant increase in cores and SRAM memory compared to state-of-the-art CPUs, promising performance benefits. However, the architecture uses a distributed memory model and bulk synchronous style programming, non-traditional choices for an on-chip processor. Therefore, algorithms will need to be reformulated to function on the IPU.

Currently, the IPU has not been widely used, let alone analyzed for workloads beyond deep learning. We address this gap in the literature by targeting HPC applications beyond deep learning, examining factors such as programmability limitations, algorithm requirements, and the predictability of achievable performance. The research aims to enhance understanding of the practical performance achievable with these processors and their capacity to accelerate a wide range of HPC applications. Specifically, we explore the IPU's capabilities for challenging sparse and irregular problems from the fields of bioinformatics, graph processing, and scientific computation.

We find that, in these fields, the IPU can offer significant performance increases over CPUs and GPUs. We further provide insights into the programmability and obstacles that need to be overcome when applying the IPU to irregular algorithms. We conclude our work by providing future suggestions of how the hardware can be improved to serve the provided use cases better.

Oppsummering

De siste årene har arbeidsmengden innen dyp læring drevet frem etterspørselen etter konvensjonelle maskinvareakseleratorer som GPU-er og utviklingen av nye maskinvareakseleratorer. Selv om GPU-er også har forbedret HPC betraktelig med sine dataparallelle beregningsmuligheter, er det behov for spesialiserte akseleratorer som kan løse spesialiserte problemer slik som dyp læring, grafbehandling eller bioinformatikk mer effektivt. Vi forventer at det vil komme flere nye maskinvareakseleratorer i fremtidige HPC-klynger. For å forberede det vitenskapelige miljøet bedre på å ta i bruk slik ukonvensjonell maskinvare, er vi interessert i å vurdere deres egenskaper for andre HPC-arbeidsbelastninger enn dyp læring.

Denne avhandlingen fokuserer på Graphcore IPU-arkitekturen, en ny SRAM-basert arkitektur som integrerer store mengder SRAM i prosessbrikken. IPU-en tilbyr en betydelig økning i antall kjerner og SRAM-minne sammenlignet med moderne CPU-er, noe som lover ytelsesfordeler. Arkitekturen bruker imidlertid en distribuert minnemodell og synkron masseprogrammering, noe som er utradisjonelt for en on-chip-prosessor. Algoritmer må derfor omformuleres for å fungere på IPU-en.

Foreløpig har IPU ikke vært mye brukt, for å ikke nevne analysert for arbeidsmengder utover dyp læring. Vi tar tak i dette gapet i litteraturen ved å fokusere på HPC-applikasjoner utover dyp læring, og undersøker faktorer som programmeringsbegrensninger, algoritmekrav og forutsigbarheten av oppnåelig ytelse. Forskningen tar sikte på å øke forståelsen av den praktiske ytelsen som kan oppnås med disse prosessorene og deres evne til å akselerere et bredt spekter av HPC-applikasjoner. Mer spesifikt undersøker vi IPU-enes evne til å løse utfordrende, glissene og uregelmessige problemer innen bioinformatikk, grafprosessering og vitenskapelige beregninger.

Vi finner at IPU kan gi betydelige ytelsesøkninger i forhold til CPU-er og GPU-er på disse områdene. Vi gir også et innblikk i programmerbarheten og hindringene som må overvinnes når IPU brukes på irregulære algoritmer. Vi avslutter arbeidet vårt med å komme med forslag til hvordan maskinvaren kan forbedres for bedre å kunne brukes til de aktuelle bruksområdene.

Preface

This thesis is submitted in partial fulfillment of the requirements for the degree of *Philosophiae Doctor* at the University of Oslo. The research presented here was conducted at the University of Oslo and at Simula Research Laboratory in the HPC Department, under the supervision of Main Supervisor Prof. Xing Cai and associate professor Co-Supervisor Johannes Langguth and Prof. Are Magnus Bruaset.

This work was partially supported by the European High-Performance Computing Joint Undertaking under grant agreement No. 956213 and the Simula Research Laboratory Strategic HPC initiative.

The thesis is a collection of four papers presented in chronological order of writing. The common theme to them is the exploration of a novel hardware accelerator architecture for HPC applications. The papers are preceded by an introductory chapter that relates them to each other and provides background information and motivation for the work. The first article is a joint work with Xing Cai and Johannes Langguth. The first article is a joint work with Kristian Gregorius Hustad, Johannes Langguth, and Xing Cai. The first article is a joint work with Max Xiaohang Zhao, Daniel Thilo Schroeder, Johannes Langguth, and Xing Cai. The first article is a joint work with Max Xiaohang Zhao, Johannes Langguth, Aydın Buluç, and Giulia Guidi.

Acknowledgements

This thesis resulted from working countless hours in the offices of the Simula Research Laboratory with collaborators. My sincere gratitude goes to Johannes Langguth and Xing Cai, who were always available during those many hours to review my research and provide insightful feedback despite late hours and tight deadlines. My special thanks go to Johannes, whom I have known for almost four years at the time of this writing. Thank you for revising and taking the time to analyze the first publications, shaping my path as an early-stage researcher, talking and chatting about topics outside of work, and sharing your insights into the many fields of computer science and beyond. This thesis would not have been possible without your help. Also, I would like to thank Xing Cai. Thank you for your patience, constant feedback, and dedication to high-quality research, letting me work freely on topics I enjoy. I know this style of supervision can only sometimes be found, and I am incredibly grateful for having the luck of being one of your students.

Furthermore, I want to thank Giulia Guidi and Aydın Buluç, whom I collaborated with. Thank you for trusting me with your time and expertise and offering me great advice and opportunities.

I want to thank all my colleagues in the Simula HPC department, especially Daniel Thilo Schroeder, who, over the years, has become a good friend. Thank you for believing in me before no one else did and opening up many possibilities for which I am forever grateful. I would also like to thank Tore. H. Larsen, who is the administrator of the eX3 supercomputing infrastructure, said that without your incredible knowledge and help, things would not work.

Thanks to my family and many supportive friends, particularly Kai Misselwitz, Iver Håkonsen, Friedrich

My grandfather working on the SUR-100 (“Siemens-Unterrichtsreaktor”, a teaching reactor)[1]

[1]The photos were created and provided by Prof. W. Kaspar-Sickermann

Rieber, Laszlo Dajka, and Hichem Dhouib. Thank you for believing in me in times of doubt and giving me memorable life advice.

I want to express my heartfelt gratitude to Carsten Schubert and Max Zhao. You shaped how we studied and always pushed me forward. Your invaluable help throughout our university journey has been pivotal in enabling me to write this thesis. Thank you sincerely for your unwavering support.

Finally, I would like to contribute my gratitude to my grandfather, Manfred Storz, who has always been my guiding figure. I remember when I was a small child, still in Kindergarten, I would spend many weekends with my grandparents. He would take me to his laboratory and teach me how to solder and build circuits, and later, when I told him we learned division, he excitedly taught me Ohm's law. He was always curious and wanted to know what I had learned in school, university, and my PhD; he would always chime in and excitedly explain connections to electrical engineering problems. I wish we had more time.

Luk Bjarne Burchard
Oslo, July 2023

List of Papers

Paper I

Luk Burchard, Xing Cai, Johannes Langguth "iPUG for Multiple Graphcore IPUs: Optimizing Performance and Scalability of Parallel Breadth-First Search". In: *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. Vol. 28, (2021, December), pp. 162–171. DOI: 10.1109/HiPC53243.2021.00030.

Paper II

Luk Burchard, Kristian Gregorius Hustad, Johannes Langguth, Xing Cai "Enabling Unstructured-Mesh Computation on Massively Tiled AI-Processors: An Example of Accelerating In-Silico Cardiac Simulation". In: *Frontiers in Physics*. Vol. 11, (2023, March), pp. 105. DOI: 10.3389/fphy.2023.979699.

Paper III

Max Xiaohang Zhao[†], Luk Burchard[†], Daniel Thilo Schroeder, Johannes Langguth, Xing Cai "iPuma: High-throughput Sequence Alignment for MIMD AI Accelerators". Submitted to *ISC High Performance 2024*.

Paper IV

Luk Burchard[†], Max Xiaohang Zhao[†], Johannes Langguth, Aydın Buluç, Giulia Guidi "Space Efficient Sequence Alignment for SRAM-Based Computing: X-Drop on the Graphcore IPU". Accepted for publication in *SC '23: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. DOI: doi.org/10.1145/3581784.3607094

[†]Shared first author, both authors contributed equally.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The development of graphics processing units (GPUs) has revolutionized the field of High-Performance Computation (HPC) [Fan+04; Göd+07; Kin+09; PSS08; TK06], allowing to speed up data-parallel computations and accelerating a wide variety of algorithms beyond the domain of computer graphics, including machine learning, scientific computing, and databases [Bre+14; Hus19; KSH12]. While GPUs have been widely adopted in HPC clusters, there is a growing need for specialized accelerators [Ang+23; SKK17] for artificial intelligence, machine learning, and other domains that can, through specialization, solve specific problems more effectively than GPUs or central processing units (CPUs). Due to the accumulation of hardware resources and the need to accelerate deep learning workloads more, there is an AI/ML HPC convergence. Thus, more of this specialized hardware will be available in HPC clusters [Ang+23; Hue+20].

In this PhD dissertation, we aim to explore the potential of a novel domain-specific hardware accelerator for its non-domain-specific use cases and investigate new possibilities for solving different problems in computer science and related fields effectively. While these accelerators currently serve a single specific purpose, researching their capabilities beyond their intended use case can pave the way for a similar development to that of GPUs, ultimately accelerating a wide variety of HPC applications in the future. Furthermore, through the AI/ML HPC convergence, we want to provide alternative use cases for novel hardware accelerators that will be more present in future HPC systems. We, in this dissertation will provide the first use cases and analysis from different areas in the field of high-performance computing. Currently, there is a gap between the availability of the hardware and the publications targeting HPC applications other than deep learning. The gap between hardware availability and usage is caused by various factors, including insufficient programmability, the need for specially-designed or adapted algorithms, and the lack of understanding of practically achievable performance.

Most architectures used nowadays are Von Neumann architectures, separating the processing unit and memory. In this architecture, data and instructions are stored in the same memory, and the processor has to fetch data from this external memory. This separation leads to the Von Neumann bottleneck, also referred to as the memory wall, a fundamental limitation in traditional computer architecture, applying to both GPUs and CPUs. This fetching creates a performance bottleneck as the processor often needs to wait for the data to be transferred between the processor and memory before it can proceed with the execution. This constant movement of data between the processor and memory significantly slows down the overall computation speed and efficiency.

The memory wall has motivated the exploration of modern SRAM computing

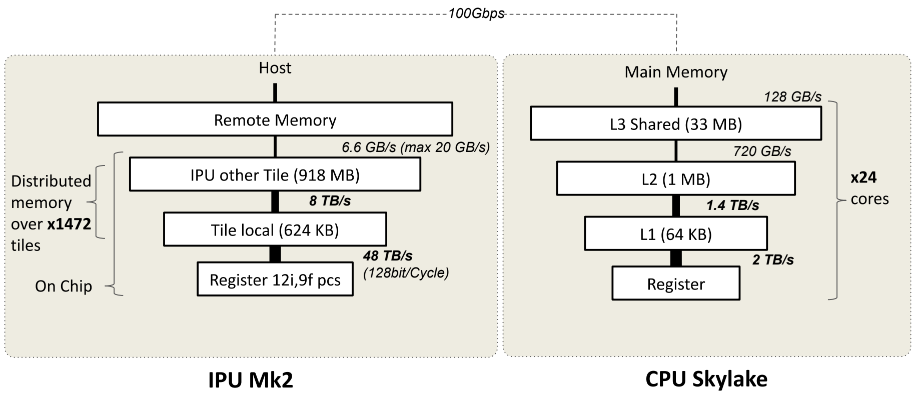

Figure 1.1: Memory hierarchy of the IPU Mk2 system compared to a CPU. The CPU is modeled after an Intel Skylake processor [HVH18].

and processing in memory (PIM) architectures. These emerging architectures aim to address the performance limitations imposed by the memory wall by integrating processing capabilities directly into the memory subsystem. PIM architectures take a first step further by incorporating more advanced processing elements, such as logic gates or specialized accelerators, directly within the (DRAM) memory subsystem [Mut+23; Seb+20]. Alternatively, SRAM-based architectures increase the SRAM size on the computing silicon and use it as memory instead of cache [Mit+21]. Thus operations can be executed with reduced latency and higher bandwidth, as the data resides in close proximity to the computational units.

There is a growing number of novel hardware accelerators for deep learning commercially available, such as Cerebras WSE [Lau21], SambaNova [Ema+21], Graphcore IPU [Kno21], Groq [Abt+22], Tenstorrent [Gwe20], and many more [Reu+20]. We choose the Graphcore IPU as our architecture of interest, as it was one of the first commercially available SRAM-based devices, providing high memory bandwidth and many independent cores, promising to eschew the memory wall. The IPU offers approximately 30× as many cores as comparable CPUs, with a higher bandwidth addressing more than an order of magnitude more SRAM memory. The IPU is a MIMD architecture where all cores can do independent instruction fetch and execution, breaking from often found SIMD patterns. This makes the architecture promising for many classes of applications. The IPU is a new hardware architecture that has not been broadly explored in previous literature for HPC algorithms other than deep learning problems. We intend to provide the missing pieces in the literature to explore the capabilities of the IPU for different HPC applications. We are focusing on sparse and irregular problems, as these are not well solved using the GPU, which currently is the predominant accelerator architecture in HPC.

The IPU forgoes the implementation of a last-level cache (LLC) due to the large SRAM requirement on the chip, which would cause large latencies. The

IPU has no memory hierarchy, such as caches or IPUs prioritize data movement and parallel processing, utilizing on-chip memory structures optimized for high-bandwidth exchanges instead. Therefore, no cache hierarchies are implemented, and all cache is subdivided akin to a distributed memory system. The omission of an LLC is driven by the need for massive parallelism, specialized memory access patterns, and the potential for contention and bottlenecks in data access. Therefore, the IPU is from its communication and access hierarchy layout quite different from a CPU or GPU (Figure 1.1), requiring different algorithmic approaches.

We seek to explore the capabilities and limitations of these specialized hardware accelerators beyond their intended use cases. Through this research, we aim to provide a better understanding of the practically achievable performance of these accelerators and their potential to accelerate a variety of HPC applications.

1.1 Outline

In Section 1.2 we provide background for the attached papers and provide a broader context of the field. In Section 1.3 we provide our main research questions guiding the creation of the papers. In Section 1.4 we summarize our four research papers and provide background, and motivation, and describe the impact of work. In Section 1.5 we finalize the introduction with a conclusion of our work and propose changes to the IPU.

1.2 Background

In this section we are providing background on the subjects studied in our articles.

1.2.1 The Intelligence Processing Unit (IPU)

The Intelligence Processing Unit (IPU), was built first and foremost to speed up deep learning tasks. This is reflected in the hardware and software design throughout the IPU. There have been three different IPU generations that were available at the time of writing. We note that the tile instruction set architecture (ISA) [AMH23], i.e. the available instructions available to interact with the processor, did not change. In addition, the instructions latencies [AMH23] stayed the same. All chip generations only vary in three major ways: local memory available for a tile, number of tiles on the chip, and clock frequency. Table 1.1 summarizes the most important specifications of the three architectures. Please note that the tile frequency dropped from the first to the second generation, but the tile memory and tile count rose. The third generation differs from the second only by the 40% increased clock frequency and the specifications scaled by that factor; the design of the chip stayed the same. This frequency increase was achieved through an additional power interposer layer.

Table 1.1: Architecture features of the first three IPU generations.

Chip	GC2	GC200	Bow	Unit
Number of tiles	1216	1472	1472	
Number of threads	7296	8832	8832	
Memory per tile	256	624	624	KB
Total SRAM memory	311	918	918	MB
Memory bandwidth	46.6	46.9	65.78	TB/s
Aggregate tile-to-tile bandwidth	7.78	7.83	10.9	TB/s
Total chip-to-chip bandwidth	320	320	320	GB/s
Clock frequency	1.6	1.33	1.85	GHz
FP32 compute	31.1	62.5	87.5	TFLOPS/s

The main programming frameworks for the IPU as an AI accelerator are TensorFlow and PyTorch. However, Poplar, a lower-level, more general-purpose programming framework is also exposed to the user to allow for developing general algorithms. Poplar is a dataflow programming framework similar to the way that TensorFlow and PyTorch define their programs. In dataflow programming, the program is modeled as a directed graph describing the transition between data and its relation to transferring functions. The dataflow graph is built with two switching layers, a data layer with the vertices being data and a compute layer with the vertices being transfer functions, the edges model the relations between data an functions. These build the three main components of the Poplar framework, tensors store data, compute-vertices in the form of codelets run user-defined functions, and the edges define data dependencies and movement. Tensors are one to multidimensional arrays, which can be sub-divided into regions. Poplar requires all tensor regions and codelets to be mapped to tiles. The mapping is required as defines how tensors are distributed on the hardware; for codelets, it explicitly defines how to parallelize the computations, as codelets are run at the same time on different tiles.

In comparison to the explicit message passing interface (MPI), commonly found in the world of HPC applications, the IPU uses the mapping information of tensor regions and the codelets and tile indices to implicitly define communication. The mapping allows the Poplar compiler to generate exchange code, from the tile owning a tensor region to the tile that computes a codelet. Once the computation is done, the data is transferred to the tile which the respective region of the output tensor was mapped to. As a consequence, mapping is a crucial step in programming the IPU, as the mapping of problems will determine the communication and, therefore, the performance of the final program. The data input and output regions of the tensors will be fully allocated on the tile, unless it is an in-place operation. A crucial aspect to note is that, when using regions with continuous memory not aligned to four-byte intervals, the compiler introduces an aligned region, leading to an increase in the allocated space for input and output tensors and additional copy operations.

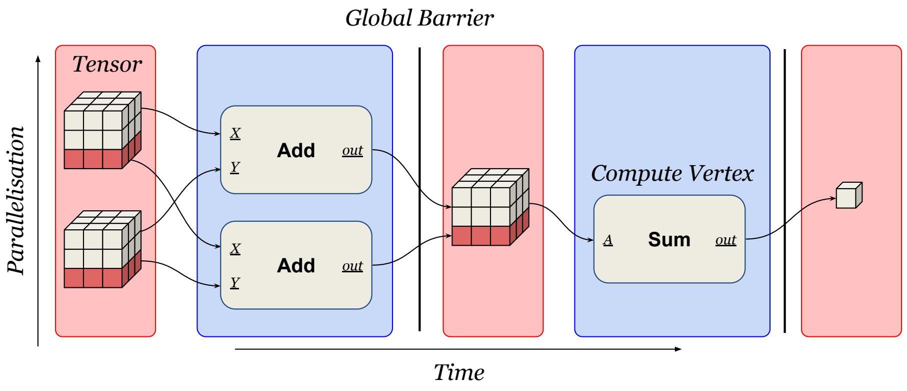

Figure 1.2: The graph represents the main components of the Poplar programming framework. Tensors store data, compute-vertices transform data and run user code, and edges implicitly define the communication. The graph alternates between state and compute layers. In this graph, the IPU executes the compute-vertices *Add* in parallel, as all compute-vertices within a synchronization step can not have data dependencies.

Scheduling and synchronizing exchanges and computation is done through the use of the bulk-synchronous parallel pattern (BSP) [Hil+98; McC95; Val90]. The BSP pattern has an exchange, compute and synchronization step creating a superstep. Remembering our two-layered dataflow graph (Figure 1.2), after the data and compute layer we can introduce a synchronization step, waiting for all computations to produce a result. After the synchronization step ensuring that all data is ready for the next computation layer. The order of supersteps can also be modified through high-level control flow, such as *if*-conditions and loops. Compute and exchange can not be done at the same time on the tile level, so we can either compute or communicate. Another restriction that comes with the BSP pattern is that exchange phases are determined at compile time and can not be dynamically created or changed, such as transfer sizes or access to and from a tensor. This is a major obstacle we have to overcome when implementing irregular algorithms on the IPU.

IPU programming is split into two parts, similar to CUDA programming for GPUs. The host code defines the dataflow graph and which codelets to use. In addition, in the host code we can define host-and-device transfer phases. The other half of the code is the definition of the codelets, that will be run on the IPU. Poplar defines codelets as templated C++ classes, defining a standard function moving data from an input tensor to an output tensor. The whole computational graph is compiled and uploaded to the IPU. At runtime, data can be provided through predefined host-and-device transfers.

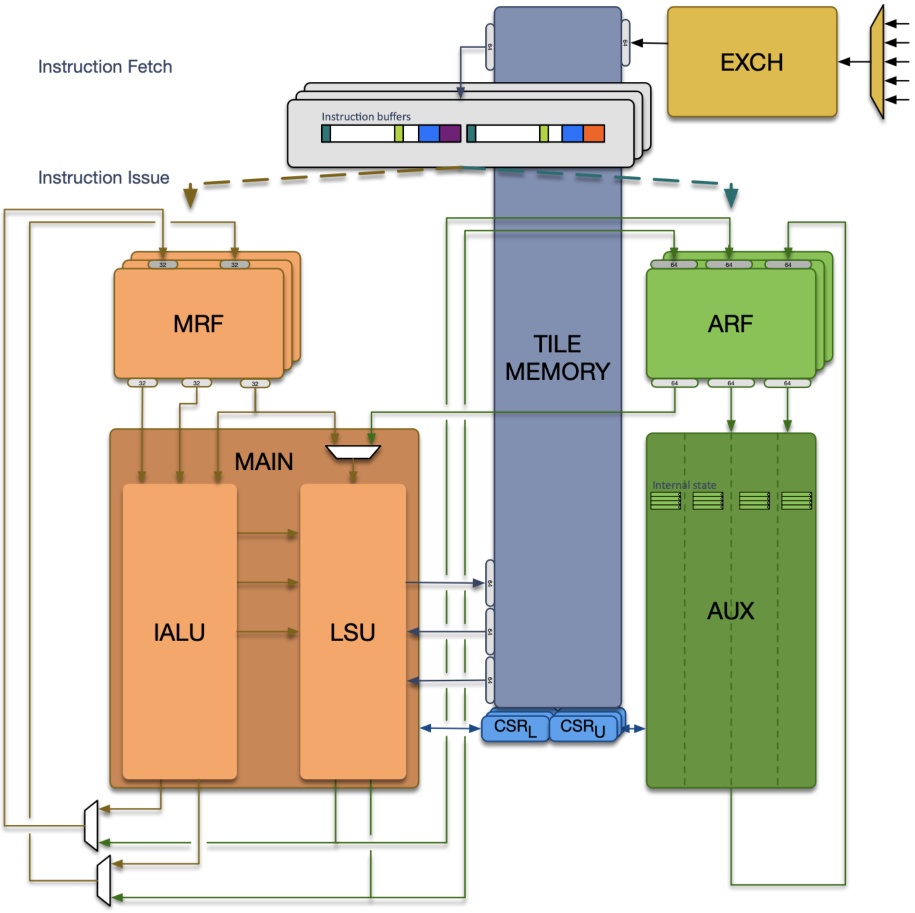

Figure 1.3: Tile processor architecture. The MAIN pipeline is used for integer processing and the control flow. It loads data into the AUX pipeline, responsible for floating point operations. The outgoing exchange is not programmable by users and is separated from the pipelines. MRF and ARF are register files containing the register values for each respective pipeline and thread. Copyright Graphcore.

1.2.1.1 IPU Tile Processor Architecture

The IPU architecture has a rather simple processor core design, without common optimizations found in CPUs. Modern CPUs try to optimize latency caused by stalls of loading data, and occupancy of functional units, and archive high instruction level parallelism. Therefore, CPUs commonly employ mechanisms, such as out-of-order execution, data prefetching, and speculative execution. As the IPU only has access to local SRAM memory, it does not need to optimize for stalls. The reduced control unit requirements to the processor reduce the physical footprint of it. Moreover, the program time estimation becomes simpler because the instruction execution is deterministic, allowing for more plannable program optimizations.

The processor, as seen in Figure 1.3, has two pipelines: `MAIN` and `AUX`. The `MAIN` pipeline is responsible for executing memory, control flow and integer operations, while the `AUX` pipeline is a floating point only pipeline, only controlled by the `MAIN` pipeline. Instructions can be dual-issued to both pipelines at the same time through the long instruction word architecture (LIW). The execution is lock-synchronous, meaning that an instruction bundle is retired once both pipelines are retired. As a consequence, the other pipeline has to wait and the scheduled instruction bundle take the maximum time of both pipeline's instructions. All Instructions that can be scheduled on the `MAIN` pipeline will retire within a single cycle, however, there are a few arithmetic operations on the `AUX` pipeline that can span multiple cycles.

The processor follows a barrel processor model with six threads, where from a thread point of view most instructions are retired within a cycle; however, the real latency is six hardware cycles. There is one register file attached to each pipeline with six execution contexts used for the six threads. These register files rotate every cycle to run another thread.

Graphcore does not provide a way of explicitly interacting with the exchange module and the exchange network on the tile processor, which is responsible for tile-to-tile communication. The only way to use the exchange capabilities is to use the provided high-level programming interface. In any case, exchange and computing can not be used at the same time per tile.

1.2.2 Sparse Linear Algebra

Sparse linear algebra refers to the branch of linear algebra that specifically deals with sparse matrices and the associated computations. A sparse matrix is a matrix that contains a significant percentage of zero elements, such that storing the matrix in a compressed form becomes advantageous. Due to the compressed representation, it is required to develop algorithms and techniques that take advantage of the sparsity structure to efficiently perform operations on these matrices. Traditional dense linear algebra algorithms are not well-suited for sparse matrices because they would waste computational resources and memory on processing the large number of zeros. Sparse linear algebra algorithms, on the other hand, aim to exploit the sparsity of the matrix to minimize the amount of computation required. In general, all algorithms employed in dense linear algebra can also be approached in sparse linear algebra to improve upon their real-world time and space complexity.

The goal of sparse linear algebra is to provide efficient and scalable solutions for problems involving large-scale sparse matrices, which commonly arise in diverse fields such as scientific computing, graph theory, network analysis, optimization, and many others. In scientific computing, it enables efficient simulation and modeling by solving systems of equations and analyzing complex systems. In graph theory and network analysis, sparse linear algebra techniques aid in processing large-scale graphs and networks for applications like social network analysis and recommendation systems. Optimization problems in operations research benefit from sparse linear algebra methods, allowing for resource allocation and logistics optimization. Finally, in deep learning and data mining, sparse linear algebra enables efficient processing and manipulation of high-dimensional sparse data structures. We focus on graph algorithms and scientific computing problems that use an irregular 3D-mesh structure. When using matrix representations, data from the real world often leads to sparse matrix problems, as the dimensions are large, but the contained data is small.

Buluç et al. [BGS11] describe the overlap of the four key sparse matrix operators for both graph algorithms and numerical algorithms:

1. *SpRef/SpAsgn* Sparse matrix indexing and assignment are used to embed a sparse matrix or submatrix in another sparse matrix or submatrix.

2. *SpMV/SpMSpV* Sparse matrix-dense vector multiplication, we extend this definition to include sparse matrix sparse vector multiplication.

3. *SpAdd* Sparse matrix pointwise operations, including addition, the Hadamard-product, or bitwise operations.

4. *SpGEMM* Sparse-matrix sparse-matrix multiplication is often used to represent multiple parallel SpMV operations.

Furthermore, Buluç, et al. [BGS11] discuss the sparse operations together, with respective storage formats in I/O complexity, which is a good fit because CPU architectures are cache reliant. However, for the IPU, lacking a cache, the RAM model is more appropriate. Thus, common algorithmic optimizations used for CPUs, that are beneficial in the I/O model, such as the blocked implementations are not useful for the IPU to improve cache performance. However, reordering into dense linear algebra blocks can yield obvious benefits, as vector instructions and wide memory loads and stores, and instruction pipelining can be used to improve performance.

In the following sections, we are presenting storage formats used for working with sparse linear algebra in scientific computing and graph applications (Section 1.2.2.1). Furthermore, we will give background on graph algorithms expressed in sparse linear algebra (Section 1.2.2.2) and motivate the Graph500 benchmark as an alternative to the Top500 benchmark to evaluate HPC systems for graph algorithms (Section 1.2.2.3).

1.2.2.1 Storage formats

The data structures used to represent sparse matrices are vital to the storage efficiency and time complexity under different algebraic operations. For example, for a matrix column access operation, the wrong data structure can lead to linear instead of constant access time. In the following, we will introduce three commonly used data structures for sparse matrices.

Let $A \in \mathbb{S}^{M \times N}$ be a sparse matrix of elements from an arbitrary domain $\mathbb{S}$. We use $nnz(A)$ to denote the number of nonzero elements in A, or simply nnz if A is apparent from context. We use $nzc(A)$ to represent the number of non-zero columns and $nzr(A)$ for the number of non-zero rows.

As an example we choose $A \in \mathbb{N}^{5 \times 5}$, the matrix to be composed of integers. An empty field in the matrix represents a zero.

$$A = \begin{bmatrix} & & 2 & & 3 \\ 1 & & 6 & & \\ & & & & \\ & & & 5 & 8 \\ 4 & & & & 7 \end{bmatrix}$$

We used two storage formats for our articles: the CSC/CSR format derived from the COO format, and ELLPack.

Tripplet Format The triplet format often called coordinate format (COO), is a simple representation of a sparse matrix. The data structure is a list or a set of triplets $D = \{(i, j, A_{ij}) | A_{ij} \neq 0\}$. We can also represent this set as a decomposed data structure $(A.I, A.J, A.V) := D$, here $A.I$ stores the row positions, $A.J$ is the columnar positions, and $A.V$ represents individual values. Analyzing the storage complexity of the representation we arrive at $\mathcal{O}(nnz)$. However, for the unsorted list we can not find ways to efficiently search the list; hence the time complexity for random access of the matrix is $\mathcal{O}(nnz)$. We can improve on this complexity, by sorting the values, row-major with a column-minor denomination or vice, versa. Thus through two binary searches, we can arrive at a time complexity of $\mathcal{O}(log(nnz) + log(M))$ or for column major $\mathcal{O}(log(nnz) + log(N))$. For example, the example a in COO format column major sorted, would look like $A = \{(2, 1, 1), (5, 1, 4), (1, 3, 2), (2, 3, 6), \ldots, (5, 5, 7)\}$

CSR/CSC Format A common optimization of the COO format is the write the major sorting direction of the triplets in a run-length compressed format. We call the format compressed sparse column (CSC), for when we majorly index by the columns, and similarly compressed sparse row (CSR) for row-major indexing. Here, we use the CSR format as the leading example. The data structure is comprised of the row-major indices $A.RI$ with $M + 1$ entries, which encode the row components of the triplets. The column component and value components are like in the decomposed COO format still stored in the $A.J$ and $A.V$ lists, without modifications. For accessing all elements in a given row, we can use the range stored in the $A.RI$ list $\{(i, A.J_k, A.V_k) | A.IR_i \leq k \leq A.IR_{i+1}\}$, yielding the COO formatted tuples of the row. The access to a row-index and the whole row list is thus $\mathcal{O}(1)$, and the scanning of the row $\mathcal{O}(log(M))$. However, the memory complexity of this format becomes $\mathcal{O}(M + nnz)$, as the row index space requirements are always in order of the matrix row dimensions.

For simplicity, in the previous paragraph, we were assuming that $nnz \geq M$, more specifically the number of non-zero rows is $nzr \approx M$. Often when we work with given problem matrices, these are fulfilling the $nzr \geq M$ criterion. However, with block partitioning matrices, it happens that the blocks become hyper-sparse matrices. We call a matrix hyper sparse when $nnz \ll M$, or more accurately, $nzr \ll M$. The DCSC format is an extension of the CSC format that trades off compute complexity for space efficiency. The DCSC format promises a storage complexity of $\mathcal{O}(nnz)$.

ELLPack Format The ELLPack (ELL) format compresses the values of a sparse matrix into fixed-size row structures. Its format is based on the concept of storing sparse matrices in a compressed format that avoids explicit storage row or column header information. It combines the benefits of a dense matrix and the CSC/CSR format, providing a balance between memory efficiency and computational performance.

In the ELLPack format, a sparse matrix is represented by two arrays: the data array and the index array. The data array $A.D$ stores the values of the nonzeros, while the index array $A.I$ is storing the columnar index for each value. The matrix also has a characteristic value $A.\lambda$, indicating the width of the longest row, indicating the fixed length of how many values are assigned to each row. Therefore, the storage complexity of this representation is $\mathcal{O}(A.\lambda \times N)$. This fixed-length representation allows for efficient parallel processing and vectorization of matrix operations, such as matrix-vector multiplication. It also enables efficient memory access patterns, minimizing cache misses and improving computational performance. While the ELLPack format offers advantages in terms of storage efficiency and computational performance, it is more suited for matrices that contain a relatively same amount of non-zeros in every row. Matrices with an inhomogeneous number of non-zeros per row may result in excessive storage requirements [MLA10] and reduced computational efficiency compared to other sparse matrix formats.

There are adaptations of the ELLPack format, e.g. sliced ELLPack [MLA10], which subdivide a row length-wise sorted part of the matrix into multiple ELLPack formats with a fitting characteristic length λ for each sliced partition.

1.2.2.2 Graph Algorithms Expressed in Linear Algebra

Graph algorithms can be expressed in the form of linear algebra [D31; KG11]. The graph is often represented as a matrix in the form of an incidence matrix, or adjacency matrix (Figure 1.4) depending on the algorithm [Kep+15]. This representation differs from the classical textbook graph algorithm formulation, which is often vertex-centric, as the linear algebra formulation is often applying vectorized operations on the dual graph-matrix structure.

This matrix representation brings several advantages, including the ability to apply a wide range of standard linear algebra operators, which enhances the generality of implementations. Moreover, expressing graph algorithms in linear algebra facilitates parallelization and optimization by relying on HPC implementations that can run parallel linear algebra operations. Therefore, graph algorithms can benefit from increased efficiency, portability, and scalability.

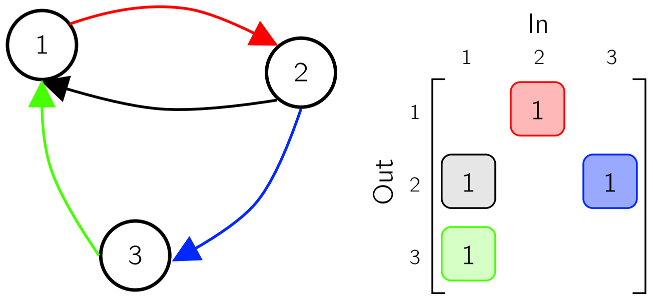

Figure 1.4: Dual graph representation (left) as an adjacency matrix (right). The boolean matrix encodes the edges between two vertices of the graph with a **1**, otherwise **0**.

A general formulation of graph operations is expressed through semiring algebra of a set of different semirings chosen specifically for each step in the formulation of the graph algorithm. A specific semiring $(\mathbb{D}, \otimes, \oplus)$ containing two binary operators called the plus $(\oplus)$, and multiplication $(\otimes)$ operators, which are defined to operate on a domain $\mathbb{D}$. Both multiplicative and additive operators have identities $\mathbf{1}$ and $\mathbf{0}$, respectively. Further, $(\otimes)$ distributes over $(\oplus)$ and the additive identity annihilates the set under the multiplication operator $\mathbf{0} \otimes a = \mathbf{0}$. For example, often used is the boolean semiring $(\mathbb{B}, \vee, \wedge)$ usign the boolean domain $0, 1$, with the identities $\mathbf{0} = 0$ and $\mathbf{1} = 1$.

1.2.2.3 Graph500

The Graph500 is a benchmark for parallel computers. It is orthogonal to the commonly used Top500 benchmark, for which supercomputers attain $\geq 97.9\%$, of the theoretical peak performance $R_{max} \approx R_{peak}$ [23b]. The Top500 benchmark measures the performance in floating point operations per second (FLOPS) over a dense matrix input. The Graph500 benchmark measures the traversal speed of a breadth-first search, traversing the whole graph from a random start node to produce a parent array, which represents one valid tree structure pointing to its parents in the traversal. The performance is given in traversed edges per second (TEPS). Compared to the Top500 benchmark, only a fraction of the theoretical peak performance of a given system can be attained when benchmarking sparse implementations [23a; SDM11].

Direction Optimization The direction optimization (DO) [BAP11] has become a prevalent technique for optimizing BFS targeted to the Graph500 benchmarks. The optimization observation reveals that during a graph breadth-first search, the frontier can become exhaustive, resulting in excessive operations when progressing towards a few remaining nodes. This occurs as the layers grow exponentially explicitly form the top, despite only requiring a single vertex to activate a remaining vertex. Consequently, a significant number of unnecessary operations are performed. The goal of the direction optimization is to identify a specific level during the traversal process where a switch can be made from the top-down approach, where each vertex claims its children, to a bottom-up approach, where the remaining vertices query their neighbors for activations. This switch results in a reduced number of computations heuristically. Thus, the term direction optimization is used as the algorithm can alternate between the top-down and bottom-up directions.

We further want to mention that this optimization only performs well on low-diameter graphs, as they are used in the Graph500 benchmark. Higher diameter graphs, such as road network graphs do not necessarily benefit from this optimization. Furthermore, current implementations [Wan+16] have shown that scaling the direction optimization to multiple hardware nodes is not possible without losing significant performance, making the implementation less scaleable.

Yang et al. [YBO20] improve upon this formulation in their GraphBLAST framework and formalize the direction optimization as pull and push-based semantics using sparse linear algebra operations.

1.2.3 Bioinformatics

All living organisms are composed of one or multiple cells that engage in intricate interactions, leading to the formation of purpose-driven organisms [Han04; SH05]. Analogous to a computer program, the genetic information required for the development of cells and other biological structures is stored within genomes, residing within cells. Bioinformatics, a specialized field, is interested in the investigation of this genomic data as it encapsulates all the necessary instructions for organismal production. A comprehensive understanding of this data is crucial in unraveling the intricacies of human biology, disease processes, and cancer etiology, among other phenomena. The majority of organisms' genomes are comprised of four nucleotide particles, collectively known as deoxyribonucleic acid (DNA), which serve as the fundamental building blocks.

The DNA contains four nucleotide *bases*, represented as $\{A, T, C, G\}$ for Adenine, Thymine, Cytosine, and Guanine, respectively. These four bases, are connected via 3'-5' phosphodiester bonds between the desoxyribose sugars coupled to the bases, create a long sequence that can be seen as a string. The DNA of bacteria, such as Escherichia coli (E. coli) have $4.5 - 5.5 \times 10^7$ bases [Rod+99], while the human genome has approximately 3×10^9 bases (GRCh38) [Sch+17]. In reality, the genome has two strands (physical strings) that make up the same sequence with complimentary bases called base pairs (bp). The two physical possible base pairs are $\{A, T\}$ and $\{C, G\}$.

The DNA creates Ribonucleic Acid (RNA), to serve different functions in the organism. To create proteins, through transcription, DNA is converted into mature messenger Ribonucleic Acid (mRNA), which contains a spliced section of DNA encoding information for the synthesis of a specific protein. The mRNA is then translated into proteins, and through the help of ribosomes, proteins are created. A ribosome successively takes three bases from the mRNA and assembles one of 22 amino acids, of which 20 are found in humans, into a protein structure. Thus, the DNA structure is responsible for creating organisms by acting as a blueprint.

To analyze DNA data the first step is to bring the data from a physical form into a digital representation. The digitalized form of DNA and protein sequences is often a coherent long string of characters of either the four bases of the amino acids, respectively. Gather digital data from the physical world sequencing devices are used.

The sequencing devices produce randomly located continuous substrings of the provided whole DNA sequence, each substring is called a *read.* The goal often is to assemble the reads into a single representation of the DNA, before other analyses can be conducted. Analogously this is like an illiterate person trying to reassemble 783 Encyclopedia Britannica's (8 billion characters) from a pile of shredded books.

Our articles (Paper III, Paper IV) focus on sequence alignment algorithms, which are used in bioinformatic pipelines to assemble genomes from a large set of sequence reads, that were provided by sequencing devices. The main motivation is that the sequence alignment methods and their implementation still make up a majority of the pipeline's runtime. This is further underscored by the exponential growth and cost-effectiveness of genomic data generation, which has propelled the field of bioinformatics toward the realm of HPC to tackle the computational demands imposed by these data-intensive processes.

1.2.3.1 Sequencing Technology

The first human genome, sequenced by the human genome project took 13-years, hundreds of research teams, and over a billion dollars. At the time of writing, the cost to sequence all important protein-creating regions in the genome, i.e. whole-exome sequencing, is less than 600 $ [Wet22]. The advances are in sequencing technology, as well as in the accompanying algorithms.

The human genome project utilized a sequencing technology known as Sanger sequencing, which was developed by Frederick Sanger and his colleagues in the late 1970s. Sanger subsequently, for this work in 1980 earned the Nobel Prize in chemistry, for determining the base sequence of nucleic acids. DNA, not RNA, is the target of these sequencing approaches, as RNA is deemed too unstable to be useful. The reads produced by the Sanger method usually have a length of 1000 bp [SL12; SNC77], at a cost-effective output of 750 Mb per day at an error rate of $\leq 0.1\%$ [Fra+13].

The next big change came with second-generation DNA sequencing devices, or next-generation sequencing (NGS) [HC16]. NGS allows for parallelization of the sequencing process, manifolding the reads output of the sequencing devices [Mar+05]. Modern sequencing devices, such as the Illumina HiSeq, can produce read sequences of approximately 150 bp, but with a much higher throughput of cost-effective 2500 Mb per day at an error rate of approximately 0.5% [Fra+13].

Modern or thrid-generation sequencing devices, using novel approaches, try to eschew the short read length and provide longer reads, which provide advantages in the late analysis. For example, at the time of writing PacBio HiFi sequencers can produce reads in the length of $10 - 25$ kb with an error rate of approximately 0.5% and better, while providing similar throughput as NGS devices. Oxford Nanopore can produce ultra-long sequences up to a length of 4 Mb, but producing lesser quality reads with more errors.

1.2.3.2 Sequence Alignment

Sequence alignment is tightly connected with string alignment methods. We interpret the biological sequences of DNA and proteins as strings. Thus, we use formal string methods to compare and relate sequences to each other. The goal of the string algorithm, as we use them, is to generate an equivalence score for two sequences. This score can be used to relate two sequences together; this will be used in biological pipelines to reassemble a long coherent sequence from a large set of smaller ones based on overlaps. A challenge these algorithms have to overcome is that the provided sequences can both contain read errors or modifications. These errors are modifications of the true sequence and can be insertions, deletions, or substitutions of characters. Another motivation is to determine the modifications applied to of homologous sequences that were introduced by biological sequences.

All discussed algorithms are given two sequences $S_1 = a_1a_2 \dots a_n$ and $S_2 = b_1b_2 \dots b_m$ of an alphabet Ω. We define a subsequence of S as a string P, that can be obtained by deleting zero or more characters from S.

One fundamental type of sequence alignment algorithm is the Longest Common Subsequence (LCS) algorithm. It is used to find the longest common subsequence present in both S_1 and S_2. For example a input strings $S_1 = \texttt{ATAT}, S_2 = \texttt{TCTC}$ will share the longest common subsequence $P = \texttt{TT}$. We can see the alignment of P to S_1 is `-T-T` and to S_2 it is `T-T-`. This can be seen as a simple form of string alignment with insertions and deletions.

A common theme to approach the LCS problem is to use a dynamic programming matrix spanned between the two input strings. The values of the dynamic programming matrix represent scores, that can be improved. The runtime is in $\mathcal{O}(n \times m)$. The algorithm then fills in the matrix by comparing characters and updating values based on specific rules. To construct the alignment, or for LCS, the longest common subsequence, we trace back the highest score from the matrix to its origin Figure 1.5.

Another popular string algorithm is the Levenshtein distance [Lev66], also known as the edit distance, the algorithm is an extension of the LCS algorithm that is specifically designed for sequence alignment. It is more robust than the LCS algorithm because it also allows for substitution in the aligned sequences.

However, in these string alignment methods, biological properties are not honored, a common theme for produced reads is that a long gap is more likely

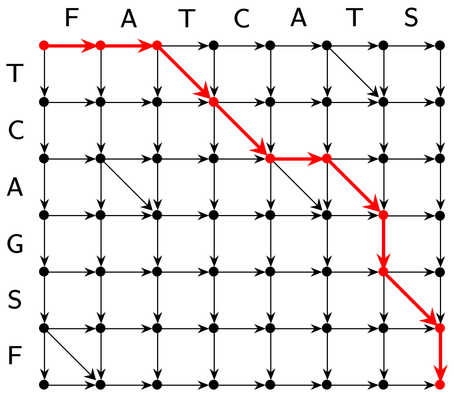

Figure 1.5: Longest Common Subsequence dynamic programming matrix. The axis is created through the input sequences, while the fields of the matrix are filled with values, depending on a match or mismatch. The result can be found by tracing back from the highest result on the path containing the most matches (`--TC-T-S-` or `--TCA--S-`).

than two adjacent ones. We call this affine-gap penalty. For protein sequences, we also want to assign a probability to special substitutions, as not all substitutions occur with the same probability in nature [HH92]. Furthermore, because reads can appear in random positions of a provided reference sequence we need sequence alignment algorithms, which can work in different scenarios. There are three well-specified alignment categories:

global alignment Global alignments are the most restrictive form of aligning two sequences S_1 with S_2. For this alignment, we assume that the sequences are homologous and have the same start and end positions. In the dynamic programming matrix, the optimal path can only begin in the top-left and end in the bottom-right corner.

semi-global alignment The semi-global alignment is loosening the restrictions of the global alignment and allows a free cost gap on either side of one sequence each. This means, in the dynamic programming matrix the path is still fixed to the sides, but not to the corners anymore. This alignment is practical when the two string overlap at the borders.

local alignment Local alignment, compared to the global and semi-global alignments, has no restrictions on the start and end gaps. The start and end positions can be completely placed within one sequence. This is especially useful when one sequence is much longer and the smaller sequence is completely contained within the longer one.

Needleman-Wunsch [NW70] proposed an exact algorithm, similar to the LCS algorithm, which produces a global alignment. The algorithm furthermore was extendable with affine-gap penalties and could custom substitution values to honor biophysical probabilities through i.e. BLOSUM [HH92] matrices. The Smith-Waterman algorithm [SW81] produces a local alignment. Later Gotoh [Got82] improved the algorithm by reducing the algorithm from cubic, to quadratic time, Myers and Miller introduced the idea of computing stripes to reduce the memory usage to linear space from quadratic space [MM88], and extensions for affine-gap penalties were added [AE86].

In addition to the named exact sequence alignment algorithms, there is a multitude of heuristic algorithms, which trade the exact result with better space and runtime.

1.2.3.3 Biological Pipelines

There are several types of pipelines we used in our articles. The term is not well defined, as a pipeline can serve different purposes, i.e. variant calling, metagenomic profiling, base calling, read mapping, or de novo assembly, amongst others. We used two kinds of pipelines, two for DNA de-novo assembly, and one for protein clustering. The inputs to these pipelines are often important, e.g. short reads need other algorithms as long reads due to algorithm complexity and biological properties. Short reads are not capable of identifying longer repeating regions in the DNA, as the short reads are completely contained within the repeating sections, it is impossible to identify the larger compositions of sequence regions adjacent to the repeats. As the human genome contains repetitive sequences [Lan+01], read mapping is often used, where a read is mapped against a known ground truth (GRCh38), to detect mutations. However, mapping is not possible without a ground truth.

De novo assembly is used to assemble a set of unordered reads into long coherent sequences, named contigs, referring to a continuous genomic section. Unlike reference-based assembly methods, de novo assembly does not rely on a pre-existing reference genome. This feature enables the assembly of contigs even for species that have not been previously studied or characterized.

Metagenomic assembly is a technique used for the simultaneous analysis of whole microbiomes. Unlike approaches that involve isolating and analyzing individual subparts of the microbiomes, metagenomic assembly allows for the comprehensive examination of a sample containing multiple organisms. The objective is to reconstruct multiple contigs representing different organisms present in the sample and subsequently analyze their assignments and quantities accurately. This approach eliminates the need to separate and analyze microbiome subparts individually, streamlining the analysis process and providing a more holistic understanding of the entire microbiome composition.

A common technique found in various pipelines to reduce the search space is the utilization of k-mers. The idea is to reduce the search space by only comparing sequences that initially have a large exactly overlapping section. Essentially, a k-mer represents a substring of length k derived from an original string, often extracted from a read. By exhaustively collecting all possible k-mers, we construct a comprehensive histogram that encapsulates the distribution of these k-mers. When dealing with a sequence of length m, we generate $(m-k+1)$ k-mers. In scenarios where multiple reads cover the original sequence, there exists the potential for read errors to creep in. However, by collecting a k-mer histogram, valuable insights into potential read errors can be gained. Specifically, k-mers that manifest with a low frequency are likely attributed to reading errors and can be discarded. Notably, in an ideal scenario where all reads overlap, correct reads yield k-mers with a higher frequency, reinforcing their accuracy and reliability. This analytical approach enables us to discern errors from authentic data, thus bolstering the robustness and fidelity of our analyses.

1.3 Research Questions

In this section, we define the research questions that lead to the creation of our four articles. The questions arise when trying to use the IPU for sparse and irregular computations. Naturally, we want to limit the scope of our research and propose three questions focused on the programmability and techniques for using a novel architecture. We apply the research questions to three fields from the HPC landscape. Namely, we applied the IPU to *graph processing*, by using unstructured SpM(Sp)V operations, *scientific simulations* on irregular grids, and high throughput *bioinformatics* applications.

Using the IPU, compared to GPUs, we are interested in three abstraction levels, the single tile performance, the whole IPU performance, and the cluster performance across multiple IPUs.

Research Question 1: How to effectively utilize a single tile? When it comes to novel hardware and many-core architectures, it is crucial to attain high utilization of the provided elemental units that make up the entire processor. A tile is the fundamental scheduling unit on the IPU. Each tile is one of the thousands of tiny dedicated processors with attached dedicated memory that constitute the IPU chip. Each of these tiles has six threads, which share a memory domain. We consider a tile efficiently utilized when all instructions are productively used for the algorithm of choice while all threads are busy.

We focus on different tile threading techniques in each article. In Paper I we focus on multiple threads inserting data into a shared queue without atomic primitives. In Paper II we look at instruction latency and instruction level parallelism for the IPU. In Paper III, we test different data types and compiler optimizations and write hand-optimized assembly to attain real-world performance close to our proposed model. In Paper IV we found that replacement techniques for atomic counters can let multiple threads cooperatively work on a pool of shared problems of undetermined runtime. Furthermore, we improved the existing X-Drop algorithm to reduce memory requirements to make the algorithm run on a single tile.

Research Question 2: How to efficiently utilize a single IPU chip? The IPU's enforcement of the BSP pattern often results in sparse problems failing to effectively utilize all cores, leading to suboptimal performance. To achieve good efficiency, implementations must maximize the utilization of tiles on the chip. Furthermore, it is essential to ensure that all tiles are utilized for an equal amount of time, avoiding uneven completion and load imbalances that result in idle tiles, thereby inefficiently utilizing the entire processor.

We have found different answers on how to utilize a whole chip effectively. In Paper I we randomly permuted the matrix to obtain a uniform distribution for the non-zero elements of the SpM(Sp)V operation. In Paper II, we used METIS to partition the underlying 3D structural graph into many equal-sized partitions. In Paper III we used a k-partitioning algorithm to balance the plannable runtime of the many problems and achieve a high utilization. In Paper IV we used heuristics and greedy subproblem splitting together with our k-partitioning from Paper III to create roughly equal-sized batches.

Research Question 3: How to utilize multiple IPUs efficiently? Scaling to multiple IPUs can trivially be achieved with the Poplar programming model. However, when seeking high performance, placement and communication need to be considered, as the intra-IPU bandwidth is an order of magnitude faster than the inter-IPU bandwidth. Furthermore, the IPU only provides a ladder network topology configuration; a restrictive layout compared to other network topologies [Jyo+16]. We addressed this question in our four articles differently, where in Paper I we introduced a pre-reduction to reduce memory requirements allowing for larger systems to scale. In Paper II through matrix partitioning we minimized communication, and through reordering the matrix into a banded form most of the communication stayed between adjacent IPUs, allowing to scale in a ladder topology. In Paper III, we scaled to 64 IPUs, using single IPUs instead of a multi-IPU approach, avoiding the IPU-to-IPU communication, but focusing more on the host-to-device bandwidth. In Paper IV, we introduced a memory reuse strategy, which can reduce the transferred data by multiple times, reducing the network saturation and thus allowing it to scale to more IPUs on a shared network.

1.4 Summary of Papers

In this section, we introduce the four articles produced during this thesis and provide further motivation, background, and summaries of their contributions.

1.4.1 Summary of Paper I

Luk Burchard, Xing Cai, Johannes Langguth “iPUG for Multiple Graphcore IPUs: Optimizing Performance and Scalability of Parallel Breadth-First Search”. In: *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. Vol. 28, (2021, December), pp. 162–171. DOI: 10.1109/HiPC53243.2021.00030.

The first article of this thesis explores the use of multiple IPUs to accelerate breadth-first search (BFS) in graphs. Multiple IPUs are used to avoid the limitations of using the slow DRAM attached to the IPU, limiting the graph placement to only on-chip memory providing more SRAM memory and higher aggregate bandwidth. The underlying technique explored and discussed is a sparse matrix-vector (SpMV) operation on a boolean semiring. The operation can be summarized as $x_{\ell+1} = A \cdot x_\ell$, where $A \in \mathbb{B}^{|V|\times|V|}$ is the sparse adjacency matrix of a graph $G(V, E)$, and x_ℓ is the frontier, representing the activated vertices of the current iteration level ℓ. By adding a micro queue system for the frontier of new activations, we are sparsifying the input vector x_ℓ, making it a sparse matrix sparse vector (SpMSpV) operation. We are interested in accelerating BFS because it is a ubiquitous algorithm found throughout an ample number of graph algorithms. Further, benchmarking BFS serves as an indicator of challenges concerning other sparse and irregular requirements for communication and hardware usage. As a means of comparison to other implementations, we used the Graph500 benchmark, as it allows us to compare the performance of multiple architectures and implementations through a single score.

The BFS input is a sparse adjacency matrix created from a graph. To avoid the advantages of specialized partitioners and only benchmark the BFS implementation, the input has to be assumed to have a randomized pattern. The inputs are scale-free graphs resembling properties found in online social network graphs, with on average of 7 edges between any two vertices in the graph [Ama+00; BW00; WS98], and a low graph diameter. This low graph diameter property makes the traversal have only a few levels increasing the parallelism by having a lot of operations in each level. For comparability, the benchmark prohibits the use of matrix permutations that make use of the specific graph generator. We also extend our analysis to graphs not specified in the benchmark and use real-world input graphs found in the SuiteSparse [Kol+19] dataset. These graphs can be more challenging to compute as they have a higher diameter and can not use a high level of parallelism, which poses a challenge for highly parallel architectures.

The goal of this first article is to discover building blocks for the general SpMV/SpMSpV algorithm on multiple IPUs, which have applications in a wider range of applications.

The Graph500 benchmark suggests to use the largest possible input instance. The Kronecker graph generator used to generate the standardized benchmark inputs is unfair in the sense that the graph densifies with higher dimension, giving especially the direction optimization a benefit with larger input instances [SPK13]. This makes it difficult to compare larger systems with smaller ones.

Motivation The idea of using the IPU to accelerate BFS is that the algorithm is heavily bounded by memory bandwidth and latency, especially through the random fine-grained reads and writes that are required. On CPUs and GPUs this leads to cache misses and makes it difficult to archive coalesced memory access. The thought is that the IPU is capable of providing fast fine-grained memory access with its order of magnitude higher memory bandwidth compared to CPUs and GPUs. Our previous publication used the earlier GC2 IPU with fewer tiles and much less SRAM per tile (Table 1.1) and only a single IPU instance used for the BFS algorithm [Bur+21].

Communication Reduction The BFS implementation is split into two phases, the fold and expansion phase, this also applies to general SpMV and SpMSpV. The expansion phase is responsible for computing the partial matrix-vector multiplication results. All tiles in this phase are working on small 2D blocked partitions of the matrix which are distributed among the tiles. All tiles are multiplying their 2D blocked partition with the partial input vector of the current level x_ℓ. The fold phase is a reduction operation that combines the results from the expansion phase into the output of the BFS iteration $x_{\ell+1}$. We view the fold phases input as unstructured as each partition will likely produce a different output with a varying amount of non-zeros which has to be reduced. Thus, we have to communicate different data for each partition. During the expansion, all partitions along the column receive the same input vector. We utilize this observation to minimize transferred data.

To implement the initial algorithm we $2D$ block partitioned the adjacency matrix and the respective in and output vectors.

To implement the initial scaling across multiple IPUs, we used a $1D$ block partition among the IPUs. When applying the $1D$ block partition of the $2D$ partitioned matrix, all partitions of one row are always placed on the same IPU. This has the advantage that the fast on-chip network is used. The division across the columns induces less communication effort because we use broadcast messages, sending data only once and receiving it many times. In poplar this is done by expressing a one-to-many relation of a tensor to the input field of multiple compute-vertices. This way, the communication effort across multiple IPUs is only at most $\mathcal{O}(\frac{|V|}{p}p) = \mathcal{O}(|V|)$, instead of $\mathcal{O}(\left[\frac{|V|}{p}p\right]p) = \mathcal{O}(|V|p)$, when using p 1D partitions.

Multi IPU Scaling Instead of using binary encoded vectors to denote the frontier, we encoded the frontiers x_ℓ as a set of small queues, where each queue is responsible for its respective partitioned section of x. This representation allowed us to compress the information in the vector. We observed that depending on the discovery of the BFS level, the number of non-zeros of these queues varies (Figure 1.6).

Observing that a lot of time is used for communicating the frontiers into the expansion phase, especially with a growing number of IPUs, we intend to reduce the communication volume. The reason for the growing latencies with more IPUs was due to the ladder topology of the interconnect network, introducing longer latencies when communicating between IPUs. Also, considering weak scaling, the bisection bandwidth stayed the same, but the volume over the middle links increased, taking more time.

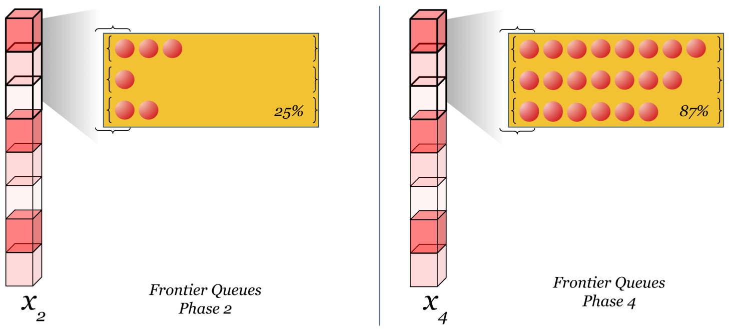

Figure 1.6: The frontier is represented as a list of smaller queues, each serving their respective partition. The queues encode the location of activations in the BFS frontier. We observed that in earlier levels the frontier is mainly zeros, while in peak levels the frontier is mainly filled with non-zeros during the BFS iterations.

To reduce the communication size, we cut off the queues containing the zeros and only send non-zeros. This is not trivially possible as communication pattern and size is not dynamic and has to be determined at compile time. We introduced a max reduction over the queue lengths to find the communication size minimally required to transfer all values. We use this maximum queue value, logarithmically rounded to reduce the number of possible communication sizes and stay 2-competitive with respect to the best communication volume as the communication cutoff. We showed that the max reduction is well amortized in the overall runtime. However, we noted that when using non-randomized graphs, the optimization is not effective, indicating that for better load balancing, a random permutation should always be applied to the input.

Conclusions We found that the performance for SpM(Sp)V operations depends on the structure of the input matrix. Especially high-diameter graphs do not benefit from our optimization. Even if the input is structured, we found that for the sake of load balancing it is beneficial to destroy the pattern of the graph and uniformly randomize the input, such that all tiles gain work and can contribute to the parallelization.

A shared memory system with synchronization operations has the advantage of splitting a problem into multiple smaller parts, where computation is parallelized but the memory complexity stays the same except for a small overhead. The IPUs do not have synchronization operations on the same tile but offer six threads. This results in data structures that need to be replicated instead of being synchronized. Even though it is possible to synchronize problems working on dense data through the deterministic instruction issue time, sparse and irregular problems can not implicitly be synchronized. Our implementation would have benefited strongly from simple tile-local atomic counters, which would have reduced the memory requirement by 6× and provided speedups.

Another takeaway from this BFS example was that not using the floating point capabilities of the IPUs was inefficient, as there are two pipelines that can be utilized but one is used, indicating that algorithms need to make use of the second pipeline (`AUX`) to attain higher performance. As the IPU is built for deep learning, much of the hardware resources are allocated for floating point operations, found in the `AUX` pipeline. Finally, the BFS benchmark does not play into the strengths of the IPU, as data is read only once; higher data reuse would show better real-world benefits due to the fast random memory access capabilities of the hardware.

Scaling more than 8 IPUs was made impossible because of the compilation process, which kills the compilation after more than 8 IPUs requiring exponential time and memory. Also, we noticed shortcomings in the network topology of the IPU as the network is built like a ladder, linear scaling leads to linearly increasing latencies and linearly increasing congestion. Ideally, a hypercube or star topology would help the IPUs to improve the scaling performance. We avoided a lot of traffic through the use of broadcast messages, made possible through our work division scheme but also note that the generalized workloads likely can not benefit from this communication reduction.

1.4.2 Summary of Paper II

Luk Burchard, Kristian Gregorius Hustad, Johannes Langguth, Xing Cai "Enabling Unstructured-Mesh Computation on Massively Tiled AI-Processors: An Example of Accelerating In-Silico Cardiac Simulation". In: *Frontiers in Physics*. Vol. 11, (2023, March), pp. 105. DOI: 10.3389/fphy.2023.979699.

A majority of HPC applications are used for modeling physical objects, such as fluid dynamics, complex systems, structural engineering, weather forecasting, chemistry and material science, and other life science applications. Of the mesh-based approaches these can be roughly divided into regualr and irregular methods. We are interested in the unstructured mesh-based methods, as they exhibit bad coalesced memory access pattern, which is difficult to accelerate with a GPU.

For the second article, we are focusing on an irregular mesh-based finite volume cardiac electrophysiology simulation. The simulation uses a reaction-diffusion model, including the cell model and the diffusion of the electrical signal through the cardiac fibers. We are building upon the LYNX code [Lan+19], which solves the monodomain model using CPUs or GPUs. It originally was built for MPI parallelization and later extended to use GPU clusters [Hus19], connected with MPI.

The fundamental idea of LYNX is that the reaction-diffusion model gets separated into two parts, the PDE and ODE parts. The ODE part, a set of ordinary differential equations is operating on a dense vector and encapsulates the main arithmetic segments. In the PDE part, the partial differential equation that couples all the computational elements is solved by a sparse matrix-vector operation (SpMV) per time step, where the matrix is sparse and the pattern is determined by the unstructured-mesh topology. The results of the PDE part are input to the ODE part and vice versa. We iteratively compute one after another, called a step. LYNX has to do multiple hundreds of thousands of these steps, thus we can spend time optimizing the computations, as they will become amortized over a long runtime of multiple minutes to hours, depending on the input.

The original LYNX code significantly benefited from using GPUs, as the cell model requires high arithmetic intensity on a dense vector of values, which the GPU can provide. However, when analyzing the runtime for PDE and ODE shares of the total runtime for a mid-sized mesh resolution, we observe that the PDE is taking 2.7× the runtime of the ODE section. This is due to the difficult coalesced access patterns. Through the learnings in SpMV calculations from the BFS code we intended to speed up the PDE part of the LYNX simulator.

Background The SpMV operation performed in LYNX is compiled specifically for the IPU, as the intention is to amortize the startup costs through good planning. The geometric nature of the mesh implies that rows of the sparse stiffness matrix required for the PDE part will never exceed 17 non-zeros per row. Thus, we choose an ELLpack format, compared to the CSR/CSC format used for the BFS, which has the advantage of being more memory efficient and also provides an advantage for SIMD operations when considering CPUs. Thus using this ELLPack format, we are looking at a 1D-partitioning problem with unstructured dependencies between the partitions.

The CPU code reorders the matrix using a standard graph partitioner METIS, for cache hit performance. This places the non-zeros into the diagonal of the matrix and minimizes cuts, which would result in communication. We keep this reordering, not to increase cache hits but to reduce the communication volume between the partitions.

Matrix Partitioning and Inter-Tile Communication The matrix partitioning we use in the article assumes that the majority of non-zeros are placed in blocks on the diagonal. This permutation of the matrix is done at a previous step through partitioning. Then applying 1D cuts to form partitions along the rows and blocks, will ideally split the matrix into multiple disjunct blocks, as well as the input vector. As in practice, the mesh is not split into many disjunct blocks non-zero values are placed into column-owned regions of other partitions. This requires the value to be communicated for the iterative result of the SpMV operation. We follow the traditional HPC fashion and divide the cells of the split matrix into three regions, the interior (local), separator (to be communicated), and halo (to be received) regions. We facilitate the data movement between the tile from the separator sections to the halo section of another tile by mapping the tensor regions in the Poplar framework.

In this article, we explore how the communication and mapping of the partitions have to be handled on the IPU, as too fine-grained communication is not well compilable and too coarse communication is overhead heavy. We thus

Table 1.2: Allocation volume of cells for each tile. The three communication reordering strategies surveyed in the article were Full-separator, Mixed-separator, and Range-separator, while the dominant strategy in all cases is the Mixed-separator strategy.

		IPUs					
Method	*Cell Kind*	1	2	4	8	16	32
Full	TotalCells	9858	6221	3829	2357	1448	845
	Interior	2058	1030	515	258	130	65
	InboundVolume	7812	5184	3314	2099	1319	780
	OutboundVolume	7656	5030	3228	2093	1339	793
Mixed	TotalCells	4272	2803	1887	1303	930	649
	Interior	2058	1030	515	258	130	65
	InboundVolume	2226	1775	1370	1045	801	585
	OutboundVolume	2121	1649	1269	1005	811	592
Range	TotalCells	6417	3919	2479	1539	1099	579
	Interior	2058	1030	515	258	130	65
	InboundVolume	4358	2890	1962	1282	970	515
	OutboundVolume	4265	2875	1913	1307	982	518

evaluate three proposed techniques to minimize the communication time of the PDE phase. As a perfect reordering of the separator is not in polynomial time, we used heuristics, which can be found in Table 1.2.

Interestingly, we observed that the effectiveness of the partitioning is becoming quite bad when using finer partitioning. The partitioning requires the cells to exchange more than 90% of their cell volume, meaning that almost all cells have to be communicated throughout the processor. However, when observing the full-time share of PDE and ODE time required the time of the PDE never exceeds the time required for an ODE, not even with 16 IPUs.

Multi IPU Partitioning When scaling to multiple IPUs, we further relied on the idea that the majority of the matrix non-zeros are placed into blocks, thus communication would in the majority only happen between adjacent blocks. Hence, an extension of the 1D splitting regime seemed feasible. For the communication volume between more IPUs, we can observe that the used approach works reasonably well, as the diagonal has the most communication. However, placing these partitions onto a hardware topology might pose a larger combinatorial problem.

Conclusions We have shown that the particular electrophysiological simulation considered in this paper can be accelerated using single and multiple IPUs. And to the best of our knowledge we were the first to show that it is feasible to use the IPU to accelerate irregular mesh-based methods. Further, we found that our approach improved the share of PDE to ODE from 2.7× on the GPU down

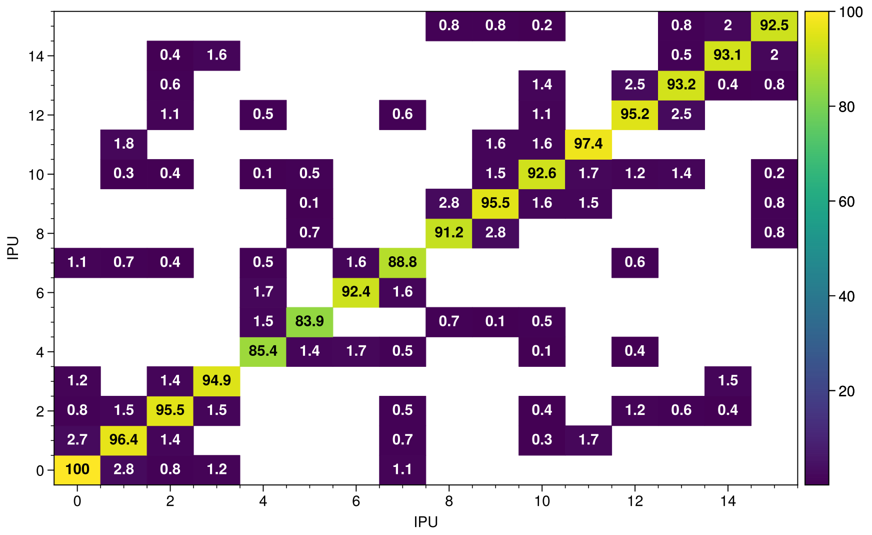

Figure 1.7: Lynx communication volume between multiple IPUs, normalized to the largest communication volume. Intra-communication volumes are on the diagonal, while inter-communication volumes are on the off-diagonal.

to 0.3× on the IPU. However, as the ODE part is heavily reliant on arithmetic instructions, the GPU is faster than the IPU for the ODE part. As the IPU only supports single precision floating point data types, we verified that the used simulation will produce accurate results compared to the double precision CPU implementation.

In the end, a single IPU was still outperformed by a single GPU in the ODE part, as the GPU is exceptionally good for regular SIMD workloads. The IPU struggles here.

As with the BFS article, we were not able to compile our code for our available 64 IPUs. We were limited to 16 IPUs, similar to the BFS article, because the compilation would run out of time and memory. We observed exponential memory usage and time requirement.

Seeing the SpMV IPU workload perform much better than on the GPU, but the arithmetic-intensive part performs worse hints to us exploring workloads that are inherently more sparse and irregular.

1.4.3 Summary of Paper III

Max Xiaohang Zhao[†], Luk Burchard[†], Daniel Thilo Schroeder, Johannes Langguth, Xing Cai "iPuma: High-throughput Sequence Alignment for MIMD

[†]Shared first author, both authors contributed equally.

AI Accelerators". In preparation, to be submitted to journal.

The third article dives into bioinformatic applications, which are reliant on string comparisons. Bioinformatic applications are part of the field of HPC applications and often require string alignment algorithms in the beginning when aligning DNA reads. An important observation is that the CPU is still the dominant architecture for most bioinformatic pipelines, as GPUs require homogeneous operations on their data which is often not given for these string alignments.

The algorithm of choice is the Smith-Waterman (SW), as GPU implementations are still performing equivalently well [Bar20] to CPU implementations and could not improve the algorithm like in other fields, where speedups in the order of magnitudes were possible. Moreover, GPU implementations are only performing well, when long sequence comparisons are assumed; this is not a valid assumption in reality as sequence reads have for most applications and next generation sequencing a more limited and varying length.

The goal of the article was to produce a library that can provide production-quality local alignments through SW, that perform in real-world use cases. While being flexible enough to be employed for DNA and proteomic data without restrictions, as most libraries optimize for special use cases i.e.: only DNA, or extremely long sequences.

Optimizing for Throughput For real-world applications, the IPU poses strong restrictions, as the interconnect to a controlling host computer is slow compared to a CPU or GPU. GPUs for the DGX-2 V100 are connected with a 300 GB/s link and have almost 900 GB/s memory bandwidth, CPUs have approximately 400 GB/s memory bandwidth and the data is already available. The IPUs numbers are pale in comparison with a shared 12.5 GB/s link and 20 GB/s on-IPU machine DRAM bandwidth shared across the four IPUs per IPU blade, of the Mk2 and Bow machines. Comparing these numbers would indicate that we can not do much against the CPU and GPU; however, we are not memory bound.

We introduced batching to send small work bundles to the IPU, keeping it busy, while preloading data to the DRAM next to the IPU. This does not help the bandwidth of the IPU but introduced pipelining, where the latency of synchronization was reduced as the IPU only had to communicate with the local DRAM. Software shortcomings made it not possible to use the approximately 400 GB DRAM, letting us only preload a single batch into the pipeline only occupying 2.4 GB of the DRAM. This did not improve the overall latency of the system, as work was preloaded to the IPU machines, but the overall throughput was increased.

Partitioning Because of the bulk synchronous parallel (BSP) pattern the IPU enforces it is not possible to do completely inhomogeneous computations. Because, when a single tile is long-running the IPU is required to wait for that single struggling tile. We used a k-partitioning approach in combination with heuristic approaches to load-balance the work batches onto the tiles, minimizing the difference of the makespan, and the time it takes to complete the last computation.

Performance Model A benefit of the IPU architecture is that building performance models becomes quite straightforward. We proposed a model which indicates that 195.776 GCUPS were attainable. Later we confirmed that the measured peak performance was 192 GCUPS. This is due to the deterministic performance of the hardware itself. We created the performance model by inspecting the instructions, thus simply improving the quality of the compilation could lead to better performance. However, as the code was hand-optimized we are confident that the peak performance of the hardware was attained.

Pipeline Integration As the goal was to produce a usable library, we showed with integration into two biological pipelines the practicality. For this, we used projects based on MPI and UPC++, PASTIS [Sel+20] and MetaHipMer [Geo+15; Geo+18]. Those projects fall into the category of distributed memory and shared memory systems, respectively. Although the design decisions were initially led by the MetaHipMer UPC++ project, we were not able to attain the high speedups we hoped for. This was due to the fact that multiple processes were placed on the same node and bad batch partitions could only be attained due to the too finely partitioned of the problem space. With the PASTIS project using MPI and OpenMP-based project, we could utilize the fact that we are on a fat node and used large batch partitioning, generating higher quality partitions. Thus, the speedups for PASTIS were higher, although the design originated from the MetaHipMer pipeline.

Conclusions We found that using the IPUs for string alignment algorithms occupies a sweet spot for next generation sequencing alignment with sequences approximately of length $200 - 350$ bp for current IPU generations. Shorter sequences are not practical as the communication overhead with the IPU and the little computational time will become amortized. For longer sequences, GPUs become increasingly more viable through their strong instruction throughput.

We want to notice here that our testing setup is present in a degenerate configuration with 64 IPUs per host node, while the proposed setup by the vendor has only 16 IPUs attached to a single host node. This leads to a faster saturation of the interconnect when scaling to all 64 IPUs. However, we were still able to show scaling for long protein sequences up to 64 IPUs, as the computational time is increased and thus the interconnect is not as much used.

1.4.4 Summary of Paper IV

Luk Burchard[†], Max Xiaohang Zhao[†], Johannes Langguth, Aydın Buluç, Giulia Guidi "Space Efficient Sequence Alignment for SRAM-Based Computing: X-Drop on the Graphcore IPU". Accepted for publication in *SC '23: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.* DOI: doi.org/10.1145/3581784.3607094

The fourth and last article of this thesis explores bioinformatics applications further, as in the previous article we established that string alignment algorithms are feasible to run on single and multiple IPUs. In addition we know that the IPU does not heavily benefit from SIMD instructions, but from tile-to-tile balanced instructions. Furthermore, our previous algorithm was performing well for short to medium sequences. Thus, we want to make it possible to accelerate medium to long sequences, from newer sequencing technologies, such as PacBio HiFi sequencing. The algorithm of choice was the X-Dorp algorithm, which exhibits a very irregular computational pattern and requires more memory. We again were interested in the real-world performance of our approach and tested it with two real-world pipelines, ELBA [Gui+21] and PASTIS [Sel+20].

GPU codes exist for the X-Drop algorithm, they parallelize the computations through the use of long diagonals using SIMD instructions in the algorithm. However, we want to note here that in real use cases with realistic parameters, it is not likely that such long diagonals exist. Thus, the GPU implementations are providing speedups in more theoretical scenarios, but can not provide acceleration in real-world applications. Furthermore, only DNA-based GPU implementations exist at the time of writing.

Space Optimization To make it possible to run X-Drop for larger sequences on the IPU we first had to reduce the memory usage of the algorithm itself. A single IPU tile currently has 624 kb of memory and input sequences are $30k$ symbols encoded as bytes. Alone the inputs occupy 360 kb of memory, a memory scratch space for the traditional algorithm would occupy another 540 kb, making it infeasible to use all six threads.

Similar to our other articles, we were not able to split the work between six threads, thus requiring 6× the memory. Therefore, because no synchronization instructions are available on the IPU, we have to provide six input instances.

Through the observation that only a small part region of the fully allocated scratch memory is actually used, we introduced an algorithmic change to the X-Drop algorithm to allocate only this active region. We analyzed the parameter empirically for theoretical and real instances and determined that in real applications we can reduce the memory usage up to 55×. This optimization made it possible to reduce the 540 kb of scratch memory down to under 10 kb of memory. Thus, we were able to use the remaining space for more input instances.

[†]Shared first author, both authors contributed equally.

Compute Graph Optimization A problem with the X-Drop algorithm is that the likely complexity the algorithm occupies is closer to linear than to quadratic. This, poses a problem for the IPU, as the slow interconnect costs are more difficult to amortize. However, we found that many pipelines need to do many-to-many comparisons. We utilize the fact that values of comparison pairs will be reused. Through a simple graph partitioning heuristic, we were able to increase the computations done per tile, without increasing the transferred data.

The increased utilization time on time IPU and decreased data transfers also resulted in better scaling efficiencies for multiple IPUs.

Thread Parallization X-Drop is an algorithm which has an undetermined runtime, as the termination conditions can be triggered spontaneously. Thus, it was very difficult to determine an accurate model for partitioning the work based on the apriori runtime, such that the BSP's global synchronization would not wait for a single tile. For this, we introduce a simple work division scheme, that splits the comparisons into smaller parts. Following the law of large numbers, we intend to minimize the runtime variance. We further improved this work division by changing the initial static scheduling with dynamic scheduling, which employs an eventual global counter. In the case of race conditions, through the idempotent property of the algorithm no information would be corrupted, only computed multiple times.

Conclusion We have contributed to the X-Drop literature by improving the space requirement of the X-Drop algorithm by $55\times$. Our implementation of X-Drop sequence alignment surpasses existing state-of-the-art implementations on both CPU and GPU platforms, delivering superior performance in both DNA and protein alignment across realistic X-Drop values, in a single library. Furthermore, we showcase near-linear strong scaling properties on common IPU host configurations, leveraging our graph view-based many-to-many sequence partitioning approach. However, due to the high work imbalances, we could only utilize approximately 40% of the IPU, indicating that better load balancing could improve our advantage to the CPU and GPU even further.

1.5 Conclusion

This thesis has provided a first step into exploring multiple applications from the field of HPC using the IPU. In this section, we look back at our research questions and try to answer them using the papers and contributions throughout the thesis. Furthermore, we will provide future work directions and recommendations to mitigate the current shortcomings of the IPU architecture itself. Together, we hope they can provide fruitful ground for easier adoption and higher-performing implementation of the IPU, most notably by requiring less engineering effort and imposing fewer restrictions on its users.

Research Question 1: How to effectively utilize a single tile?

Our first research question concerned the performance attainable on a single IPU tile. This topic comes up as the core of each research article. In articles I and IV, we used hand-optimized assembly to utilize dual issuing on the float and integer pipeline simultaneously. In article II, due to the large codelets we omitted hand-optimised assembly. In the last article, as the compiler became more efficient we only provided codelets in C++. However, attention still has to be paid when working with C++ code, as the compiler in some cases will generate badly performing code. Therefore, we suggest always having the C++ and assembly code side by side for performance-oriented programming.

Splitting work per tile is possible when the amount of work is known at compile time; this has been applied in article II. However, doing dynamic fine-grained splitting and work division is infeasible. In article IV, we proposed a more coarse-grained splitting of work that allows threads to partially share a highly inhomogenous single-string alignment algorithm on a probabilistic basis.

Therefore, under consideration of our explored applications, we would like to suggest improvements to the platform:

- In order to allow for a tile with all of its threads to share a problem instance, we suggest extending the ISA with atomic capabilities, such as an atomic add or a compare and swap. Already simple capabilities, such as a single atomic add counter, would have allowed us in article I and IV to work on a shared single problem instance instead of six problem instances per tile. This would also have the advantage of oftentimes requiring 6× less memory.
- When programming more irregular algorithms, especially when doing hand optimization, we would have benefited from an increased number of registers. Especially noting that only 12× 32 bit integer and 14 floating-point registers are available.

Research Question 2: How to efficiently utilize a single IPU chip?

The second research question concerns the whole IPU chip. Work can not easily be distributed, because memory and operation mapping will result in communication and global synchronization. In all articles, we dealt with workload balancing. In paper *I* we applied a random permutation of the inputs to balance work. While in article *II*, we relied on the partitioning of a graph partitioner for communication and workload balancing. In article *III*, we used an accurate model of the individual work batches to load-balance computation. However, not all work could be successfully load-balanced. In the last article *IV*, we coarsely split work to allow for eventual load-balancing. However, this still fell short, as we could only utilize approximately 40% of the IPU tiles, indicating that better load balancing could improve our advantage over the established architectures further.

Dynamic communication is not possible (Section 1.4.1), meaning that communication partners and transmission sizes can not be determined during runtime, but must be provided during compile time (Section 1.2.1). This makes creating a program for dynamically changing inputs and data difficult. In article *I*, we proposed a semi-dynamic scheme to reduce the transmission size although, communication peers could not be changed.

Another obvious problem we did not address in the writing of this thesis is the use of the DRAM for allowing us to work on larger problem instances. In articles *III* and *IV*, we used the DRAM tangentially to preload data closer to the IPU. However, this does only work better than using the interconnect directly but does still impose a long communication phase, which can only be amortized because the IPU chip itself does provide fast problem capabilities compared to CPUs and GPUs. The first articles completely avoided the storing of problem instances on the DRAM next to the IPUs, because the shared bandwidth to the IPU is about 5 GB/s, which is 400× slower than modern A100 GPUs with an approximate 2 TB/s access to high bandwidth memory. In addition, there is a trend of more architectures adding high bandwidth memory, such as the Intel Sapphire Rapids CPU. In articles *I* and *II*, we avoided the DRAM problem by adding more IPUs.

Hence, we would suggest a few recommendations to the hardware developers, based on our applications:

- A weakening, away from the strict BSP regime, would allow for more flexible synchronization, not requiring waiting for a single processor. This could be done through multiple BSP domains. This would eschew inefficiencies arising from uneven work distribution. Other architectures like any CPU or GPU have a scheduler, which naturally leads to these architectures having higher utilization of the available silicon.

- Dynamic communication on a chip level will also aid the distribution of larger problems. Currently, only compile-time static communication is possible. Dynamic communication can be used to distribute larger problems onto multiple tiles, such that IPU programs do not need to be recompiled for every input instance.

- The slow memory bandwidth of approximately 5 GB/s is a limitation for many applications. Although, when having high data reuse within the 900 MB, it might often not be sufficient to keep the IPU feed with data. Adding memory technologies with a higher bandwidth would place the IPU into more competitive territory with the high bandwidth memory GPUs and CPUs.

Research Question 3: How to utilize multiple IPUs efficiently?

Our third research question concerns the scaling-out aspect of using multiple IPUs. The IPUs are not directly attached to the host nodes but are provided on their bespoke system. The interconnect is part of that system called a POD; in our case we used a a POD-64 containing 64 IPUs. In all four papers, we have managed to scale out to using multiple IPUs. The scaling-out approaches can be categorized into two groups: *connected* and *isolated*. In the connected approach, a problem instance is created for a multi-IPU setup, while the isolated approach utilizes multiple IPUs from the host-connected node.

In articles I and II, we used a *connected* multi-IPU. In article I, we identified that scaling becomes an issue due to the ladder topology. We provided the reader with a strategy of mapping SpMV operations onto multiple IPUs and showed that through mapping the operations accordingly the data on the inter-IPU links would only grow linearly, compared to quadratic growth through using broadcast operations. Article II showed that using a $1D$ block distribution of a graph-partitioned ELLPacked SpMV operation will enable scalability without the downside of having infeasibly large partitions. We unveiled the shortcomings of using current partitioners and are motivated to use novel partitioning techniques.

Articles III and IV used an *isolated* IPU configuration, as no global communication was required. These implementations scale relatively well and are restricted by the node and interconnect performance between the host computer to the IPU POD instance.

In general, a problem for *connected* programs was the compile time and memory requirement of the compilation, which would grow exponentially. This compilation issue does not apply to the *isolated* IPU programs. Hence, for *connected* programs, we were not able to scale over 8 and 16 IPUs for article I and II, respectively. This is currently the most troubling issue for us, making it impossible to test hardware configurations reaching our 64 IPUs.

Therefore, we propose the following improvements to the IPU hardware setup:

- The network topology offered by the IPU is restrictive, as only a ladder configuration is available. This increases latency and creates bottlenecks along the communication pathway. Allowing for user-defined topologies, such as trees would allow to provide better scaling performance.

- Compile time requirements for *connected* multi-IPU programs are prohibitively long. An obvious solution is to improve the compilation and reduce compile time. We do not know what the reason for the slow compile times is, but allowing users to disable some optimizations and get slightly worse communication performance would be a sensible tradeoff.

Using the IPU for multiple applications in HPC, we noticed a pattern emerging for determining when to use the IPU. Firstly, all problems that are solved well on the GPU and have a very regular memory access pattern are not well suited for the IPU, as GPUs excel under these conditions. On the other hand, applications, which are data-intensive and exhibit a low computational complexity, are also not well-suited, as the off-chip memory bandwidth is one order of magnitudes slower than that of a CPU. We see an opportunity for the IPU, when applications are irregular, thus not well suited for the GPU, but also not data intensive. Furthermore, based on the large SRAM size, all problems which fit into memory can expectedly be solved fast on the IPU, as the cache bandwidth is one order of magnitude more than that of a modern CPU.

In our expirience when coding for the IPU, a significant portion, 90% of the time, is dedicated to setting up the host code in C++, while only 10% of the time is spent working on the actual codelets. To program the codelets effectively, we recommend analyzing the assembly code generated, as it offers an accurate representation of the expected runtime behavior. However, it's crucial to acknowledge a common misconception regarding the IPU, which is the assumption of straightforward and dynamic data transfer. This aspect must be carefully considered when programming for the IPU, as problems on a larger scale need to be regular in shape but not on a smaller scale. During the development process, we encountered several subtle performance issues, which were revealed through the utilization of the provided IPU profiling tools. However, the state of debugging code on the IPU remains somewhat obscure, making development a challenging task. While the profiling tooling could still benefit from further improvement, particularly for irregular programs, the existing programming frameworks enable the creation of custom tools.

Future Work The IPU is one of the more interesting AI accelerator architectures currently available. We hope to see future usages of the IPU in the field of HPC but are also aware of the challenges and hurdles we discovered. On the other hand, the platform provides good performance results, given adequate implementation efforts. Especially for sequence alignment algorithms, we intend to do further work, as the IPU seems to be excellent providing advantages over the current state-of-the-art, even at 40% utilization.

Given that SpMV operations perform well on the architecture, problems using multiple SpMV iterations seem to be suited and future research in these areas appears promising. This also applies to graph algorithms which are more compute-intensive than the explored BFS application.

Because 90% of the development time is used to set up the host code, we intend to do further research in the direction of domain-specific languages (DSL) for the IPU, or more expressive programming languages such as Python or Julia.

References

[23a] *HPCG - November 2022 | TOP500*. June 12, 2023. URL: http://web.archive.org/web/20230612135525/https://www.top500.org/lists/hpcg/2022/11/ (visited on 07/17/2023).

[23b] *TOP500 List - November 2022 | TOP500*. Apr. 1, 2023. URL: http://web.archive.org/web/20230401203322/https://www.top500.org/lists/top500/list/2022/11/ (visited on 07/16/2023).

[Abt+22] Abts, D. et al. “The Groq Software-defined Scale-out Tensor Streaming Multiprocessor : From chips-to-systems architectural overview”. In: 2022 IEEE Hot Chips 34 Symposium (HCS). IEEE Computer Society, Aug. 1, 2022, pp. 1–69.

[AE86] Altschul, S. F. and Erickson, B. W. “Optimal sequence alignment using affine gap costs”. In: *Bulletin of Mathematical Biology* vol. 48, no. 5 (Sept. 1, 1986), pp. 603–616.

[Ama+00] Amaral, L. A. N. et al. “Classes of small-world networks”. In: *Proceedings of the National Academy of Sciences* vol. 97, no. 21 (Oct. 10, 2000). Publisher: Proceedings of the National Academy of Sciences, pp. 11149–11152.

[AMH23] Alexander, A., Mangnall, J., and Hedinger, P. *Tile Codelet ISA*. July 12, 2023. URL: https://web.archive.org/web/20230712222926/https://docs.graphcore.ai/projects/isa/en/latest/_static/Tile-Vertex-ISA_1.2.3.pdf (visited on 07/12/2023).

[Ang+23] Ang, J. A. et al. “Codesign for Extreme Heterogeneity: Integrating Custom Hardware With Commodity Computing Technology to Support Next-Generation HPC Converged Workloads”. In: *IEEE Internet Computing* vol. 27, no. 1 (Jan. 2023). Conference Name: IEEE Internet Computing, pp. 7–14.

[BAP11] Beamer, S., Asanovic, K., and Patterson, D. “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).

[Bar20] Barnes, R. *A Review of the Smith-Waterman GPU Landscape*. Tech. rep. University of California at Berkeley, Aug. 2020, p. 27.

[BGS11] Buluç, A., Gilbert, J., and Shah, V. B. “13. Implementing Sparse Matrices for Graph Algorithms”. In: *Graph Algorithms in the Language of Linear Algebra*. Software, Environments, and Tools. Society for Industrial and Applied Mathematics, Jan. 2011, pp. 287–313.

[Bre+14] Breß, S. et al. “GPU-Accelerated Database Systems: Survey and Open Challenges”. In: *Transactions on Large-Scale Data- and Knowledge-Centered Systems XV: Selected Papers from ADBIS 2013 Satellite Events*. Ed. by Hameurlain, A. et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2014, pp. 1–35.

[Bur+21] Burchard, L. et al. “iPUG: Accelerating Breadth-First Graph Traversals Using Manycore Graphcore IPUs”. In: *International Conference on High Performance Computing*. Springer. 2021, pp. 291–309.

[BW00] Barrat, A. and Weigt, M. “On the properties of small-world network models”. In: *The European Physical Journal B - Condensed Matter and Complex Systems* vol. 13, no. 3 (Feb. 1, 2000), pp. 547–560.

[D31] D, K. “Graphok es matrixok (Hungarian) [Graphs and matrices]”. In: *Matematikai es Fizikai Lapok* vol. 38 (1931), pp. 116–119.

[Ema+21] Emani, M. et al. “Accelerating Scientific Applications With SambaNova Reconfigurable Dataflow Architecture”. In: *Computing in Science & Engineering* vol. 23, no. 2 (Mar. 2021). Conference Name: Computing in Science & Engineering, pp. 114–119.

[Fan+04] Fan, Z. et al. “GPU Cluster for High Performance Computing”. In: *SC ’04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*. SC ’04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing. Nov. 2004, pp. 47–47.

[Fra+13] Frank, M. et al. “Genome sequencing: a systematic review of health economic evidence”. In: *Health Economics Review* vol. 3 (Dec. 12, 2013), p. 29.

[Geo+15] Georganas, E. et al. “merAligner: A Fully Parallel Sequence Aligner”. In: *2015 IEEE International Parallel and Distributed Processing Symposium*. May 2015, pp. 561–570.

[Geo+18] Georganas, E. et al. “Extreme Scale De Novo Metagenome Assembly”. In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. SC18: International Conference for High Performance Computing, Networking, Storage and Analysis. Nov. 2018, pp. 122–134.

[Got82] Gotoh, O. “An Improved Algorithm for Matching Biological Sequences”. In: *Journal of Molecular Biology* vol. 162, no. 3 (Dec. 1982), pp. 705–708.

[Gui+21] Guidi, G. et al. “Parallel String Graph Construction and Transitive Reduction for De Novo Genome Assembly”. In: *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS). ISSN: 1530-2075. May 2021, pp. 517–526.

[Gwe20] Gwennap, L. “Tenstorrent scales ai performance”. In: *URL https://www. linleygroup. com/mpr/article. php* (2020).

[Göd+07] Göddeke, D. et al. “Exploring weak scalability for FEM calculations on a GPU-enhanced cluster”. In: *Parallel Computing.* High-Performance Computing Using Accelerators vol. 33, no. 10 (Nov. 1, 2007), pp. 685–699.

[Han04] Handelsman, J. “Metagenomics: Application of Genomics to Uncultured Microorganisms”. In: *Microbiology and Molecular Biology Reviews* vol. 68, no. 4 (Dec. 2004). Publisher: American Society for Microbiology, pp. 669–685.

[HC16] Heather, J. M. and Chain, B. “The sequence of sequencers: The history of sequencing DNA”. In: *Genomics* vol. 107, no. 1 (Jan. 2016), pp. 1–8.

[HH92] Henikoff, S. and Henikoff, J. G. “Amino Acid Substitution Matrices from Protein Blocks”. In: *Proceedings of the National Academy of Sciences of the United States of America* vol. 89, no. 22 (Nov. 1992), pp. 10915–10919.

[Hil+98] Hill, J. M. D. et al. “BSPlib: The BSP programming library”. In: *Parallel Computing* vol. 24, no. 14 (Dec. 1, 1998), pp. 1947–1980.

[Hue+20] Huerta, E. A. et al. “Convergence of artificial intelligence and high performance computing on NSF-supported cyberinfrastructure”. In: *Journal of Big Data* vol. 7, no. 1 (Oct. 16, 2020), p. 88.

[Hus19] Hustad, K. G. “Solving the monodomain model efficiently on GPUs”. MA thesis. http://urn.nb.no/URN:NBN:no-74080: University of Oslo, 2019.

[HVH18] Hammond, S., Vaughan, C., and Hughes, C. “Evaluating the Intel Skylake Xeon Processor for HPC Workloads”. In: *2018 International Conference on High Performance Computing & Simulation (HPCS).* 2018 International Conference on High Performance Computing & Simulation (HPCS). July 2018, pp. 342–349.

[Jyo+16] Jyothi, S. A. et al. “Measuring and Understanding Throughput of Network Topologies”. In: *SC ’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* SC ’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. ISSN: 2167-4337. Nov. 2016, pp. 761–772.

[Kep+15] Kepner, J. et al. “Graphs, Matrices, and the GraphBLAS: Seven Good Reasons”. In: *Procedia Computer Science.* International Conference On Computational Science, ICCS 2015 vol. 51 (Jan. 1, 2015), pp. 2453–2462.

[KG11] Kepner, J. and Gilbert, J. *Graph algorithms in the language of linear algebra.* SIAM, Jan. 2011.

[Kin+09] Kindratenko, V. V. et al. “GPU clusters for high-performance computing”. In: *2009 IEEE International Conference on Cluster Computing and Workshops.* 2009 IEEE International Conference on Cluster Computing and Workshops. ISSN: 2168-9253. Aug. 2009, pp. 1–8.

[Kno21] Knowles, S. “Graphcore”. In: *2021 IEEE Hot Chips 33 Symposium (HCS).* 2021 IEEE Hot Chips 33 Symposium (HCS). ISSN: 2573-2048. Aug. 2021, pp. 1–25.

[Kol+19] Kolodziej, S. P. et al. “The SuiteSparse matrix collection website interface”. In: *Journal of Open Source Software* vol. 4, no. 35 (2019), p. 1244.

[KSH12] Krizhevsky, A., Sutskever, I., and Hinton, G. E. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems.* Vol. 25. Curran Associates, Inc., 2012.

[Lan+01] Lander, E. S. et al. “Initial sequencing and analysis of the human genome”. In: *Nature* vol. 409, no. 6822 (Feb. 2001). Number: 6822 Publisher: Nature Publishing Group, pp. 860–921.

[Lan+19] Langguth, J. et al. “Towards Detailed Real-Time Simulations of Cardiac Arrhythmia”. In: *2019 Computing in Cardiology (CinC).* IEEE. 2019, Page–1.

[Lau21] Lauterbach, G. “The Path to Successful Wafer-Scale Integration: The Cerebras Story”. In: *IEEE Micro* vol. 41, no. 6 (Nov. 2021). Conference Name: IEEE Micro, pp. 52–57.

[Lev66] Levenshtein, V. I. “Binary Codes Capable of Correcting Deletions, Insertions and Reversals”. In: *Soviet Physics Doklady* vol. 10 (Feb. 1, 1966). ADS Bibcode: 1966SPhD...10..707L, p. 707.

[Mar+05] Margulies, M. et al. “Genome sequencing in microfabricated high-density picolitre reactors”. In: *Nature* vol. 437, no. 7057 (Sept. 2005). Number: 7057 Publisher: Nature Publishing Group, pp. 376–380.

[McC95] McColl, W. F. “Scalable computing”. In: *Computer Science Today: Recent Trends and Developments.* Ed. by Leeuwen, J. van. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1995, pp. 46–61.

[Mit+21] Mittal, S. et al. “A survey of SRAM-based in-memory computing techniques and applications”. In: *Journal of Systems Architecture* vol. 119 (Oct. 1, 2021), p. 102276.

[MLA10] Monakov, A., Lokhmotov, A., and Avetisyan, A. “Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures”. In: *High Performance Embedded Architectures and Compilers.* Ed. by Patt, Y. N. et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 111–125.

[MM88] Myers, E. W. and Miller, W. “Optimal Alignments in Linear Space”. In: *Bioinformatics* vol. 4, no. 1 (Mar. 1988), pp. 11–17.

[Mut+23] Mutlu, O. et al. “A Modern Primer on Processing in Memory”. In: *Emerging Computing: From Devices to Systems: Looking Beyond Moore and Von Neumann*. Ed. by Aly, M. M. S. and Chattopadhyay, A. Computer Architecture and Design Methodologies. Singapore: Springer Nature, 2023, pp. 171–243.

[NW70] Needleman, S. B. and Wunsch, C. D. “A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins”. In: *Journal of Molecular Biology* vol. 48, no. 3 (Mar. 1970), pp. 443–453.

[PSS08] Phillips, J. C., Stone, J. E., and Schulten, K. “Adapting a message-driven parallel application to GPU-accelerated clusters”. In: *SC ’08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. SC ’08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing. ISSN: 2167-4337. Nov. 2008, pp. 1–9.

[Reu+20] Reuther, A. et al. “Survey of Machine Learning Accelerators”. In: *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. 2020 IEEE High Performance Extreme Computing Conference (HPEC). ISSN: 2643-1971. Sept. 2020, pp. 1–12.

[Rod+99] Rode, C. K. et al. “Type-Specific Contributions to Chromosome Size Differences in Escherichia coli”. In: *Infection and Immunity* vol. 67, no. 1 (Jan. 1999), pp. 230–236.

[Sch+17] Schneider, V. A. et al. “Evaluation of GRCh38 and de novo haploid genome assemblies demonstrates the enduring quality of the reference assembly”. In: *Genome Research* vol. 27, no. 5 (May 2017), pp. 849–864.

[SDM11] Shalf, J., Dosanjh, S., and Morrison, J. “Exascale Computing Technology Challenges”. In: *High Performance Computing for Computational Science – VECPAR 2010*. Ed. by Palma, J. M. L. M. et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 1–25.

[Seb+20] Sebastian, A. et al. “Memory devices and applications for in-memory computing”. In: *Nature Nanotechnology* vol. 15, no. 7 (July 2020). Number: 7 Publisher: Nature Publishing Group, pp. 529–544.

[Sel+20] Selvitopi, O. et al. “Distributed many-to-many protein sequence alignment using sparse matrices”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’20. Atlanta, Georgia: IEEE Press, Nov. 9, 2020, pp. 1–14.

[SH05] Schloss, P. D. and Handelsman, J. “Metagenomics for studying unculturable microorganisms: cutting the Gordian knot”. In: *Genome Biology* vol. 6, no. 8 (Aug. 1, 2005), p. 229.

[SKK17] Sommer, L., Korinth, J., and Koch, A. “OpenMP device offloading to FPGA accelerators”. In: *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP).* 2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP). ISSN: 2160-052X. July 2017, pp. 201–205.

[SL12] Stranneheim, H. and Lundeberg, J. “Stepping stones in DNA sequencing”. In: *Biotechnology Journal* vol. 7, no. 9 (2012). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/biot.201200153, pp. 1063–1073.

[SNC77] Sanger, F., Nicklen, S., and Coulson, A. R. “DNA sequencing with chain-terminating inhibitors”. In: *Proceedings of the National Academy of Sciences* vol. 74, no. 12 (Dec. 1977). Publisher: Proceedings of the National Academy of Sciences, pp. 5463–5467.

[SPK13] Seshadhri, C., Pinar, A., and Kolda, T. G. “An in-depth analysis of stochastic Kronecker graphs”. In: *Journal of the ACM (JACM)* vol. 60, no. 2 (2013), pp. 1–32.

[SW81] Smith, T. F. and Waterman, M. S. “Identification of Common Molecular Subsequences”. In: *Journal of molecular biology* vol. 147, no. 1 (1981), pp. 195–197.

[TK06] Takizawa, H. and Kobayashi, H. “Hierarchical parallel processing of large scale data clustering on a PC cluster with GPU co-processing”. In: *The Journal of Supercomputing* vol. 36, no. 3 (June 1, 2006), pp. 219–234.

[Val90] Valiant, L. G. “A bridging model for parallel computation”. In: *Communications of the ACM* vol. 33, no. 8 (Aug. 1, 1990), pp. 103–111.

[Wan+16] Wang, Y. et al. “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.

[Wet22] Wetterstrand, K. A. *The Cost of Sequencing a Human Genome.* Genome.gov. Sept. 14, 2022. URL: https://www.genome.gov/about-genomics/fact-sheets/Sequencing-Human-Genome-cost (visited on 05/31/2023).

[WS98] Watts, D. J. and Strogatz, S. H. “Collective dynamics of ‘small-world’ networks”. In: *Nature* vol. 393, no. 6684 (June 1998). Number: 6684 Publisher: Nature Publishing Group, pp. 440–442.

[YBO20] Yang, C., Buluc, A., and Owens, J. D. *GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU.* 2020.

Papers

Paper I

iPUG for Multiple Graphcore IPUs: Optimizing Performance and Scalability of Parallel Breadth-First Search

Luk Burchard, Xing Cai, Johannes Langguth

Published in *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, Vol. 28, (2021, December), pp. 162–171. DOI: 10.1109/HiPC53243.2021.00030.

Abstract

Parallel graph algorithms have become one of the principal applications of high-performance computing besides numerical simulations and machine learning workloads. However, due to their highly unstructured nature, graph algorithms remain extremely challenging for most parallel systems, with large gaps between observed performance and theoretical limits. Furthermore, most mainstream architectures rely heavily on single instruction multiple data (SIMD) processing for high floating-point rates, which is not beneficial for graph processing which instead requires high memory bandwidth, low memory latency, and efficient processing of unstructured data.

On the other hand, we are currently observing an explosion of new hardware architectures, many of which are adapted to specific purposes and diverge from traditional designs. A notable example is the Graphcore Intelligence Processing Unit (IPU), which is developed to meet the needs of upcoming machine intelligence applications.

Its design eschews the traditional cache hierarchy, relying on SRAM as its main memory instead. The result is an extremely high-bandwidth, low-latency memory at the cost of capacity. In addition, the IPU consists of a large number of independent cores, allowing for true multiple instruction multiple data (MIMD) processing. Together, these features suggest that such a processor is well suited for graph processing.

The second and third authors were partially supported by the European High-Performance Computing Joint Undertaking under grant agreement No 956213 and the Research Council of Norway under contract 303404. The work has benefited from the Experimental Infrastructure project eX3, which is financially supported by the Research Council of Norway under contract 270053.

We test the limits of graph processing on multiple IPUs by implementing a low-level, high-performance code for breadth-first search (BFS), following the specifications of *Graph500*, the most widely used benchmark for parallel graph processing. Despite the simplicity of the BFS algorithm, implementing efficient parallel codes for it has proven to be a challenging task in the past. We show that our implementation scales well on a system with 8 IPUs and attains roughly twice the performance of an equal number of NVIDIA V100 GPUs using state-of-the-art CUDA code.

Contents

I.1 Introduction

In the recent years, it has become increasingly clear that energy poses the ultimate limit to processor performance, and that in order to attain higher energy efficiency, specialized processor architectures are needed. As a consequence, in the last decade, the GPU, which offers higher performance and energy efficiency for many scientific computing and machine learning workloads, has made its successful entrance to the world of high performance computing.

Now, with the end of Moore's law getting closer and closer [Lei+20], a large number of additional architectures, many of them geared towards machine learning applications, have been developed. This *"Cambrian explosion of novel computer architectures"* [HP19] has yielded several mature processors which became available in the last two years. Among the first such processors is the Graphcore Intelligence Processing Unit (IPU), which consists of a large number of independent tiles, each having a multiple instruction multiple data (MIMD) core and a small amount of SRAM memory. With more than 1000 such cores, the IPU provides a massive amount of low precision FLOPS. Furthermore, the absence of a cache hierarchy makes data access extremely fast and efficient. This also makes the IPU well-suited for highly irregular workloads such as graph algorithms. Our goal is to study the practical performance benefits that can be attained from this architecture.

Two IPU versions have been released so far, the GC2, which became first available in 2018, and the GC200 released in late 2020. The GC200 improves upon the GC2 w.r.t. the number of cores, FLOPS, and SRAM memory. In this paper, we focus on the second-generation hardware.

For testing the graph processing performance of a new architecture, the *Graph500* [Mur+10] breadth-first search (BFS) benchmark offers an ideal starting point. While there are more complex graph algorithms, efficient parallelization of BFS remains challenging. It requires irregular communication patterns and creates unbalanced workloads, thereby capturing the principal challenges of implementing parallel graph algorithms. Since 2010, *Graph500* has collected BFS performance results for a wide range of hardware platforms and instance sizes, making it by far the most studied parallel graph problem. Highly optimized codes have been presented for both CPU and GPU systems [Che+20; Wan+16; YFG13]. Furthermore, graph-specialized hardware accelerators were proposed [Son+18]. This work facilitates a fair comparison between IPUs and GPUs using only preexisting benchmark parameters.

While the *Graph500* specifications allow for running small instances, the memory of a single IPU is typically too small for most applications of practical interest. And while it is possible to access data from the attached DRAM or an external network connection on the IPU, the best-performing solution is to scale out and use as many IPUs as needed for the problem to fit in the combined SRAM.

Our work builds upon the iPUG BFS code [Bur+21] which was written for a single GC2 IPU. In this paper we present a code that is capable of scaling to multiple IPUs, as well as using the newer GC200 IPU. The main challenges to overcome are grounded in the fact that the IPU was not originally designed for running graph algorithms. First, the data structures are not well-suited for maximizing data locality in BFS. In order to obtain high performance, we have to design a manual data distribution. The optimum distribution also changes between the two IPU versions. Second, IPU communication follows static patterns, and has to be planned at compile time. To get around this limitation, a 2-competitive solution was designed for iPUG [Bur+21]. However, when moving to the multi-IPU scenario, communication becomes much more costly since the device-to-device links are considerably slower than the core-to-core communication inside the IPU.

We present our solutions to these problems in the following sections. While the new multi-IPU code is capable of running BFS on larger graphs, the primary goal of this paper is to outline techniques that can serve as a model for the implementation of advanced graph algorithms in the future, many of which use BFS as a subroutine. These include graph centralities and other algorithms used in the analysis of social networks, such as triangle counting, clustering, and matching. Thus, our paper makes the following contributions:

1. We present the first implementation of a graph algorithm on the new GC200 IPU which scales to multiple IPUs.

2. We present an optimization that enables sparse communication in a dense framework, allowing us to implement the sparse communication required for graph algorithms in the communication model of the IPU.

Table I.1: Key architectural features of GC2 and GC200 IPU.

Chip	GC2	GC200
Number of tiles	1216	1472
Number of threads	7296	8832
Memory per tile	256 KB	624 KB
Total SRAM memory	311 MB	918 MB
Memory bandwidth	46.6 TB/s	46.9 TB/s
Aggregate tile-to-tile bandwidth	7.78 TB/s	7.83 TB/s
Total chip-to-chip bandwidth	320 GB/s	320 GB/s
Clock frequency	1.6 GHz	1.33 GHz
FP32 compute	31.1 TFLOPS/s	62.5 TFLOPS/s

3. We investigate the performance of our implementation on a cluster of 8 IPUs and compare it to state-of-the-art GPU codes. The results show that our iPUG code is highly competitive, delivering more than twice the performance of a cluster of 8 V100 GPUs in the *Graph500* benchmark.

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

I.2 IPU Hardware

The Graphcore IPU consists of a large number of independent units called *tiles*. Each tile consists of a core and a small amount of SRAM memory. Each core runs six concurrent threads in a fine-grained *temporal multithreading* scheme. Unlike simultaneous multithreading, which is commonly used in modern CPU and GPU designs, IPU threads are scheduled consecutively in a fixed order. For that reason, the design is also referred to as a *barrel processor*. In general, IPU instructions, including loads and stores from the local tile memory, take exactly 6 cycles. Thus, individual threads do not experience latency since they execute one instruction per cycle in which they are scheduled.

The tiles are organized into *islands* which themselves are grouped into *columns*. Together, the columns form the IPU, as illustrated in Figure I.1. The number of cores depends on the IPU model. Table I.1 gives an overview of the most important features of the GC2 and GC200 IPUs. In previous work, architectural details of the GC2 IPU were studied and benchmarked exhaustively [Jia+19]. Since all cores can read from memory concurrently, the aggregate memory bandwidth is much higher than that of CPUs or GPUs. However, data that is not local to a core must be moved between the tiles. A tile is capable of sending 4 bytes and receiving 4 bytes per cycle, which amounts to amounts to 5.3 GB/s or 7.83 TB/s for all 1472 cores of the GC200. The network that connects the

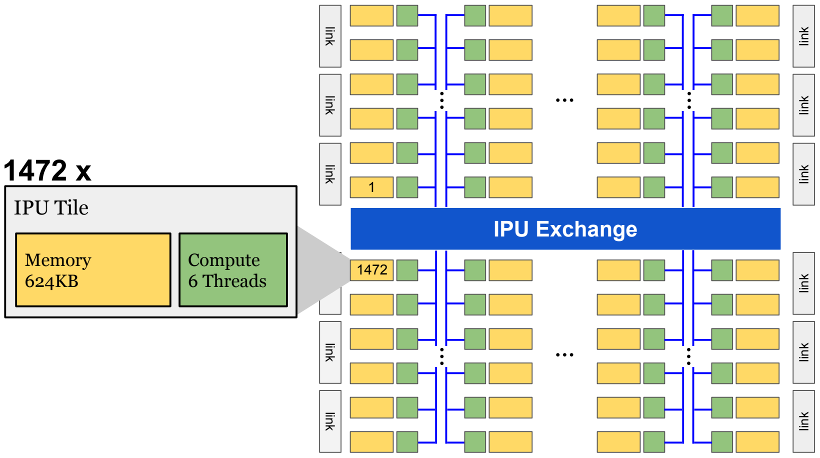

Figure I.1: Tile layout on the GC200 IPU processor.

cores inside the IPU is called the IPU exchange. The GC200 IPU can also access DRAM memory at a speed of about 20 GB/s. However, in this paper we only study problems that are placed entirely in the SRAM memory.

Between the IPUs, data is transferred via the IPU-Link, which performs both intra-node and inter-node communication. It thus corresponds to both PCIe and Infiniband in CPU/GPU systems (or alternatives such as NVIDIA NVLink and CRAY Shasta). Each IPU has 10 IPU-links with a total bandwidth of 320 GB/s. Pairs of IPUs are connected with 12 links, among themselves, which amount to a bandwidth of 192 GB/s. This leaves 8 links to connect to other IPUs. These connections use double-link cables. Thus they operate at 64 GB/s. Up to 32 such pairs can be connected in a ladder configuration with a bisection bandwidth of 128 GB/s. See Figure I.2 for an example. The ladder can be closed to form a torus, which doubles the bisection bandwidth. Multiple such groups of 64 IPUs, which are referred to as PODs, can be connected via an additional interface called Gateway Link, although this is not being used in this paper. The network is the same for both IPU versions. Note that a single IPU has 150 W TDP, which is approximately half of a competitive GPU. Thus, w.r.t. power, each IPU pair is comparable to one powerful GPU, such as the NVIDIA V100 or A100.

There are multiple ways to program the IPU. Standard machine learning workloads can interface via *PyTorch* [Pas+19] or *TensorFlow* [Aba+16], which is the primary intended usage. Their IPU implementation is built upon the *Poplar* framework which follows the dataflow model. A layered graph is used to structure programs, where vertices in each layer alternate between representing states stored in multidimensional arrays called tensors and subroutines that perform the transition from one state to the next. These subroutines are called *codelets*. Each vertex in a computation layer has such a codelet, and all codelets

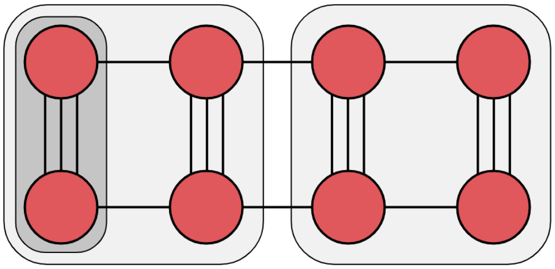

Figure I.2: The 8 IPUs used in this paper are configured in a ladder interconnect. A single IPU is denoted as a red circle, and the lines represent double IPU-links with a capacity of 64 GB/s. The dark grey area shows a double IPU configuration connected via three 64 GB/s lanes. The light gray area represents a physical 4 IPU blade. The two blades are connected with two 64 GB/s cables.

within one layer can be executed in parallel without race conditions. Data is distributed in a bulk-synchronous parallel (BSP) [Val90] manner. The size of the tensors and thus the communication in each step is determined at compile time.

I.3 Implementation

BFS is a fundamental graph algorithm, finding the shortest path from a source node to each other node on unweighted graphs. Due to the high-level definition, various implementations of this algorithm exist. As the IPU is designed for machine learning and artificial intelligence computing, linear algebra naturally fits the design of the IPU. Therefore, our IPU code uses a linear algebra version of the BFS algorithm, described by Kepner et al. [KG11]. The linear algebra formulation of one level-synchronous BFS step is $A^T x_t = x_{t+1}$. A is the adjacency matrix representation of the graph $G(V, E)$, where $\mathbf{A} : \mathcal{R}^{|V| \times |V|}$, and $a_{i,j} = 1$ if $(v_i, v_j) \in E$ else 0. Here, x_t is the current frontier, where $x_t(k)$ expresses that node k is in the current frontier containing the level of nodes to be visited in the current iteration, and x_{t+1} is the next iteration level to be visited. The algorithm terminates when x_{t+1} does not contain any unseen nodes.

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.

We use a 2D decomposition to split the adjacency matrix into a $p \times p$ checkerboard pattern, similar to other distributed memory implementations [Bul+17; Yoo+05], and we let $\mathbf{A}(i, j)$ denote a partition. Moreover, we want to maximize the size of each partition while having at most one partition mapped to a tile. Therefore, we choose $p = \lfloor\sqrt{P}\rfloor$. This partitioning makes each tile only responsible for a subset of the outgoing edges of a vertex. After all partitions have explored their locally adjacent neighboring nodes, these results need to be merged into a coherent global state indicating if a vertex was visited and by which node. This horizontal reduction on the IPU requires the exchange of data using the IPU exchange fabric, as we do not have a global memory space.

To balance our computation, we randomly permute the adjacency matrix, which on average leads to a good load balance, even compared to graph partitioners [Pan+17].

In Algorithm 1, we describe a bulk synchronous parallel (BSP) and level synchronous BFS algorithm based on linear algebra primitives. Like in the Poplar framework, we do not need to describe communication explicitly, as data-exchange is handled by the compiler by describing input and output tensor regions for a partition on each tile.

Our algorithm can be divided into two major phases:

1. **Expansion**: Each processor $P_{i,j}$ receives a part of $\mathfrak{Q}$, i.e. the frontier queue, and uses the locally stored $\mathbf{A}(i, j)$ to expand the vertices from $\mathfrak{Q}$. The output is a new intermediate status array dense vector $\mathfrak{S}\mathcal{A}(i, j)$

2. **Fold**: A reduction that uses the intermediate output $\mathfrak{S}\mathcal{A}$ from the expansion phase to update the parent array $\mathbf{b}$. The reduction takes place along the rows of the partitions into the new frontier $\mathfrak{Q}$. Each tile in the fold phase owns the result of $\frac{|V|}{p^2}$ vertices. Each tile generates a small queue as one piece of $\mathfrak{Q}$.

Our algorithm implements an SpMSpV algorithm, which receives the input frontier $\mathfrak{Q}$, split it into p parts that are shared across each column. These p partitions in $\mathfrak{Q}$ are themselves made out of smaller p partitions. As $\mathfrak{Q}$ is a queue structure, the data can be viewed as sparse data. The fold phase linearly maps the tiles along with the columns and merges results across the rows. The linear mapping is defined through $(i - 1) * p + j$. The algorithm outputs $\mathbf{b}$ which stores the direct parent on the path the source node s. The global barrier symbolizes data exchange and global synchronization. Single tiles working longer before reaching the barrier create imbalance.

I.3.1 Using multiple IPUs

The Poplar framework transparently allows us to expand our algorithm to multiple IPUs thereby increasing the processor count P. As with a single IPU, the compiler automatically generates exchange code for intra-IPU and inter-IPU communication. The user can control data exchange only via an explicit mapping of the partitions to the tiles, since the mapping determines both global and local communication patterns.

Algorithm 1 Parallel BFS algorithm.

```
1: $p \leftarrow \lfloor \sqrt{P} \rfloor$
2: $q_s \leftarrow \frac{|V|}{p^2}$
3: $\mathfrak{Q} : \mathbb{R}^{p \times p \times q_s} \leftarrow \{s\}$    ▷ Tiny queues
4: $\mathcal{SA}(:,:,:) : \mathbb{R}^{p \times p \times \frac{|V|}{p}} \leftarrow 0$
5: $\mathbf{b}(:) : \mathbb{R}^{|V|} \leftarrow 0$
6: while $\mathfrak{Q} \neq \emptyset$ do
7:     for all processor $P_{i,j}$ in parallel do
8:         for $q \in \mathfrak{Q}(i,:)$ do
9:             for $v_{in} \in q$ do
10:                for $v_{adj} \in adj(\mathbf{A}(i,j), v_{in})$ do
11:                    $\mathcal{SA}(i,j,v_{adj}) \leftarrow v_{in}$
12:                end for
13:            end for
14:        end for
15:        Global BSP Barrier    ▷ End ComputeSet
16:        $\mathfrak{Q}(:,:) \leftarrow \emptyset$
17:        $l \leftarrow$ Linearmapping$(i,j)$
18:        for $r \leftarrow lq_s$ to $(l+p)q_s$ do
19:            if $\mathbf{b}(r) = 0$ then
20:                for $j_f \leftarrow 1$ to $p$ do
21:                    $v_{in} \leftarrow \mathcal{SA}(i : j_f, jq_s : (j+1)q_s)$
22:                    if $v_{in}$ then
23:                        $\mathbf{b}(r) \leftarrow v_{in}$
24:                        Union()$\mathfrak{Q}(i,j)$ ,$\{r\}$)
25:                        break
26:                    end if
27:                end for
28:            end if
29:        end for
30:        Global BSP Barrier    ▷ End ComputeSet
31:    end for
32: end while
```

We continue to use a 2D grid but split it by 1D cuts along row-blocks to the available IPUs. Therefore, during the fold phase, as all communication occurs row-wise, and thus uses only the fast IPU exchange interconnect.

All communication for the expansion phase happens column-wise, where the input frontier $\mathfrak{Q}$ is sent to all rows in their respective parts. As the 2D grid is distributed across the IPUs, communication using the slower IPU-Link interconnect is required here. Still, the 1D split between IPUs means that we communicate between the IPUs only in one phase, as opposed to a naïve mapping across P where we do so in both phases. In the following we further decrease the required communication.

I.3.2 (Sub)-queue packing

Sending data inside the IPU using the high-bandwidth IPU Exchange is much faster than between IPUs. Packet sizes and transmission patterns do not significantly affect the on-chip bandwidth achieved. Due to a much smaller interconnect bandwidth when scaling to multiple IPUs, we are forced to use a more bandwidth-efficient mapping and algorithmic optimizations. In the following section, we describe our "(sub)-queue packing" technique which saves a significant amount of data sent in inter-IPU transmissions.

The expansion phase receives a list of queues from the p^2 tiles in the fold phase. These small queues build a bigger queue $\mathbf{\Omega}$ which gets communicated along each column. A tile in the expansion phase will receive p of those small queues from the fold phase. Our queues have the property that the values they are always densely packed. We know the maximum number of values that can be placed in the queue S originating from a folding tile, which is equal to $cap(S) = |V|/p^2$ with values in the queue $size(S) \leq cap(S)$. Therefore, the final frontier exchanges into each partition is defined as $size(\mathbf{\Omega}(i, :, :) = size(S) * p$.

The activation queues get copied across the IPU interconnects in our algorithm, making up a significant portion of the runtime. This is illustrated in Figure I.6. As only $size(S)$ is required, we could optimally save $cap(S) - size(S)$ data from transmission. However, this is not trivial as the communication code is generated at compile-time, and there is no possibility to shorten the exchange phases by omitting data. We could generate an exchange program for all tiles sending queues with data sizes from 0 to $cap(S)$, but doing this in all possible combinations would lead to in total $(cap(S) + 1)^{p^2}$ exchange programs, which is not tractable. Instead, we assume that our partitioning resulted in fairly good load balancing, which means that the sizes of the queues are similar. If all queues are assumed to be of the same size, we can choose $max(size(S)) \in Q$ to be the maximum size bound of all queues in the current phase. We assume that the number of values to be transmitted per partition is approximately the same.

This optimization needs a max-reduction phase for all sizes of S to determine the maximum size to be transmitted prior to the expansion phase. We interleave the max-reduction with the fold phase for checking if the next frontier is empty, making it close to zero-cost. If we generate an exchange program for each possible queue size, we would have to compile $cap(S)$ programs which is still too high for bigger graph instances where $cap(S)$ can be in the order of thousands. In addition, a complex compute graph leads to more memory usage and a significantly increased compile time. Therefore, we introduce a 2-competitive online algorithm for deciding upon the data transmission size.

To limit the amount of exchange code generated, we only compile exchange phases that transmit $\{1, 2, 4, ..., 2^n, cap(S)\}$ sized queues. We always choose the next bigger transmission size which fits our data. Choosing the possible exchange sizes in an exponential manner decreases the number of exchange programs generated to $\mathcal{O}(\log(cap(S)))$. In the worst case, assuming a well-balanced graph, we choose a transmission size bigger than our current maximum frontier with a difference of $2n - (n + 1)$ which makes a worst-case limit of

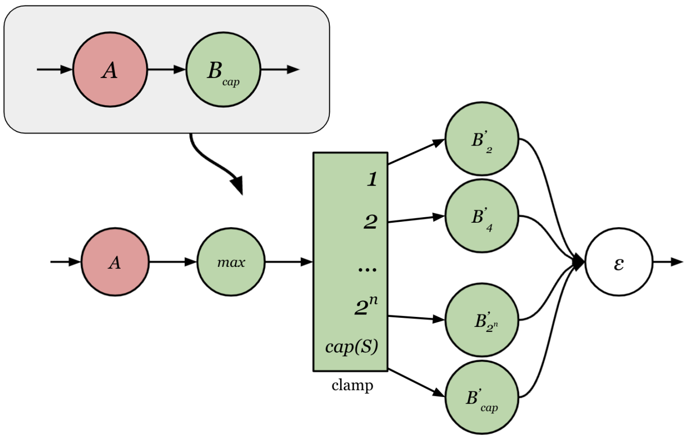

Figure I.3: Switching between $log(cap(S))$ different exchange programs which heuristically decrease the amount of data to be transmitted between tiles across IPUs. Here A denotes the BFS fold phase generating the new frontier used in B_{cap}. We find the biggest queue which needs to be allocated and use this as our exchange size for all other queues.

$\lim_{x\to\inf} 2n-(n+1) = 2$ times the amount of the necessary data to send. However, for highly unbalanced graphs, it is possible that no saving is achieved. Figure I.3 shows the inflation of the compute graphs with the max-reduction and the following switching of the (sub)-queue packing optimized exchange phases for the expansion phase. At the start of the BFS program, we always choose a transmission size of one as only one node is active, and thus no queue can hold more values.

I.4 Experiments

For our experiments, we used a system of 8 GC200 Graphcore IPUs spanning two IPU-POD4 blades, connected with 2 IPU lanes. Technical details are given in Table I.1. We compile our code with the *Poplar* 2.1.0 SDK and *popc*, i.e. the latest releases at the time of writing.

For comparison with the GPU, we use the Gunrock framework [Wan+16], on up to 8 Volta V100 GPUs inside a DGX-2 system. Unlike previous work [Bur+21], we omit comparison with the CPU. The reason is that the CPU typically exceeds GPU and IPU performance only on high-diameter graphs where the available concurrency is too low for the GPU or IPU to benefit from their high number of parallel threads. On the other hand, reported results for *Graph500* show that CPUs perform much weaker on the Kronecker graphs. Thus, the GPU versus IPU comparison is more relevant since both devices excel on high-concurrency, low-diameter graphs.

The current version of Gunrock is 1.0, which we use for single-GPU results.

Multi-GPU capabilities were discontinued after version 0.5, so we use this version for multi-GPU. For compatibility, we are using CUDA version 10.0.130.

The DGX-2 system has 16 Volta V100 GPUs, connected via NVlink to a crossbar switch. This allows concurrent communication at 300 GB/s between every pair of GPUs. Furthermore, the NVlink-connected GPUs have a slightly higher clock frequency than their PCIe counterparts. We use a one-to-one ratio between GPUs and IPUs as the basis for our comparison. As mentioned in Section I.2, based on power consumption, it is also reasonable to compare two IPUs to one GPU. The same is true w.r.t. rackspace and approximate price.

For performance measurements, we follow the guidelines set by the *Graph500* benchmark [Mur+10], which offers a wide range of comparison results. However, since *Graph500* uses only a single type of graph obtained from a Kronecker graph generator with initiator parameters $\{A, B, C, D\} = \{0.57, 0.19, 0.19, 0.05\}$, we supplement the generated graphs with instances from SuiteSparse [Kol+19]. These instances were selected to match previous work [YBO20], although some graphs are too large to run on IPUs. Table I.2 lists both types of instances, along with their size and diameter.

For the generated instances, in names such as *kron20_16* the first number refers to the number of vertices as a power of 2, while the second number shows the *edge factor*. We use a *Graph500* compliant edge factor of 16, as well as higher values to investigate edge scaling.

We included graphs from the SNAP group which come from the Stanford Large Network Dataset Collection [LK14] and represent real-world networks. The *wiki-topcats* graph represents the most popular hyperlinks between top Wikipedia pages. The *soc-Pokec, com-Youtube, loc-Gowalla*, and *com-Orkut* graphs represent online social networks, where a friendship is represented as an edge. We also included *roadNet-CA* and *belgium_osm* as road network graphs, where road intersections are vertices connected by edges representing roads. We include other real-world graphs: *coAuthorsDBLP, coPapersDBLP, coAuthorsCiteseer*, and *coPapersCiteseer* which represent online collected co-authroships and citation networks. The *preferentialAttachment* graph was generated by a synthetic graph generator. The *delaunay_n*{*scale*} graphs were created from a triangulation of points in a plane. The *kron_g500-logn*{*scale*} graphs are generated like the *Graph500* instances using the same initiator parameters, but despite their name have an edge factor of 48. They are included to enhance the reproducibility of our results. Furthermore, we included the *hollywood-2009* graph with actors as vertices joined together by movies as edges. Finally, the *webbase-1M* graph connects websites together through hyperlinks.

The *Graph500* benchmark splits timings into a preparation kernel, and the BFS graph traversal itself. We follow this specification and report the timings of the graph traversal starting from the time when the search key is sent to the device and ending with the termination of the last BFS round. Performance is measured in traversed edges per second (TEPS). Following previous work [LH15; YBO20], we count TEPS using the directed edges, excluding self-loops and duplicated edges. In the following, we present a series of experimental results that investigate the performance of our implementation.

Table I.2: Datasets split into synthentically generated Kronecker graphs and SuiteSparse instances. Graph diameters marked with a (*) are approximated with 10k random BFS walks.

SuiteSparse			
graph	diameter	$\|V\|$	$\|E\|$
com-Orkut	10	3.072.441	117.185.083
webbase-1M	28	1.000.005	3.004.970
coAuthorsDBLP	24	29.9067	977.676
delaunay_n17	167	131.072	393.176
delaunay_n18	228	262.144	786.396
delaunay_n19	319	524.288	1.572.823
kron_g500-logn18	6	262.144	10.583.222
kron_g500-logn19	7	524.288	21.781.478
kron_g500-logn20	7	1.048.576	44.620.272
coAuthorsCiteseer	33	227.320	814.134
coPapersDBLP	23	540.486	15.245.729
coPapersCiteseer	34	434.102	16.036.720
citationCiteseer	36	268.495	1.156.647
preferentialAttachment	7	100.000	499.985
belgium_osm	1987	1.441.295	1.549.970
roadNet-CA	865	1.971.281	2.766.607
hollywood-2009	12	1.139.905	57.515.616
soc-Pokec	14	1.632.803	22.301.964
wiki-topcats	11	1.791.489	25.447.873
wikipedia-20060925	13	2.983.494	35.065.981
com-Youtube	24	1.134.890	2.987.624
loc-Gowalla	16	196.591	950.327
Generated			
graph	diameter	$\|V\|$	$\|E\|$
kron20_16	8	1.048.576	15.700.872
kron21_16	8	2.097.152	31.770.104
kron22_16	8	4.194.304	64.153.736
kron23_16	9	8.388.608	129.340.089
kron21_32	7*	2.097.152	61.720.652
kron21_64	7*	2.097.152	118.598.465
kron21_128	7*	2.097.152	224.855.774
kron21_256	6*	2.097.152	536.847.237
kron20_64	7*	1.048.576	67.104.118
kron20_128	6*	1.048.576	134.208.329
kron20_256	6*	1.048.576	268.416.593
kron20_512	6*	1.048.576	536.833.050

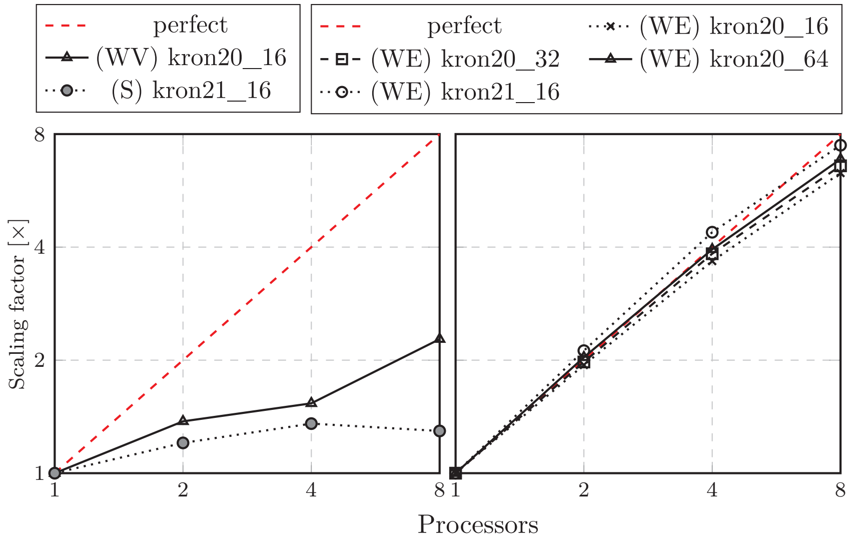

Figure I.4: Strong Scaling (S), Weak Vertex Scaling (WV), and Weak Edge Scaling (WE).

I.4.1 Scaling

We run weak and strong scaling experiments on the IPU and GPU systems with 1, 2, 4, and 8 devices respectively. As suggested by the *Graph500* specification, we take the average from a sample of 64 starting nodes for the weak scaling results. Furthermore, we are using the Kronecker graph instances for scaling experiments as it is possible to set the scaling parameter of the problem instance. Moreover, the Kronecker graph generator allows us to define two parameters, the edge factor, and the vertex scale. Henceforth, we are adjusting both parameters, which will lead to a larger instance. For strong scaling, we are using a *kron21_16* as it is the biggest instance that fits on a single IPU; for the GPU, we use a *kron25_16*. In our weak scaling experiments, we start with *kron20_16* for the vertex scaling experiment and four different instances for the edge scaling experiment.

Results

In Figure I.4 we show strong and weak scaling results. On 1, 2, 4, and 8 IPUs we archive 55.6, 67, 75.4, and 72 *GTEPS* on *kron21_16* for strong scaling respectively. When scaling to multiple IPUs the tiles get less saturated and more imbalanced. Combined with the rather slow interconnect bandwidth, this leads to a flat scaling curve. However, our main focus is on larger instances, and therefore we study weak scaling behaviour. As shown in Figure I.4, we archive better results with weak edge scaling, where we observe a superlinear speedup. This can be explained by the decreased overhead for each vertex being visited.

We archive up to 1.03 $TTEPS$ on 8 IPUs for the *kron20_512* instance. For the weak vertex scaling, which is the basis of the *Graph500* benchmark, we go from *kron20_16* to *kron23_16*, as it is the largest graph we can fit on 8 IPUs. In order to fit the graph onto the IPUs, we needed to use $int16$ datatypes, which at the time of writing lead to redundant instructions, thus decreasing the performance. We expected this issue will disappear in future compiler versions.

I.4.2 Overall Performance

We compare the performance of our implementation by sampling 64 BFS runs from connected vertices and taking the average of these results. Here, we collect data on all of our graph instances. Furthermore, we compare our results to a top-down BFS and a direction optimizing BFS (DOBFS). We include both results, as previous work [Pan+17] has shown that DOBFS is effective on one GPU but does not scale well.

The top row of Figure I.5 shows the SuiteSparse graph instances, mainly containing real-world graphs. We can observe that the IPUs outperform the GPUs in most cases. The GPU attains its best performance with the DOBFS implementation on a single device, while the GPU top-down implementation performs consistently worse but scales better.

We also show Kronecker graph performance in the second row of Figure I.5. DOBFS shows the biggest advantage over our implementation with higher density graphs as the optimization allows for an exponential increase in performance with linearly increasing runtime. However, compared to the top-down GPU implementation our IPU implementation is consistently faster by a factor of 3× to 4.6×.

I.4.3 Runime Analysis

We aim to give an in-depth overview of the inner workings of our algorithm and show the execution and inner timings of our implementation. Each iteration of our algorithm contains two major phases: (1) the expansion phase, in which the current frontier is distributed to discover the future nodes to be visited, and (2) the fold phase, which merges all discoveries from the expansion phase and reduces them into a single vertex. Finally, the fold phase generates the frontier for the next expansion phase.

We dissect the timings from weak and strong scaling runs on Kronecker graphs. For the strong scaling results, we used a *kron21_16*. We started with a *kron20_16* using a single IPU for the weak scaling experiment and scaled up to a *kron23_16* on eight IPUs. The results were generated with the PopVision™ graph analyzer suite of tools used to extract profiling information generated by the Poplar framework during the compilation and execution phase. All results make use of all of our optimizations, such as (sub)-queue packing.

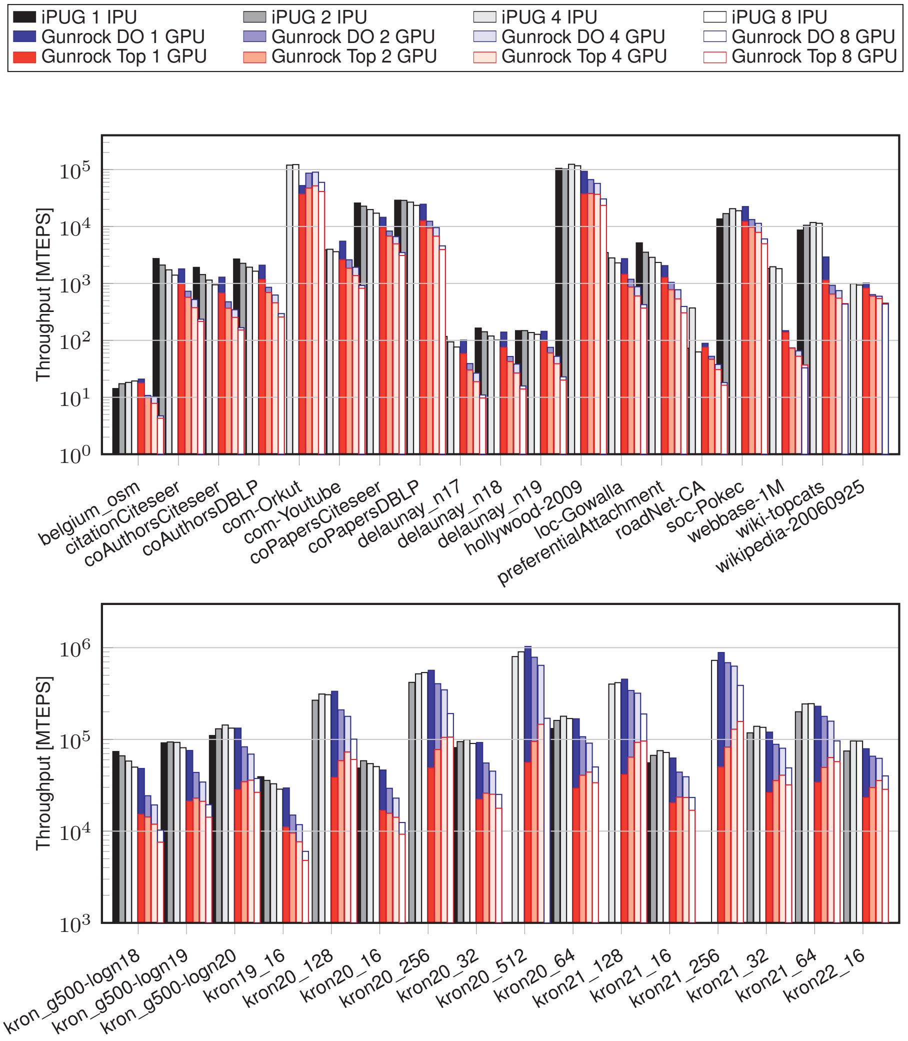

Figure I.5: Performance of the SuiteSparse instances (top) for the IPU M2000 and the V100 DGX-2 using iPUG and Gunrock, respectively. The lower plot shows performance numbers of our synthetic Kronecker graphs, combined with Kronecker graphs from the SuiteSparse collection. GPU results are stacked for scaling factors. The lower result is in the front and the higher result is stacked into the back.

Results

Figure I.6 shows that the compute time to solve the *kron21_16* instance decreases by adding IPUs. The communication and execution time of the fold phase

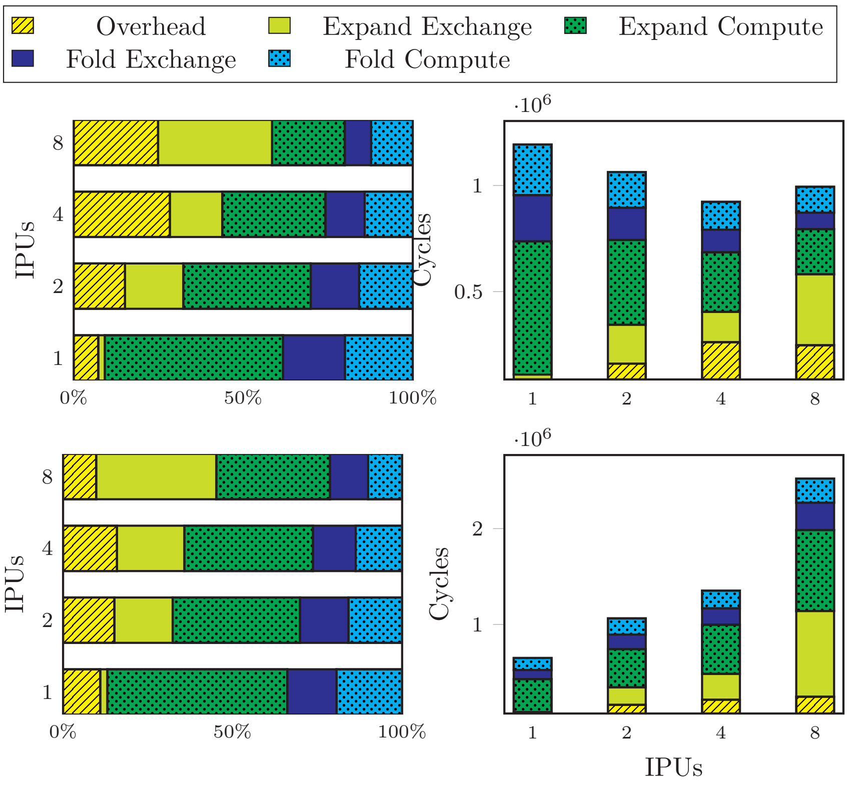

Figure I.6: The graphs show the timings of the fold and expansion phase divided into data exchange between tiles and computation and all additional cycles. The first two charts are strong scaling results on an kron21_16. The first chart shows the total runtimes of each stage, the second chart gives the relative share of the steps taken for each IPU scale size. The last two charts depict weak vertex scaling results from a kron20_16 to a kron23_16. The third chart shows weak scaling execution times of each phase on each and the last chart the shares each phase takes executed on N IPUs respectively.

decreases as the communication and computation only take place within one IPU. The computing time in the expansion phase decreases every round by 36% to 24%. The communication increases by 8× going from one to two IPUs. However, this trend does not continue linearly, as the communication time reduces when going to 4 IPUs, only to increase by 2× going from 4 to 8 IPUs. In addition to the expand and fold phase, the program has further overhead, such as globally switching the BSP-supersteps, the reduction determining if the algorithm should continue, and the reductions from our optimizations. We can observe that the additional compute and communication time within this overhead category increases to up to 28% of the global runtime. The overhead increase is due to the need for five additional global communication steps between the IPUs.

For the weak edge-scaling experiments, we can observe that the cycle overhead does not increase much when scaling from 1 to 8 IPUs. Like in the strong scaling experiment, both fold communication and exchange decrease. The communication phase of the expansion phase constantly increases from 2% to 35% and, together with the compute phase, the expansion phase takes more than 68% of the total time.

I.4.4 (Sub)-queue packing

We tested our (sub)-queue packing optimization on three representative graphs. First, we choose a low diameter generated Kronecker graph with an edge factor of 16 and a scale of 2^{21} nodes, and then a real-world graph problem from the SuiteSparse representing a co-author network with a medium diameter. As the last graph, we choose a high diameter Delaunay graph with an average edge factor of 3.

We compare the saved amount of data sent to the previous dense message size transmitted for each BFS level in all graphs instances. We additionally show the best possible communication cost calculated from all values in the frontier compared to all activations over time. We also compare a well-balanced Delaunay graph with an unbalanced Delaunay graph to show the impact of load-balancing on this optimization.

Each BFS execution started from the largest node for each graph. As execution is deterministic, we only executed a single run for each setup. For the Delaunay graphs, there is no single largest degree as the graph is made from a triangulation of points.

Results

In Figure I.3 we compare the data (sub)-queue packing to our previous method of transmitting data on the IPU. A general speedup can be seen for the initial phases of the Kronecker graphs, which in the explosion phase converge to dense transmission. However, the optimal data transmission size is still close to ours. For the real-world coPapersCitiseer network, we can observe the same effect in the explosion phase but recovering in the thinly expanding graph phases. The Delaunay graphs perform the worst as the actual needed data send size is small, but slight load imbalances cause the transmission size to increase.

We observe decreasing effectiveness when considering strong scaling from a single to eight IPUs. This trend can be seen in all graphs and can be explained by the reduced queue capacities. Small partition queues are more likely to become imbalanced and have fewer sizes to choose from, which prevents queue savings through tiny imbalances.

Overall we can observe a reduction in data size of 78.23% for the Kronecker graph on a single IPU, and 73% for eight IPUs. For the real-world co-author network, we observe an overall data reduction between 91% one IPU and 80%

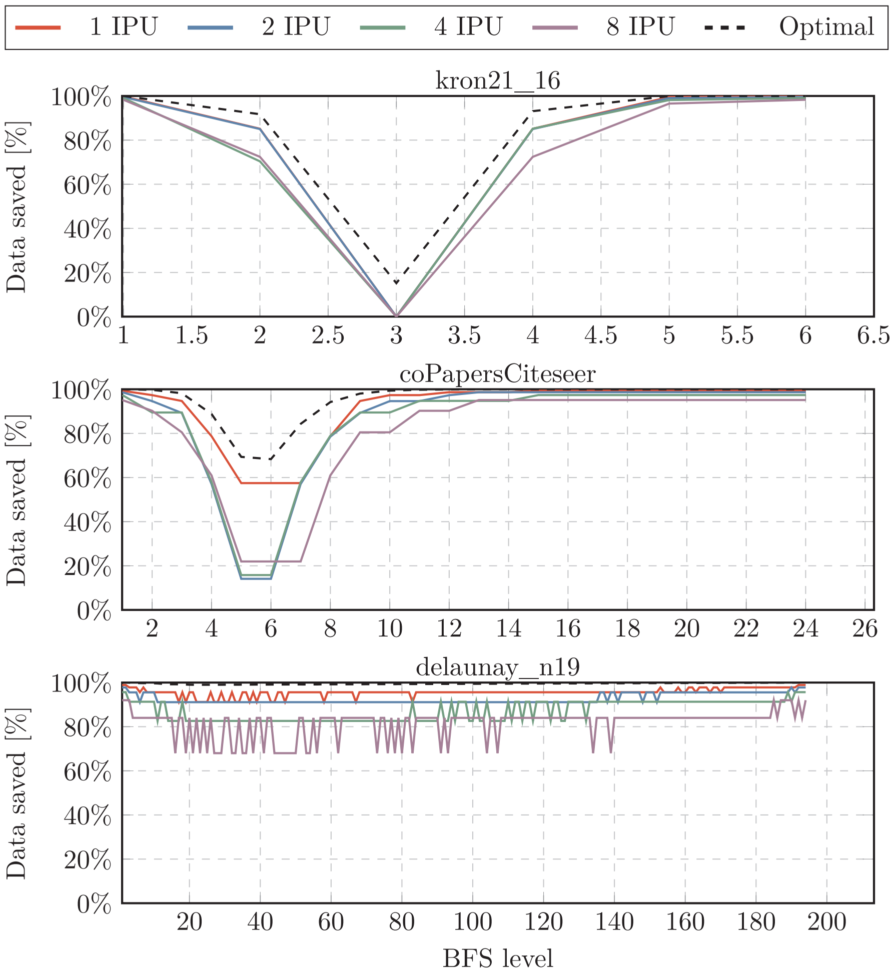

Figure I.7: The graphs show the possible amount of data saved compared to dense communication of the expansion phase input queues. Each BFS level displays the potential saving that could have occurred. The optimal size is the number of values in the current frontier by counting all values over time in the frontier. Kronecker graphs are with our heuristic close to optimal. In contrast, Delauney graphs have the highest potential to save data than dense communication but show the widest gap to the optimal transmission size. Real-world co-author networks perform similar to the Kronecker graphs but are not as effective in the explosion phase.

for eight IPUs. Depending on balancing, we can reach from 95% to 81% for eight IPUs or 80% to 36% for eight IPUs with bad load balancing, respectively.

In Figure I.8, we plotted the effect of unbalanced and balanced graphs on our optimization. Delaunay graphs generate many small frontiers. Henceforth,

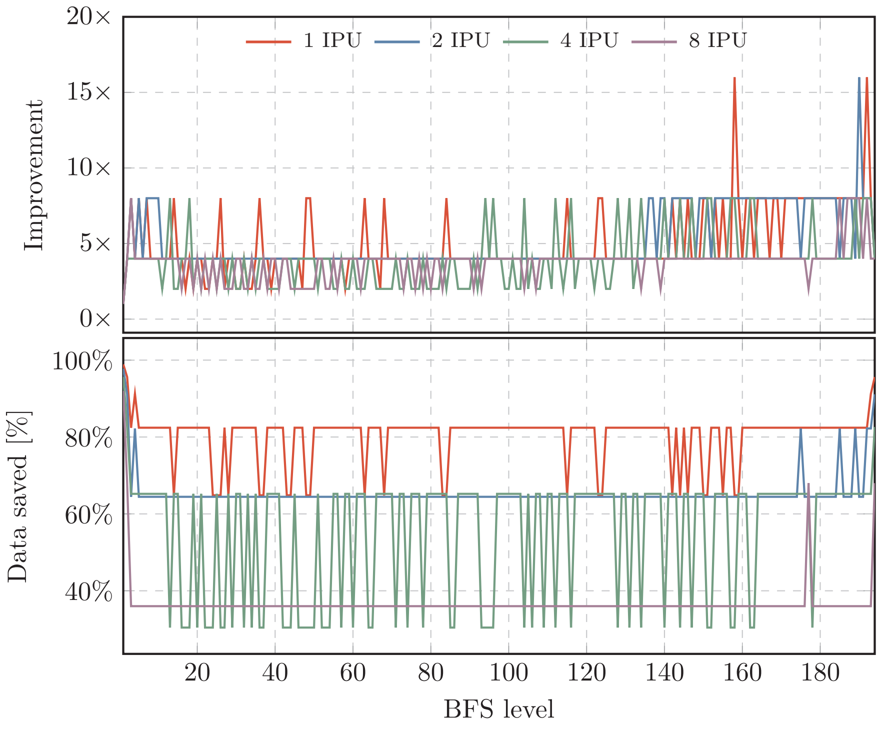

Figure I.8: Difference between a randomly balanced graph and an unbalanced graph, for smaller queue sizes on more IPUs, the imbalance leads to oversizing transmission sizes are leading to less efficient communication. The graphs show the possible amount of data saved compared to dense communication of the expansion phase input queues.The top plot shows the improvement in data size over an unbalanced graph.

when queues are only generated in a few partitions, the data transmission size is significant as the largest queue inflates the communication size. Random permutation of the graph reduces the imbalance and leads to a 3.5× smaller transmission size on average.

Figure I.9 shows an overall performance improvement ranging from −13% for high diameter graphs to 33% for small-world graphs. The performance decrease can be explained as a result of our initial max-reduction. Also, the graph program switching is not completely zero-cost. With a high diameter, repeated execution means that the small overhead adds up. Furthermore, higher IPU counts show a slight performance decrease compared to all double IPU implementations since we are introducing two weak IPU links into our topology with only 2×64 GB/s, while a double IPU pair has three of these links and thus a bisection bandwidth of 192 GB/s.

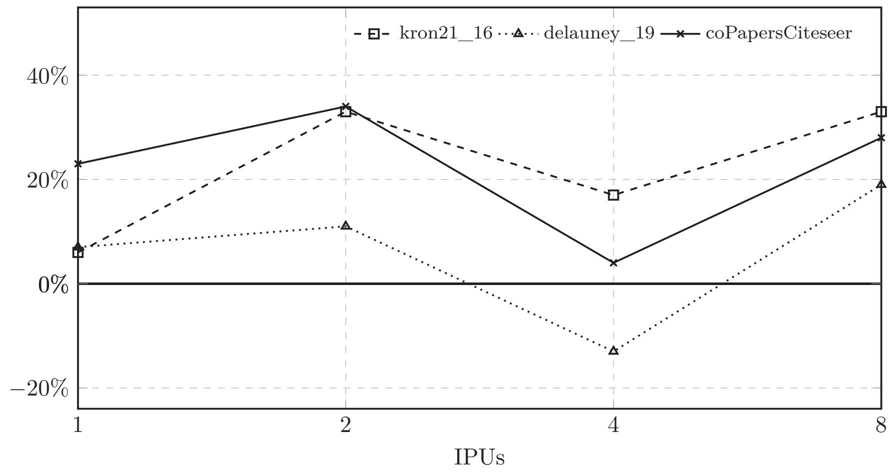

Figure I.9: Performance difference by enabling our (sub)-queue packing optimization. We used a synthetic Kronecker, real-world high-diameter road network, and citation network graph for comparision across different classes of graphs.

I.5 Related Work

BFS is a fundamental graph algorithm and its sequential version is frequently implemented as a subroutine in other graph algorithms. On the other hand, efficient implementation of parallel BFS is challenging since workload and communication patterns shift during the execution of the algorithm. With the wider adoption of parallel computers, efficient implementations have been presented for distributed and shared memory computers, as well as GPUs [BM06; GL05; HN07; KS05; Yoo+05].

After the introduction of the *Graph500* benchmark [Mur+10] in 2010, it became the starting point for a new series of implementations on CPUs [BM11; Che+20; CP14; HOO11; YFG13] and GPUs [Gai+19; LH15; Wan+16; YBO20]. Due to the structure of the test instances in Graph500, direction optimizing searches [BAP11] are highly efficient and thus employed by most modern implementations. However, these are difficult to implement effectively on distributed memory, and they did not lead to performance gains in our IPU implementation.

While AI processors only recently became available, the first papers that evaluate their usefulness for non-AI workloads have appeared recently, both for the Graphcore IPU [Bur+21; LM21; Moh+20] and for the Cerebras Wafer Scale Engine [Roc+20], presenting improvements on the state-of-the-art for a wide range of problems.

I.6 Conclusion

We have implemented the *Graph500* compliant BFS on a multi-IPU systems and tested it on up to 8 IPUs. In a direct comparison, the IPUs are typically faster than GPUs. However, since BFS does not scale linearly, the difference varies widely with the test instance. Due to the direction optimizing BFS, a single GPU can sometimes outperform 8 IPUs or GPUs. On the other hand, direction optimization does not scale well to multiple GPUs, and in the *Graph500* benchmark an equal number of IPUs is around four times faster. We identified three challenges to scaling the computation to larger clusters of IPUs:

I.6.0.1 (Sub)-queue packing

The decreasing effectiveness of this optimization is a more significant concern for strong scaling than for weak scaling, as partition sizes increase with increasing instance size. To counteract the effect of small queue imbalance, an IPU-local queue-merging phase can be introduced.

I.6.0.2 Interconnect

We were able to support weak scaling on the 8 available IPUs, but in larger IPU clusters, such as the IPU-POD64 systems, the bisection bandwidth remains the same and is thus likely to cause communication bottlenecks. It would be remedied if IPUs or other AI processors support more performant topologies in the future.

I.6.0.3 Square partitioning

Our current implementation only supports square partitions of the adjacency matrix. Hence, this leads to a $p \times p$ partitioning pattern, leaving some tiles unused. This imbalance becomes more significant with an increasing number of IPUs, since the partition size becomes more restricted due to the constraint of an even 1D split between the IPUs.

A new GPU system called DGX A100 was recently released, with eight even more powerful A100 GPUs with an even faster connection. Thus, we expect that the GPU results will improve in the near future. Note however that standard GPU systems that rely on PCIe interconnects rather than switched NVLINK are far less competitive. A recent preprint [Gre21] claimed a performance of more than 300 $GTEPS$ for a large Kronecker Graph on 16 V100 GPUs in a DGX-2 system, but at the time of this writing, the code was not publicly available.

While this exceeds the IPU performance reported in this paper, based on power consumption one could argue that 32 IPUs should be the equivalent of one DGX-2. At these sizes the above limitations will play a significant role.

Acknowledgement

The second and third authors were partially supported by the European High-Performance Computing Joint Undertaking under grant agreement No 956213 and the Research Council of Norway under contract 303404. The work has benefited from the Experimental Infrastructure project eX3, which is financially supported by the Research Council of Norway under contract 270053.

References

[Aba+16] Abadi, M. 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.

[BAP11] Beamer, S., Asanovic, K., and Patterson, D. “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).

[BM06] Bader, D. A. and 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)*. IEEE. 2006, pp. 523–530.

[BM11] Buluç, A. and Madduri, K. “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.

[Bul+17] Buluç, A. et al. “Distributed-memory breadth-first search on massive graphs”. In: *arXiv preprint arXiv:1705.04590* (2017).

[Bur+21] Burchard, L. et al. “iPUG: Accelerating Breadth-First Graph Traversals Using Manycore Graphcore IPUs”. In: *International Conference on High Performance Computing*. Springer. 2021, pp. 291–309.

[Che+20] Chenglong, Z. et al. “Efficient Optimization of Graph Computing on High-Throughput Computer”. In: *Journal of Computer Research and Development* vol. 57, no. 6 (2020), p. 1152.

[CP14] Checconi, F. and 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*. IEEE. 2014, pp. 425–434.

[Gai+19] Gaihre, A. et al. “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.

[GL05] Gregor, D. and Lumsdaine, A. “Lifting sequential graph algorithms for distributed-memory parallel computation”. In: *ACM SIGPLAN Notices* vol. 40, no. 10 (2005), pp. 423–437.

[Gre21] Green, O. “ButterFly BFS–An Efficient Communication Pattern for Multi Node Traversals”. In: *arXiv preprint arXiv:2103.13577* (2021).

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

[HOO11] Hong, S., Oguntebi, T., and Olukotun, K. “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.

[HP19] Hennessy, J. L. and Patterson, D. A. “A new golden age for computer architecture”. In: *Communications of the ACM* vol. 62, no. 2 (2019), pp. 48–60.

[Jia+19] Jia, Z. et al. “Dissecting the Graphcore IPU architecture via microbenchmarking”. In: *arXiv preprint arXiv:1912.03413* (2019).

[KG11] Kepner, J. and Gilbert, J. *Graph algorithms in the language of linear algebra*. SIAM, Jan. 2011.

[Kol+19] Kolodziej, S. P. et al. “The SuiteSparse matrix collection website interface”. In: *Journal of Open Source Software* vol. 4, no. 35 (2019), p. 1244.

[KS05] Korf, R. E. and Schultze, P. “Large-scale parallel breadth-first search”. In: *AAAI*. Vol. 5. 2005, pp. 1380–1385.

[Lei+20] Leiserson, C. E. et al. “There’s plenty of room at the Top: What will drive computer performance after Moore’s law?” In: *Science* vol. 368, no. 6495 (2020).

[LH15] Liu, H. and Huang, H. H. “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.

[LK14] Leskovec, J. and Krevl, A. *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data. June 2014.

[LM21] Louw, T. and McIntosh-Smith, S. *Using the Graphcore IPU for traditional HPC applications*. Tech. rep. EasyChair, 2021.

[Moh+20] Mohan, L. R. M. et al. “Studying the potential of Graphcore IPUs for applications in particle physics”. In: *arXiv preprint arXiv:2008.09210* (2020).

[Mur+10] Murphy, R. C. et al. “Introducing the Graph 500”. In: *Cray Users Group (CUG)* vol. 19 (2010), pp. 45–74.

[Pan+17] Pan, Y. et al. “Multi-GPU Graph Analytics”. In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2017, pp. 479–490.

[Pas+19] Paszke, A. et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by Wallach, H. et al. Curran Associates, Inc., 2019, pp. 8024–8035.

[Roc+20] Rocki, K. et al. “Fast stencil-code computation on a wafer-scale processor”. In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2020, pp. 1–14.

[Son+18] Song, L. et al. “GraphR: Accelerating graph processing using ReRAM”. In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2018, pp. 531–543.

[Val90] Valiant, L. G. “A bridging model for parallel computation”. In: *Communications of the ACM* vol. 33, no. 8 (1990), pp. 103–111.

[Wan+16] Wang, Y. et al. “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.

[YBO20] Yang, C., Buluc, A., and Owens, J. D. *GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU*. 2020.

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

[Yoo+05] Yoo, A. et al. “A scalable distributed parallel breadth-first search algorithm on BlueGene/L”. In: *SC’05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. IEEE. 2005, pp. 25–25.

Authors’ addresses

Luk Burchard Simula Research Laboratory, Kristian Augusts gate 23, 0164 Oslo, Norway, luk@simula.no

Xing Cai Simula Research Laboratory, Kristian Augusts gate 23, 0164 Oslo, Norway, xingca@simula.no

Johannes Langguth Simula Research Laboratory, Kristian Augusts gate 23, 0164 Oslo, Norway, langguth@simula.no

Paper II

Enabling Unstructured-Mesh Computation on Massively Tiled AI-Processors: An Example of Accelerating In-Silico Cardiac Simulation

Luk Burchard, Kristian Gregorius Hustad, Johannes Langguth, Xing Cai

Published in *Frontiers in Physics*, Vol. 11, (2023, March), pp. 105. DOI: 10.3389/fphy.2023.979699.

Abstract

A new trend in processor architecture design is the packaging of thousands of small processor cores into a single device, where there is no device-level shared memory but each core has its own local memory. Thus, both the work and data of an application code need to be carefully distributed among the small cores, also termed as *tiles*. In this paper, we investigate how numerical computations that involve unstructured meshes can be efficiently parallelized and executed on a massively tiled architecture. Graphcore IPUs are chosen as the target hardware platform, to which we port an existing monodomain solver that simulates cardiac electrophysiology over realistic 3D irregular heart geometries. There are two computational kernels in this simulator, where a 3D diffusion equation is discretized over an unstructured mesh and numerically approximated by repeatedly executing sparse matrix-vector multiplications (SpMVs), whereas an individual system of ordinary differential equations (ODEs) is

The work of the second author was partially supported by the European High-Performance Computing Joint Undertaking under grant agreement No. 955495 and the Research Council of Norway under contract 329032. The work of the third and fourth authors was partially supported by the European High-Performance Computing Joint Undertaking under grant agreement No. 956213 and the Research Council of Norway under contracts 303404 and 329017. The research presented in this paper has benefited from the Experimental Infrastructure for Exploration of Exascale Computing (eX3), which is financially supported by the Research Council of Norway under contract 270053.

explicitly integrated per mesh cell. We demonstrate how a new style of programming that uses Poplar/C++ can be used to port these commonly encountered computational tasks to Graphcore IPUs. In particular, we describe a per-tile data structure that is adapted to facilitate the inter-tile data exchange needed for parallelizing the SpMVs. We also study the achievable performance of the ODE solver that heavily depends on special mathematical functions, as well as their accuracy aspect on Graphcore IPUs. Moreover, the topics of using multiple IPUs and performance analysis are addressed. In addition to demonstrating an impressive level of performance that can be achieved by IPUs for the monodomain simulation, we also provide a discussion on the generic theme of parallelizing and executing unstructured-mesh multiphysics computations on massively tiled hardware.

Contents

II.1 Introduction

Real-world problems of computational physics often involve irregularly shaped solution domains, which require *unstructured* computational meshes [BP00] to accurately resolve them. The resulting numerical couplings between the entities of an unstructured mesh are irregular, thus in implementations of unstructured-mesh computations, irregular and indirectly-indexed accesses to arrays of numerical values are unavoidable. With respect to performance, there arise several challenges. First, irregular and indirect accesses to array entries are prohibitive for achieving the full speed of a standard memory system [Una+17]. Second, for using a distributed-memory parallel platform, explicit partitioning of an unstructured mesh is required, which is considerably more difficult than partitioning a structured mesh [Bul+16]. Third, there currently exists no universal solution that ensures perfect quality of the partitioned sub-meshes. One specific problem inside the latter challenge is associated with the so-called *halo regions*. That is, the sub-meshes that arise from a non-overlapping decomposition need to be expanded with one or several extra layers of mesh entities, which constitute a halo region of each sub-mesh. These halo regions will be used for facilitating the necessary information exchange between the sub-meshes.

However, most of the partitioning strategies are incapable of producing an even distribution of the halo regions, leading to imbalanced volumes of communication between the sub-meshes, as well as different memory footprints of the sub-meshes.

One particular trend of the latest hardware development requires new attention to the above mentioned challenges. Namely, we have recently seen the arrival of processors with a huge number of small cores sharing no device-level memory. The most prominent examples are the *wafer-scale engine* (WSE) from Cerebras [22b] and the *intelligence processing unit* (IPU) from Graphcore [21]. For example, the second-generation WSE-2 is a chip that consists of 2.6 trillion transistors and houses 850,000 cores. The MK2 GC200 IPU is smaller in scale, but still has 59.4 billion transistors and 1,472 cores. A common design theme of these processors is that there is no device-level memory that is shared among the cores. Instead, each core has its private SRAM and the aggregate on-chip memory bandwidth is staggeringly high, at 20 PB/s for WSE-2 [22b] and 47.5 TB/s for MK2 GC200 IPU [21], respectively. Although these processors are primarily designed for AI workloads, the available aggregate memory bandwidth is appealing to many tasks of scientific computing for which the performance relies on the speed of moving data in and out of the memory system. Storing data directly in SRAM can avoid many “data locality problems" that are typically associated with the multi-leveled caching system found on standard processor architectures. However, we need to remember that a core with its private SRAM, termed as a *tile*, constitutes the basic work unit on WSE-2 or IPU. This leads to higher requirements related to partitioning unstructured-mesh computations, both due to the huge number of available tiles and because all the data needed by each tile must fit into its limited local SRAM. In addition, enabling the necessary communication and coordination between the sub-meshes (i.e., the tiles) requires a new way of programming, unlike using the standard MPI library [Mes21].

Motivated by the extreme computing power that theoretically can be delivered by the massively-tiled AI processors, researchers have recently started applying these AI processors to the “traditional" computational science. For example, Graphcore IPUs have been used for particle physics [Moh+20], computer vision [Ort+20] and graph processing [BCL21; Bur+21]. Regarding mesh-based computations for numerically solving partial differential equations (PDEs), the current research effort is limited to porting stencil methods that are based on uniform meshes and finite difference discretization. This limitation applies to both Cerebras WSE [JAM22; Roc+20] and Graphcore IPU [Håp21; LM21]. To the best of our knowledge, the subject of porting unstructured-mesh computation to massively tiled processors has not yet been addressed.

This paper aims to investigate how to program massively tiled processors for unstructured-mesh computations. The Graphcore IPU, which is a relatively mature AI processor, is adopted as the hardware testbed. Specifically, we study the performance impact of partitioning an unstructured mesh into huge numbers of sub-meshes, as well as how to facilitate communication between the sub-meshes through special IPU programming. Another topic of generic characteristic to study in this paper is the achievable performance and accuracy of evaluating special mathematical functions, such as `exp`, `log` and `sqrt`, on the IPU. As a real-world application of computational physics, we port an existing code that simulates the electrophysiology inside the ventricles, which need to be resolved by high-resolution 3D unstructured meshes. Our work thus makes the following contributions:

- We present a new programming scheme, based on Graphcore's Poplar software stack, for implementing parallel sparse matrix-vector multiply (SpMV) operations that arise from partitioning unstructured meshes.
- We study the impact of mesh partitioning on the size of the halo regions and the associated memory footprints, as well as the performance loss due to the communication overhead. These will be investigated for both single and multi-IPU scenarios.
- We benchmark, in detail, the achievable accuracy and speed of some chosen mathematical functions on IPUs using single precision.
- We demonstrate how a real-world application of computational physics, namely simulating cardiac electrophysiology on 3D unstructured meshes, can utilize the computing power of IPUs.
- We also address the subject of performance analysis of unstructured-mesh computational on IPUs.

It is remarked that the chosen existing code numerically solves a widely used mathematical model in computational electrophysiology. The choice is also motivated by the fact that the performance of the code on GPUs has been carefully optimized and studied in [Hus19], thus providing a comparison baseline for this paper where one of the research subjects is the achievable performance of unstructured-mesh computations on IPUs. No new numerical algorithm will be introduced in this paper, specifically, cell-centered finite volume discretization in space and explicit integration in time (used by the existing code and more details can be found in Section II.3) will continue to be used. The readers will recognize two familiar computational kernels, namely, parallel SpMV operations arising from unstructured meshes and explicit integration of systems of nonlinear ordinary differential equations (ODEs). The findings of this paper, regarding both the programming details needed to enable inter-tile communication on IPUs and the achievable performance, will thus be applicable beyond the domain of computational electrophysiology.

The remainder of this paper is organized as follows. Section II.2 gives a brief introduction to the mathematical model considered, whereas Section II.3 explains the numerical strategy and its parallelization of the existing simulator. Section II.4 focuses on the new programming details required for using the IPU. Numerical accuracy is considered in Sections II.5 and II.6. The former measures general floating-point accuracy of the IPU while the latter tests the fidelity of the simulation using an established benchmark. Section II.7 presents the measured performance of our IPU implementation, together with a comparison with the GPU counterpart. The impact of imbalanced halo region distribution is also investigated. Section II.8 concludes the paper.

II.2 Monodomain Model of Cardiac Electrophysiology

The increase in computing power over the past decades has facilitated the use of computer simulations to better understand cardiac pathologies. One of the most important cardiac processes is the propagation of the electrical signals, which trigger the contraction of the heart muscle. This process of electrophysiology, in its simplest form, can be mathematically described as a reaction-diffusion system using the following monodomain model (see e.g. [Cla+11]):

$$\frac{\partial v}{\partial t} + \chi I_{\text{ion}}(t, v, \vec{s}) = \nabla \cdot (M \nabla v). \tag{II.1}$$

In the above model, $v(x, y, z, t)$ denotes the transmembrane potential and is mathematically considered as a time-dependent 3D field, $M(x, y, z)$ is a conductivity tensor field describing the space-varying cardiac muscle structure, χ is the ratio of the cell membrane area to the cell volume, and $I_{\text{ion}}(t, v, \vec{s})$ denotes the total ionic current across the cell membrane. Here, $\vec{s}(t)$ denotes a vector of state variables (apart from v) that also contributes to the evolvement of I_{ion} at each spatial point, where the evolution of $\vec{s}$ is governed by a system of ODEs (see below).

Despite its simplicity, the monodomain model (II.1) is frequently used by researchers to capture the main features of the signal propagation, especially over the entire heart. Numerically, this reaction-diffusion system is often solved using an operator splitting method to decouple the reaction term, $\chi I_{\text{ion}}(t, v, \vec{s})$, from the diffusion term, $\nabla \cdot (M \nabla v)$. The reaction part of the monodomain model is then formulated as a system of non-linear ODEs:

$$\frac{\mathrm{d}v}{\mathrm{d}t} = -\chi I_{\text{ion}}(t, v, \vec{s}), \tag{II.2}$$

$$\frac{\mathrm{d}\vec{s}}{\mathrm{d}t} = \vec{f}(t, v, \vec{s}). \tag{II.3}$$

This ODE system needs to be *individually* solved on each mesh entity of a computational grid. The diffusion part of the monodomain model is thus a linear 3D PDE as follows:

$$\frac{\partial v}{\partial t} = \nabla \cdot (M \nabla v). \tag{II.4}$$

This PDE needs to be solved involving all the discrete v values in a computational grid.

In this paper, the ten Tusscher–Panfilov model (TP06, see [TP06]) is used for modeling I_{ion}, where the entire ODE system (II.2)-(II.3) involves v and 18 state variables. These state variables include various ionic currents and so-called gating variables. Special math functions such as `exp` are heavily used in the ODEs. The source code of a straightforward implementation of the specific ODE system can be found in [ten21]. Furthermore, the TP06 model prescribes different parameters for three types of cardiac cells located in the *epicardium* (outer layer), the *myocardium* (middle layer), and the *endocardium* (inner layer) of the muscle wall in the cardiac ventricles.

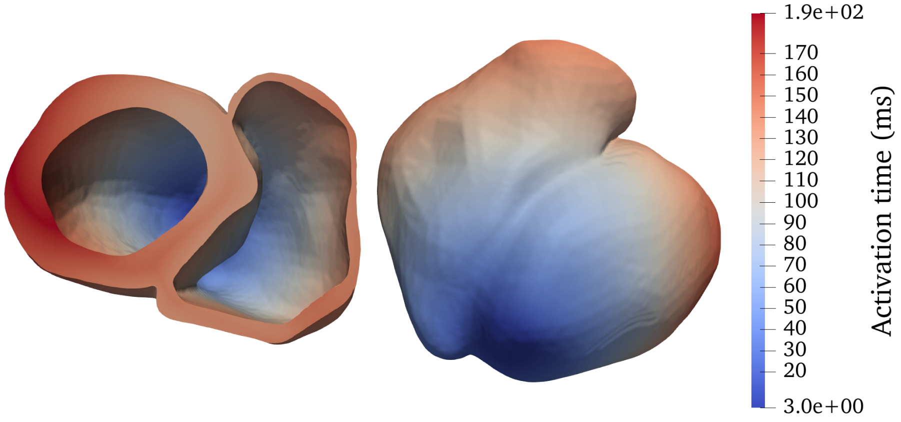

Figure II.1: Activation map for a realistic bi-ventricular mesh, named `heart04`, with a stimulus applied initially to the apex of the left ventricle (darkest blue). The panels show two different viewing angles of the same mesh.

II.3 Numerical Strategy and Distributed-Memory Parallelization

II.3.1 Unstructured Tetrahedral Meshes

To sufficiently resolve the irregular shapes inside a heart, we adopt unstructured tetrahedral meshes. The adopted bi-ventricular meshes are based on the dataset published by Martinez–Navarro et al. (see [Mar+19]), and the LDRB algorithm (see [Bay+12]) for determining muscle fibre directions. We have used transversely isotropic conductivity tensors such that the conductivity only depends on the longitudinal fibre direction, i.e., parallel to the muscle fibres. Two realistic bi-ventricular meshes have been used for the numerical experiments to be presented later in this paper. The `heart04` mesh has in total 3,031,704 tetrahedrons and a characteristic length of 0.4 mm, whereas the finer `heart05` mesh has in total 7,205,076 tetrahedrons and a characteristic length of 0.3 mm. Figure II.1 shows a simulated activation map of `heart04` where a stimulus is applied initially to the apex of the left ventricle.

II.3.2 Discretization of the ODE Systems

The nonlinear ODE system (II.2)-(II.3) needs to be individually solved per tetrahedron. We follow previous studies [Mir+12; MZS12] in using an augmented forward–Euler scheme, where one of the state variables (the so-called m gate) is integrated using the Rush–Larsen method [RL78] for improved numerical stability. That is, the other 17 state variables and v are explicitly integrated by the standard forward–Euler scheme. In order to obtain sufficient accuracy, we use a time step $\Delta t_{\mathrm{ODE}} = 20\,\mu\mathrm{s}$ for the ODE system (II.2)-(II.3).

II.3.3 Discretization of the Diffusion Equation

The numerical solver for the diffusion equation (II.4) is based on *cell-centered* finite volume discretization in space and explicit integration in time [Lan+15a; Lan+15b]. (There exist other numerical strategies that are based on finite element discretization in space and/or implicit integration in time. We will provide a discussion in Section II.8.) The resulting computational stencil from the cell-centered finite volume discretization covers the four direct neighboring tetrahedrons plus the twelve second-tier neighbors, such that the computational formula per tetrahedron depends on its 16 neighbors in addition to itself. This can be expressed straightforwardly as an SpMV operation:

$$\vec{v}_{n+1} = Z\vec{v}_n, \tag{II.5}$$

where the vectors $\vec{v}_{n+1}$ and $\vec{v}_n$ contain the numerical solutions at $t = (n+1)\Delta t_{\mathrm{PDE}}$ and $t = n\Delta t_{\mathrm{PDE}}$, respectively. Each value of vectors $\vec{v}_{n+1}$ or $\vec{v}_n$ is the numerical approximation of the transmembrane potential v at the center of a tetrahedron (which will also be called a cell in the remaining text). The matrix Z has a dense diagonal, and each row, corresponding to a specific tetrahedron, has up to 16 off-diagonal non-zero entries. The column positions of these non-zero entries are irregular due to the unstructured tetrahedral mesh.

Suppose the number of tetrahedrons of the 3D unstructured mesh is denoted by N. The dense diagonal of the matrix Z is stored as a separate 1D array, $\mathbf{D}$, of length N, whereas the off-diagonal entries are stored in the ELLPACK format (see [GKY79]) with 16 values per row. This results in two 2D arrays each with $N \times 16$ entries: $\mathbf{A}$ contains the non-zero floating-point values in the off-diagonal part of the matrix Z, and $\mathbf{I}$ contains the corresponding column indices.

As is common for explicit methods, the entries of Z impose a limit on how large the PDE time step Δt_{PDE} can be chosen without giving rise to numerical instability. When the criterion on Δt_{PDE} is much stricter than that on the ODE time step Δt_{ODE}, we use multi-stepping, meaning that p PDE steps with $\Delta t_{\mathrm{PDE}} = \frac{\Delta t_{\mathrm{ODE}}}{p}$ are executed for each ODE step. Otherwise, if the accuracy-determined value of Δt_{ODE} is roughly the same as the stability-determined value of Δt_{PDE}, the minimum of the two is used for solving both parts of the monodomain model.

II.3.4 Distributed-Memory Parallelization

On distributed-memory architectures, we need to partition the unstructured tetrahedral mesh in such a manner that the computation is evenly distributed among the hardware units, e.g., the GPUs on a cluster or the tiles within an IPU. Furthermore, the partitioning should ideally lead to a minimal communication volume to allow for scaling of the parallelized simulator. Specifically, we first construct an undirected graph based on the cell connectivity of the tetrahedral mesh and then use a graph partitioner (e.g., from the METIS library [KK98]) to create a partitioning that attempts to minimize the total communication volume within the constraints of a given maximal work imbalance ratio.

With respect to parallelizing the SpMV operation (II.5) that constitutes the computation of each PDE step, the non-overlapping sub-meshes that are produced by the graph partitioner dictate how the rows of the sparse matrix Z are distributed among the hard units (such as the tiles within an IPU). Also, the input vector $\vec{v}_n$ that is to be multiplied with Z needs to be partitioned accordingly. On each hardware unit, besides its non-overlapping partition of $\vec{v}_n$, additional *halo* values of $\vec{v}_n$ need to be included. These halo values represent the needed contributions from the neighboring partitions, see Section II.1. Before a distributed-memory parallel SpMV is executed, the halo values must be communicated from the neighbors. In return, some values within each non-overlapping partition of $\vec{v}_n$ are needed as halo values on the neighbors, so these values must be communicated to the neighbors.

II.4 Porting to Graphcore IPU

We have chosen as the starting point an existing code, which is described in [Hus19; Lan+15a; Lan+15b], for simulating the monodomain model. The numerical strategy of the existing monodomain simulator is as described in Section II.3. The same distributed-memory parallelization will also be used, with the exception of how halo-data exchanges are enabled.

II.4.1 Halo-Data Exchanges

Before describing the communication details, we need to introduce some definitions first. The cells of each non-overlapping partition are of two types: the *interior* and *separator* cells, where the interior cells are not needed by any other partition, whereas each separator cell is needed by at least one neighboring partition. Therefore, the interior cells are not included in any communication. Values of the separator cells are computed on the owner partition, but need to be communicated outwards to the requiring partition(s). On the receiving partition(s) the corresponding cells are called *halo* cells.

On each sub-mesh, the interior and separator cells are identified on the basis of the non-overlapping partitioning result produced by the graph partitioner (see Section II.3.4). The needed addition of halo cells uses the same partitioning result. However, the cells and their dependencies arise from a 3D geometry that is represented in 1D memory. Therefore, the separator cells on each partition *may not* be rearranged in such a way that the “outgoing" data to the different neighbors appear as non-overlapping memory ranges.

Suppose destination partitions A, B, C all require some values from the same source partition, where A requires cells $\{1, 2\}$, B requires $\{2, 3\}$, and C requires $\{1, 3\}$. Since the “outgoing" data from the source partition to the destinations A, B, C have overlapping values, it is impossible to arrange the separator cell values in the memory of the source partition as three non-overlapping ranges.

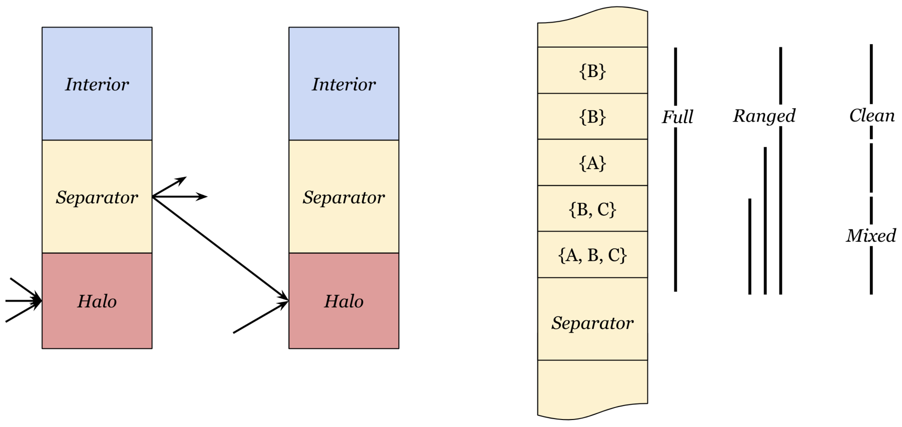

Figure II.2: Definition of *interior*, *separator*, and *halo* cells. The left side depicts the three cell regions, where the *interior* region is not communicated, values in the *separator* region need to be sent to other partitions, and the *halo* region receives *separator* values from neighbor partitions. The right side depicts three *separator* range methods, (full) transmits the entire *separator* cell range to every neighbor, (ranged) uses the smallest enclosing range for each receiving partition, and (mixed-clean) splits the *separator* cell range into a mixed part and a clean part.

Consequently, if each outgoing data sequence must be marked by a contiguous range, using one start position and one end position in memory, such as required to enable inter-tile data exchanges on the IPU (see Section II.B.4), the destination partitions may have to receive some “unwanted" data together with the needed data. This issue is particularly important for using the IPU, because Poplar programming (see Section II.B.4) does not provide explicit communication, such as sending and receiving messages in the MPI standard. Instead, the need for inter-tile data exchange is automatically identified and arranged by the *popc* compiler during compilation, based on shared data ranges between the tiles.

We thus reorder the separator cells in each partition such that the data range for each destination partition contains minimal unused values. An alternative approach would be to generate many tiny ranges for each cell in each location. However, this fine-grained mapping would generate many communication programs, which will require a significant amount of program memory. Therefore, we consider three separator division schemes that are responsible for reordering the separator cells and generating outgoing ranges over segments of memory, see Figure II.2 and also an algorithmic description in Appendix II.A.

II.4.1.1 Full Separator Communication

The simplest form of *separator* reordering and outgoing range determination is to declare the full *separator* region as the outbound region for all neighbors. This has the advantage that the compiler may optimize the internal exchanges to use broadcast operations, having fewer data transmissions on the exchange fabric. The downside is that the whole *separator* contains values that are not needed for all neighbors. Since memory is a scarce resource on the IPU, a concern is about the unused values included in these exchanges, as they will also increase the size of the *halo* region on the receiving partitions.

II.4.1.2 Ranged Separator Communication

The ranged *separator* communication scheme creates one outgoing range from a source tile to each of its destination tiles. Instead of using the whole *separator*, we can use the smallest range for each destination tile as the outgoing memory region range, which contains all values that must be transferred. As the values can not be sorted such that ranges do not overlap, we still transfer unused values (but fewer) similar to the full separator communication scheme.

II.4.1.3 Mixed-Clean Separator Communication

To reduce the number of unused values sent, we can create two ranges for each destination. One range includes values only destined to the specific destination and another range contains values destined to multiple neighbors. We reorder the values destined to multiple partitions into a “mixed" range, which gets transferred to all neighbor partitions. The values only destined to a single destination partition are in a “clean" range, not containing unused values. Therefore, up to two exchange packets are sent to each neighbor partition.

The advantage of this scheme is that the mixed range can still be broadcast as they are the same for all neighbor partitions. Furthermore, clean ranges reduce the number of values transmitted and stored in the *halo* regions, thus reducing memory usage.

II.4.2 Specific IPU Programming

Porting the existing monodomain simulator to the IPU requires porting two components, a PDE step and an ODE step, each programmed as a Poplar codelet (see Section II.B.4). The ODE step is independent between the cells; therefore, it can be easily distributed and executed in parallel on all tiles without communication. Each tile goes through all its interior and separator cells and integrate the ODE system per cell. The required Poplar programming is mostly about deriving a Poplar `Vertex` class that wraps C functions in the existing monodomain simulator.

The PDE step does an SpMV (II.5) involving the voltage values on all the cells. This requires beforehand the communication and propagation of halo values between each pair of neighboring tiles. In Poplar, inter-tile communication is described implicitly, see Section II.B.4. When partitioning the unstructured tetrahedral mesh, we also determine the ranges that need to be communicated from one tile to another. Section II.4.1 has described how these ranges are obtained. The globally addressed vector of voltages, V, is required and partitioned into different partitions to be multiplied with the sparse matrix Z, partitioned and stored in the ELLPACK format per tile. Specifically, vector V is decomposed into multiple partitions $V_{|i,j} = V_p$ with the global index ranging from i to j for the tile with index p.

Here, we recall that each value in vector V is owned by one tile, and the value may be a halo value in one or several tiles. Poplar requires defining the locality of data on subdomains of tensors. We can do this mapping with, e.g., `graph.setTileMapping(V.slice(i,j), 123)`, which maps the range i to j of V to the tile with the identifier of 123. To append halo data into an existing tensor that is already on the tile we can create a virtual tensor, composed of different tensor regions. This virtual tensor will be created through copies and exchanges once the BSP-superstep is launched using this tensor (see Section II.B.4). Such a virtual tensor can be created by `localV = concat(localV, V.slice(m,n))` $\forall(m,n) \in$ `separatorReceive`, where `localV` is a tensor representing V_p. This simple statement implies the needed tile-to-tile communication, as the ranges (m,n) are owned and computed by other tiles. The compiler automatically generates the exchanges and synchronization steps needed when the PDE *compute-vertex* using `localV` is running. On the other hand, `localV` is not necessarily materialized when the PDE *compute-vertex* is not running. The memory space can be overwritten by other operations.

Attention must be paid when working with big exchanges of data, as data is duplicated when transferred. Therefore, bigger exchanges can transfer at most half of the per-tile memory in an optimal scenario. An alternative approach is to communicate in smaller chunks to mitigate this problem.

II.4.3 Performance Modeling

On CPUs and GPUs, all hardware optimizations need to be accounted for to determine the expected runtime. Hence, if not done rigorously, the actual performance can differ dramatically from a pre-determined performance model. However, the IPU does not have special hardware optimizations such as caches or instruction pipelining (see Section II.B.3) and stalling.

Due to the simplicity of the IPU architecture, we can count instructions in any algorithm in the computation phase, to create a performance model. In the following, our focus is on the computations involved in the PDE and ODE parts of the monodomain simulator. Using Graphcore's `popc` compiler with the `-O3` flag we generated the assembly output for the GC200 platform.

II.4.3.1 Performance Model for the PDE Part

On conventional platforms, due to the unstructured computational mesh used, the SpMV operation during each PDE step has to access values that are randomly stored in the main memory. On such systems, due to the random access pattern, the values cannot be fully cached, leading to long runtimes for SpMV.

On IPU, however, once the sparse matrix and the associated vectors (with halo parts) are partitioned among the tiles, data is only accessed from the local SRAM of each tile. The Poplar code that implements the compute phase of SpMV is given in Listing II.1.

Listing II.1: ELLPACK formatted SpMV used for the PDE computation.

```
const int RNZ = 16; // Rank Non-Zero
class TestPDEPartMW : public Vertex
{
  public:
    // Constants
    Input<Vector<float>> A;
    Input<Vector<int>> I;
    Input<Vector<float>> D;

    // Vertex In/Out
    Input<Vector<float>> V;
    Output<Vector<float>> newV;

    bool compute(unsigned workerId)
    {
        for (int i = 0; i < newV.size(); i++) {
            const int j = RNZ * i;
            float a = 0;
            a += A[j + 0] * V[I[j + 0]];
            a += A[j + 1] * V[I[j + 1]];
            // ...
            a += A[j + 15] * V[I[j + 15]];
            newV[i] = D[i] * V[i] + a;
        }
        return true;
    }
};
```

Counting the instructions of the inner SpMV loop we estimated for each tetrahedron 107 execution context local cycles, and measured a real performance of 106 execution context local cycles. We noticed that of the 107 cycles, 18 cycles are used to store and load spilled registers. Furthermore, the generated code makes use of instruction bundles but does not yet include more specialized instructions available to the GC200, such as headless loops. The one-time start and teardown overhead is 153 instructions.

II.4.3.2 Performance Model for the ODE Part

As mentioned in Section II.2, the ten Tusscher–Panfilov model (TP06, see [TP06]) is used for modeling the ionic currents (II.2)-(II.3). An augmented forward–Euler scheme is used to explicitly integrate the ODE system (see Section II.3.2). Through counting instructions of the TP06 model, we arrive at 1077 instruction bundles. Adding latencies for the involved math functions (see Section II.B.2), we get an estimate of 1401 cycles per tetrahedron (per time step). Adding the loop overhead gives us a total of 1418 cycles. Our measurements can confirm this model with the 1415 cycles actually used. We noticed that 111 instructions, or about 7.9%, were used to handle register spilling. Furthermore, we noticed that the compiler generated in some places 2-element vectorized operations, which is the maximum SIMD operation width for $32bit$ single-precision float values.

II.5 Math Accuracy

This section has the purpose of rigorously verifying the accuracy of some representative mathematical functions when executed on the IPU using single precision. We consider it a very important step before adopting this new processor architecture for numerically solving the monodomain model.

The IPU has access to the IEEE754 [IEE08] standard for single-precision floating-point numbers. IEEE754 specifies these as *binary32* with one sign bit indicating a positive or negative number, an exponent of 8 bits, and 23 mantissa bits. The general form of a normalized floating number can be given as

$$(m_0.m_1 \ldots m_p)_2 \times \beta^e$$

with one non-zero digit in front of the comma. For the IPU, the base is $\beta = 2$ and according to IEEE754 $m_0 = 1$ is specified.

II.5.1 Subnormal Numbers

The IEEE754 [IEE08] standard formulates an extension of floating-point numbers to subnormal numbers. Subnormal numbers are used to represent numbers smaller than the minimum floating-point number at the loss of accuracy. Subnormals have are represented with $e_{min} = 1 - e_{max}$ and have $m_0 = 0$. The numbers they can represent are numbers between zeros and the minimum normalized single float number. The subnormals are linearly spaced, while normalized floating-point numbers have logarithmic spacing.

The IPU does not support hardware accelerated subnormals. However, CPUs and GPUs are also not primarily optimized to deal with subnormals. As shown by Fasi et al. [Fas+21], the V100 and A100 have hardware-accelerated support for subnormals. However, noted in both Intel and Nvidia developer documentation, subnormals can be an order of magnitude slower than normalized floats.

II.5.2 Units in the Last Place

To compare floating-point numbers in one format, such as the IEEE754 single-precision format, we choose the *units in the last place* (ULP) as a metric rather than the relative error. One ULP is equivalent to

$$ulp(\beta, e) = (0_1.0_2 \dots 0_{p-1}1_p)_2 \times \beta^e$$

and defined under a constant basis and exponent. One ULP has the scalar distance between representable numbers, with bases β and exponent e.

When considering the error, we generally are speaking about the difference from a calculated result $\hat{y}$ to the mathematically precise and accurate result y:

$$\text{error}_{\text{absolute}} = |y - \hat{y}|$$

Even with perfect accuracy, we can not precisely represent y in any machine format, as the real value can always be between two representable numbers. Considering that we are rounding to the nearest number, even with a perfect calculation, the error can still be up to $0.5 \times ulp(\beta, e)$, because we have $\hat{y} \leq y \leq \hat{y} + 0.5ulp(\beta, e) \leq \hat{y} + ulp(\beta, e)$.

II.5.3 Experiment Setup

We are interested in determining the accuracy of the most important mathematical functions used in our simulator. The functions with one argument we want to analyze are $\{exp, expm1, log, sqrt\}$. The function with two operands of interest is $\{\div\}$.

All functions operands x_n are represented in the IEEE754 single-precision floating-point format. The results $\hat{y}$ are like the input operands in single precision. When considering the mathematically accurate result y for a comparison with $\hat{y}$, we are using double-precision floating-point numbers. However, for $|y - \hat{y}|$, we are interested in the error represented in ULPs, which gives us a fair estimate of how far we are off in our given representation. Therefore, to get the difference of $y - \hat{y}$ in ULPs, we need to convert y into the same floating-point format as $\hat{y}$ with the same base and exponent. This conversion gives us the most accurate result in our chosen representation. We assume the double floating-point result is accurate and precise only with a minimal error, which occurs beyond the representation of single-precision floating-point numbers. Therefore, when converting from double to single precision, we do not introduce an error bigger than $0.5 \times ulp(\beta, e)$. The final error in ULPs represents how many places in our single-precision representation we are off. For example, we are able to only represent one place behind the comma with a basis and exponent β^e, our accurate mathematical result would be $y = 1.3315 \times \beta^e$. If we calculated $\hat{y} = 1.5 \times \beta^e$, while the mathematical result in our representation would be $y = 1.3 \times \beta^e$, the observed result $\hat{y}$ is wrong by by two ULPs.

For functions with one operand, we can iterate over all single-precision floating-point values as the possible inputs, thus bounded by just 2^{32} different representable states. However, we are unable to iterate through functions with two operands as the number of possible inputs is not iterable with 2^{64} possible states. Hence, we randomly sample 10 billion different input operand pairs.

II.5.4 Experiment Results

All single-operand math functions $\{exp, expm1, log, sqrt\}$ and the two-operand function $\{\div\}$ have been implemented as hardware instructions and specifiable in the assembly. We noticed that subnormal values are rounded off to zero in input, also, if a not normalized output could be expected. Furthermore, operands that are normalized and produce a subnormal output are rounded off to zero. Both cases are covered in the hardware implementation and produce faster results after a single cycle.

All functions showed no more than a $1 \times ulp(\beta, e)$ error. Errors of one ULP were found randomly distributed throughout the results. Thus, we can say that the single-precision floating-point implementation of the IPU can be considered accurate.

II.6 Niederer Benchmark

II.6.1 Running the Benchmark

Now the task is to verify the correctness of our IPU implementation. Ideally we should choose a well-known problem instance and compare our computational results with real-world measurements. However, cardiac simulations are very complex, with multiple parameters and non-trivial geometries. Therefore, the scientific community has agreed to compare multiple codes against each other to create a golden solution that is between all viable simulations. Niederer et al. [Nie+11] formulated a *N-version* benchmark to compare cardiac electrophysiology simulators independent of the numerical scheme used.

More specifically, the Niederer benchmark uses a box-shaped geometry to represent a $20 \times 7 \times 3$ mm slab of cardiac tissue with fibers aligned along the long (20 mm) axis. A stimulus is applied within a small $1.5 \times 1.5 \times 1.5$ mm cube at a corner of the geometry for a duration of 2 ms and with a stimulation strength of 50000 $^{\mu A}/_{cm^3}$.

II.6.2 Benchmark Results

As described above, the Niederer Benchmark compares multiple simulation results against each other. As the baseline, we used the existing monodomain code (its CPU version denoted as "OMP" in Figure II.3), executed on an Intel Xeon CPU with *IEEE754-float64* double precision. We compared the IPU implementation of the same cell model to the CPU implementation with more accurate math.

The benchmark is run for three meshes of 0.5, 0.2, and 0.1 mm in resolution. The PDE and ODE time steps are fixed to 0.05 ms. In Figure II.3 we observe that the benchmark results computed by the IPU do not diverge significantly from the CPU baseline. When using the same mesh resolution, e.g., 0.1 mm, the two implementations produce results that are difficult to distinguish with naked eyes.

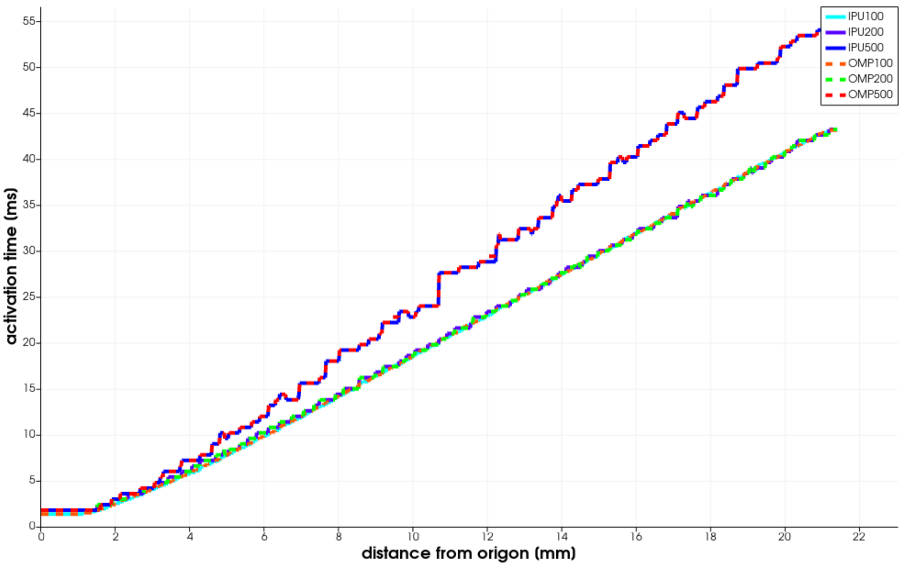

Figure II.3: The Niederer Benchmark results reported as the activation time versus the distance from the stimulus origin.

In Section II.7.4, we will present another comparison of the simulation results between the ported IPU code and an existing GPU implementation. This comparison addresses a realistic heart geometry and unstructured computational meshes.

II.7 Experiments

II.7.1 Separator Experiments

To determine the effectiveness of the three schemes for reordering and dividing the separator cells on each tile, as discussed in Section II.4.1, we use the `heart04` mesh (see Section II.3.1) as an example. For a given number of IPUs, the unstructured mesh is decomposed by the k-way partitioner of METIS [KK98] into N sub-meshes, with N equaling the total number of tiles available. For all the experiments, we have always used an imbalance ratio of 3% while minimizing the adjacency edge-cut, meaning that the number of cells on the largest partition can not be more than 1.03 times greater than the ideal partition size. This offered us a good tradeoff between runtime and partitioning quality. With more

Table II.1: Partitioning the `heart04` mesh using 1 to 16 IPUs under three different exchange strategies. The inbound cells correspond to the *halo* cells. The total cells are of the biggest partition. Unused cells refer to the cells in the halo region not used by the receiving partition.

	Inbound Cells Max			Total Cells Max			Unused Cells Median		
IPU Count	Full	Mixed	Ranged	Full	Mixed	Ranged	Full	Mixed	Ranged
1	3.4×	1×	2.1×	2.6×	1×	1.7×	5.3×	1×	2.7×
2	2.8×	1×	1.8×	2.3×	1×	1.6×	3.9×	1×	2.0×
4	2.2×	1×	1.5×	2.0×	1×	1.4×	3.0×	1×	1.6×
8	1.9×	1×	1.3×	1.7×	1×	1.3×	2.4×	1×	1.3×
16	1.5×	1×	1.2×	1.5×	1×	1.2×	1.8×	1×	1.3×

workload imbalance, the downsides are twofold: (1) As the IPU uses a BSP (*bulk synchronous parallel*) programming model, all tiles need to wait until the last tile is finished, leading to poor use of the hardware resources. (2) The maximum problem size which fits on a set of IPUs is reduced, because once a single partition becomes too large, the *popc* compiler fails. Setting a lower work imbalance ratio would reduce the computation time in the PDE and ODE steps, but it might increase the communication volume and thereby the time spent on the halo-data exchange phase that occurs before each PDE step.

First, we are interested in the maximum and median memory used on the tiles, because the maximum memory requirement per tile indirectly determines how big meshes can fit on a single or multiple IPUs. The second metric of interest is the maximum and median inbound communication volume per tile and the number of unused cells included in the halo regions. To evaluate the effect on a real-world application, we benchmarked the PDE exchange phase and count the number of cycles used for the exchanges (see details in Table II.2 later in Section II.7.3).

A high-level comparison is shown in Table II.1. We can observe that the mixed-clean strategy outperforms the full and ranged strategies. The latter two are on average 2× worse. As expected, the full separator strategy performs consistently worse than the ranged separator strategy. However, the advantage of the mixed-clean strategy may not hold for very small partitions, e.g., when the number of IPUs used is very large. This is because as the partitions get smaller, the overlapping ranges in the mixed-clean separator strategy increase, approaching the same level of the full separator strategy. The ranged separator strategy, therefore, may have a small advantage.

When running the real-world benchmarks, such as those to be presented in Section II.7.2, we observed a correlation between the number of inbound cells and the exchange time usage. The mixed-clean strategy was thus used when running the IPU-ported monodomain simulator.

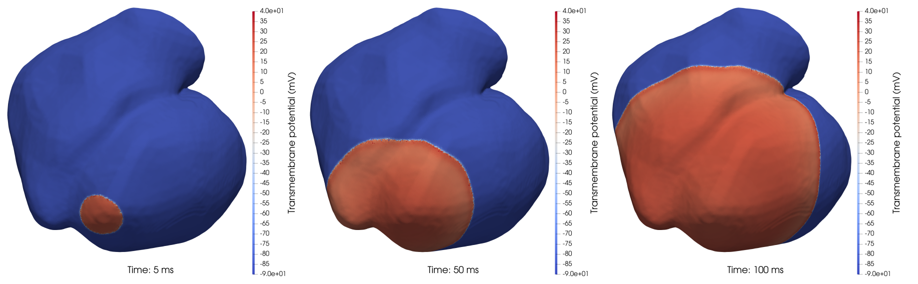

Figure II.4: Three snapshots of the 3D transmembrane potential field from a realistic monodomain simulation using the `heart04` mesh.

II.7.2 Strong-Scaling Experiments

We continue with some strong-scaling experiments, i.e., the computational mesh is fixed while the number of IPUs increases. For all the IPU numbers tested, we used the `heart04` mesh (with 3,031,704 cells), as it is the biggest fitting instance that runs on a single IPU. Figure II.4 shows three snapshots from a monodomain simulation using the `heart04` mesh. It simulates the propagation of the electrical signal in the heart for 500 ms, with $\Delta t_{\text{ODE}} = 20\,\mu\text{s}$ and $\Delta t_{\text{PDE}} = 5\,\mu\text{s}$, i.e., one ODE step per 4 PDE steps. Besides `heart04`, we also used `heart05` (with 7,205,076 cells) for which two IPUs were the smallest configuration. This mesh has a finer resolution than `heart04`, thus a smaller value of $\Delta t_{\text{PDE}} = 4\,\mu\text{s}$ was used, i.e., one ODE step per five PDE steps. In addition, we compared the IPU performance with the GPU performance of the existing code [Hus19] using one to eight GPUs in an NVIDIA DGX A100 system. A high-level summary of the comparison is shown in Figure II.5, where Section II.7.4 contains a deep-dive analysis.

The k-way partitioner from the METIS software package was used to divide the unstructured meshes, with k equal the total number of tiles used for each IPU experiment. For the GPU experiments, the METIS partitioner only needs to divide the unstructured meshes into sub-meshes equaling the number of GPUs used. (The intra-GPU parallelism utilized the device-level memory accessible for all the CUDA threads.) All the experiments used a METIS load imbalance constraint of 3%.

Figure II.5 shows the results of the strong-scaling experiment. We can observe that both IPU and GPU implementations scale almost linearly. The A100 has almost twice the performance and is always matched by twice the number of IPUs. However, when approaching eight GPUs the scaling efficiency drops. This is not the case for the IPUs, which are able to scale up to 16 IPUs. We were not able to run this experiment on more than 16 IPUs, as the *popc* compilation ran out of memory when trying to compile for 32 IPUs.

When increasing the number of IPUs from 1 to 16 for the `heart04` mesh, we observe that the time decreases almost linearly with added hardware. From

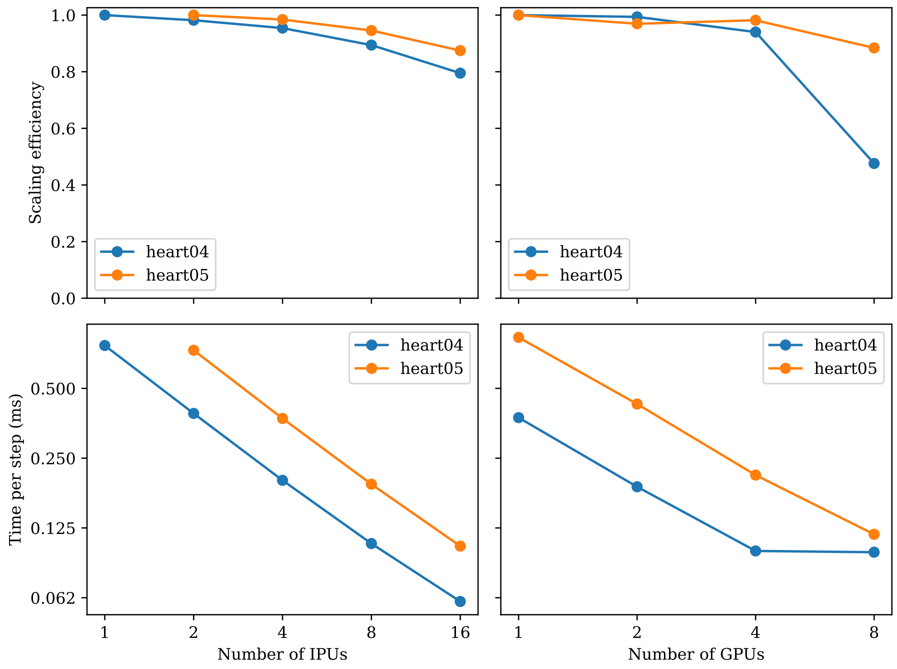

Figure II.5: Strong-scaling experiments using the meshes of `heart04` and `heart05`. (The `heart05` mesh is too large for a single IPU.)

the perspective of computation alone, this scaling trend is expected, because each tile is responsible for less work. However, due to the increasing number of tiles used, the amount of halo cells increases. When using a single IPU for the `heart04` mesh, we see a median of 2058 interior+separator cells per tile and a median of 2226 halo cells per tile. That is, the halo cells occupy about 51.9% of the total cells per tile. Let us also recall that the values of these halo cells need to be transferred from other tiles before every diffusion step. With more IPUs, the transfer volume increases to about $63.33\%, 72.60\%, 80.20\%, 86.13\%$ of all the cells, for $2, 4, 8, 16$ IPUs respectively. The memory footprints of the halo cells are thus steadily increasing.

II.7.3 Phase Breakdown

We used the Graphcore PopVision tools to visualize the inner workings of our IPU-enabled simulator. In order to generate a profile containing runtime and compile information, an environment variable has to be passed during compilation, which will make the `Poplar` libraries generate profiling information. The profiling information contains static runtime-independent data, such as the memory usage on each tile. Runtime-dependent information is also collected. However, runtime profiling incurs a small performance and memory usage penalty. We are thus only

Table II.2: Breakdown of the different algorithmic phases (in number of clock cycles) for the strong-scaling experiments using the `heart04` mesh. The PDE per ODE factor p is always 4, i.e., four PDE steps per ODE step.

IPUs	PDE Ex	PDE Comp	ODE Comp	PDE Total	ODE Total	PDE/ODE	$\frac{\text{PDE}}{p}$/ODE
1	11,365	250,916	3,018,062	1,049,124	3,018,062	34.76%	8.69%
2	14,642	125,600	1,512,146	560,968	1,512,146	37.10%	9.27%
4	12,960	63,296	763,441	305,024	763,441	39.95%	9.99%
8	14,144	32,144	383,185	185,152	383,185	48.32%	12.08%
16	13,101	15,860	193,406	115,844	193,406	59.90%	14.97%

interested in the proportions of the execution steps that can be back-adjusted through the non-profiled walltime usage.

We profiled the four phases of our IPU code, i.e., PDE_{compute}, PDE_{exchange}, ODE_{compute}, ODE_{exchange}, respectively based on the computing and exchange phases of the PDE and ODE parts. Table II.2 gives a breakdown of the PDE and ODE phases. For example, when using 1 IPU, the PDE part took about 25.8% of the total execution time, while the data exchanges within the PDE steps only took about 1.1%. The compute phase of a PDE step consumes about 22× as much time as the exchange phase. The ODE computation is by large the slowest part, requiring approximately 74.2% of the total execution time. Recall that four PDE steps were executed per ODE step, each ODE step thus requires about 11.5× the time of a single PDE step. No communication is required to start the ODE step after the preceding PDE step has finished. Therefore, ODE_{exchange} is 0. We also noticed that the execution of the ODE step starts without any synchronization.

The PDE_{compute} phase took an average of $238K$ cycles per tile, where the fastest tile finished after $231K$ cycles and the last after $245K$. The standard deviation is $5.2K$ cycles. However, workload imbalances have no significant impact on the performance, because the ODE part can start without requiring a global synchronization. ODE_{compute} took on average $2.8M$ cycles per tile, with a minimum of $2.7M$ and a maximum of $2.9M$ cycles. The standard deviation for ODE_{compute} is $63.1K$ cycles.

If we quantify effectiveness as the percentage of time during which the tiles on average remain non-idle, then the effectiveness is 97% for both the ODE_{compute} and PDE_{compute} phases. However, the effectiveness is only 57.4% for the PDE_{exchange} phase. These numbers are associated with the single-IPU experiment. Using more IPUs will see lower effectiveness particularly for the PDE_{exchange} phase, due to a more unbalanced distribution of the halo cells. This problem is currently not properly handled by the mainstream mesh partitioners.

Breaking down the phases of the strong-scaling experiments in Figure II.6 shows that the share of communication (Ex) time increases. This increasing communication share can be explained due to the ladder topology (see Figure II.8), which increases the latencies linearly with the number of IPUs used.

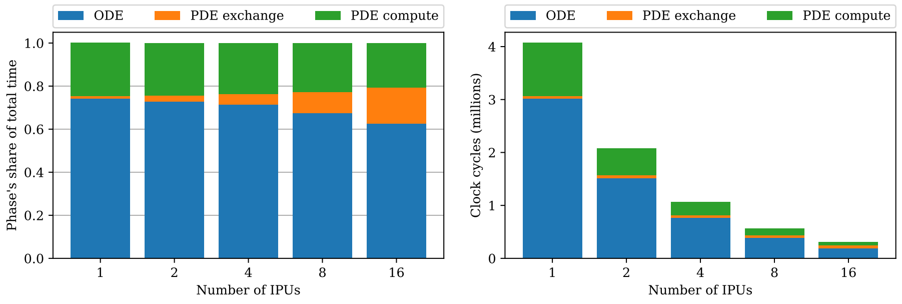

Figure II.6: Breakdown of the strong-scaling experiments using the `heart04` mesh. Blue denotes the total ODE step time composed of only compute, yellow is the compute phase of the PDE step and red denotes the cycles used for the PDE exchanges.

Table II.3: Performance comparison between GC200 IPUs and A100 GPUs, related to a monodomain simulation using the `heart04` mesh with 25,000 ODE steps and 100,000 PDE steps.

IPUs	Total time	PDE part	ODE part	GPUs	Total time	PDE part	ODE part
1	76.57 s	19.75 s	56.82 s	1	37.40 s	27.34 s	10.05 s
2	38.99 s	10.55 s	28.44 s	2	18.77 s	13.26 s	5.51 s
4	20.06 s	5.73 s	14.33 s	4	9.75 s	6.89 s	2.86 s
8	10.71 s	3.49 s	7.22 s	8	9.54 s	8.81 s	0.73 s
16	6.02 s	2.26 s	3.76 s				

II.7.4 Detailed Performance Comparison between IPUs and GPUs

Figure II.5 has already shown a high-level comparison of the performance between IPUs and GPUs. Now, we want to provide a more detailed performance comparison between the two processor architectures, using dissected time measurements of the strong-scaling test with the `heart04` mesh.

It can be seen in Table II.3 that the GC200 IPUs and A100 GPUs behave very differently for the PDE and ODE parts. (Note that the PDE part includes the time spent on halo-data exchanges.) The GC200 IPU is considerably more powerful than the A100 GPU for running the SpMVs that constitute the PDE part. We remark that the GPU version of the monodomain simulator is highly optimized as studied in [Hus19]. This can be confirmed by a simple study on the memory traffic. Namely, let us assume perfect data reuse in the L2 cache of the A100 GPU, i.e., each value of the input vector $\vec{v}_n$ to the SpMV (II.5) is loaded exactly only once from the GPU's device memory per SpMV. This will produce an idealized lower-bound estimate of the memory traffic on the GPU, which can in turn be translated to a minimum bandwidth of 1507.48 GB/s that has been achieved when one A100 GPU is used. Compared with the realistically achievable memory bandwidth of 1774.37 GB/s per A100 GPU, which is measured by the BabelStream micro-benchmark [22a], we can conclude that the GPU implementation of the SpMV has achieved *at least* 85% of the realistically achievable maximum performance. Considering the high level of A100's SpMV performance, it is very impressive that the GC200 IPU beats the A100 GPU in this regard. The explanation lies in the aggregate on-chip SRAM bandwidth of IPU. On the other hand, the GPU performance of the ODE part is much higher than the IPU counterpart, thanks to the GPU's tremendous floating-point capability. One remark, however, is that the GPU implementation can allow a certain level of overlap between the halo-data exchange and the ODE step, which may make the ODE time measurement on the GPU seem shorter than it actually is. This is evident for the ODE time measurement in Table II.3 when 8 GPUs are used.

Last but not least, we have also taken a closer look at the simulated results that are produced by the IPU and GPU implementations. (Three example snapshots from the simulation are shown in Figure II.5.) Human eyes can not detect any difference between the two numerical solutions. During the entire simulation that spans 500 ms, the largest maximum difference between the IPU-produced and GPU-produced v values is found to be 0.18 mV. Considering that the simulated v values lie in the range of [-90 mV, 40 mV], this small discrepancy is acceptable.

II.8 Conclusion

In this work, we have ported an existing simulator of cardiac electrophysiology to Graphcore IPUs. In this process, we investigated the Poplar programming needed and the impact of partitioning and halo-data exchange on SpMV operations that arise from unstructured computational meshes. The speed and accuracy of some special math functions were also rigorously checked on IPUs. These topics are by no means constrained to the particular cardiac simulator, but with a good possibility of becoming useful for other computational physics applications.

II.8.1 Limitations

There are several limitations of the present work. First, the SpMV operations that have been ported to the IPU use the ELLPACK format to store the off-diagonal part of a sparse matrix. This is due to the cell-centered finite volume discretization adopted for the diffusion equation, resulting in the same number of non-zeros per matrix row. In the case of node-centered finite volume discretization or finite element discretization in general, the number of non-zeros per matrix row will no long be the same. For example, the standard *compressed sparse row* format can then be used to store the resulting sparse matrix. This will however require a change in the partitioning step, where each row should be weighted by its number of non-zeros. Thus the sub-meshes can be assigned with different numbers of rows. Such a weighted partitioning scheme actually suits the IPU very well, because the computing cost per tile will be strictly proportional to the total number of non-zeros assigned. The GPU counterpart, on the other hand, may need to adopt other storage formats to achieve its full memory bandwidth capacity. (This is an active research field illustrated by many recent publications such as [Moh+21].)

Second, still with respect to the partitioning step, the present work has another limitation related to using multiple IPUs. Our current partitioning strategy is *single-layered*, i.e., an unstructured mesh is decomposed into as many pieces as the total number of IPUs available on multiple IPUs. No effort is made to limit the halo-data exchanges that span between IPUs. As shown in Figure II.8 in Section II.B.1, the communication speed between IPUs is heterogeneous and much lower than the intra-IPU communication speed between the tiles. An idea for improvement is to introduce a *hierarchical partitioning* scheme, where a first-layer partitioning concerns only the division between the IPUs, whereas a second-layer partitioning divides further for the tiles within each IPU. Moreover, the partitioning on both layers should attempt to evenly distribute the halo cells. Otherwise, tiles over-burdened with halo cells will quickly become the bottlenecks.

Third, we have only considered explicit time integration for the diffusion equation. The up-side is that the PDE solver only needs SpMVs, but the down-side is the severe stability restriction on the size of Δt_{PDE}. Implicit time integration is good to have with respect to numerical stability, but will give rise to linear systems that need to be solved. Here, we remark that SpMVs constitute one of the building blocks of any iterative linear solver.

II.8.2 Lessons Learned

The design of algorithms must be reconsidered for massively tiled processors. While the HPC community already has ample experience in using technologies like distributed memory and BSP, their use in the IPU has not been explored equally well. Most of current projects and their underlying design considerations are not adjusted to the tradeoffs of this new class of accelerators. New ways to think of communication and load balance are necessary.

Furthermore, Poplar programming requires us to implicitly define communication at compile time. This makes it impossible to have predefined kernels such as those commonly found on CPUs and GPUs. One could argue that the compilation of regular-mesh kernels for IPUs only needs to happen once for all inputs. However, compiling for irregular meshes is required for every single communication scenario unless a regular representation can be found. This unavoidable compilation requires us to be aware of the expensive compilation time. We also found that the compilation time and mesh preprocessing time substantially increase with multiple IPU devices.

The current IPU architecture also has clear limitations. These include the support of only single-precision computing and limited SRAM resource per tile. Instead of waiting for Graphcore to develop new IPUs capable of double-precision computing, a possible strategy can be to identify the most accuracy-critical parts of a computation and emulate double-precision operations by software for these parts, whereas the remaining parts use single precision. Better partitioning algorithms, which adopt a hierarchical approach and are aware of the heterogeneity in the communication network, will also be useful for optimizing the usage of the limited SRAM per tile. On the other hand, we should not forget about one particular advantage of AI processors such as IPUs. That is, these processors are already good at running machine-learning workloads. Thus, if “conventional" scientific computing tasks can be efficiently ported to AI processors, the distance to converged AI and HPC is short.

In future work, we will investigate further optimizations of halo-data exchanges both on and between the IPUs (with help of better partitioning strategies) and extend our work to other more general unstructured-mesh computations.

Conflict of Interest Statement

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Author Contributions

All the authors contributed to the conceptualization of the work. LB designed and implemented the IPU code, performed the related experiments and collected the measurements. The original GPU version of the Lynx code was designed and implemented by JL, with the ODE part added by KGH. KGH performed all the GPU experiments. All the authors participated in the discussions and analyses of the experimental results. All the authors contributed to the writing of the manuscript.

Funding

The work of the second author was partially supported by the European High-Performance Computing Joint Undertaking under grant agreement No. 955495 and the Research Council of Norway under contract 329032. The work of the third and fourth authors was partially supported by the European High-Performance Computing Joint Undertaking under grant agreement No. 956213 and the Research Council of Norway under contracts 303404 and 329017. The research presented in this paper has benefited from the Experimental Infrastructure for Exploration of Exascale Computing (eX3), which is financially supported by the Research Council of Norway under contract 270053.

Acknowledgments

Julie J. Uv segmented the surface files that were used to generate fibre directions for the bi-ventricular meshes. James D. Trotter generated the meshes with fibre

directions.

Data Availability Statement

The original contributions presented in the study are included in the article/supplementary material, and further inquiries can be directed to the corresponding author.

References

[21] *Designed for AI–Intelligence Processing Unit.* https://www.graphcore.ai/products/ipu. 2021.

[22a] *BabelStream.* https://github.com/UoB-HPC/BabelStream. 2022.

[22b] *The Future of AI is Here.* https://cerebras.net/chip/. 2022.

[AIM17] Abadi, M., Isard, M., and Murray, D. G. "A computational model for TensorFlow: an introduction". In: *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages.* 2017, pp. 1–7.

[Bay+12] Bayer, J. D. et al. "A novel rule-based algorithm for assigning myocardial fiber orientation to computational heart models". In: *Ann Biomed Eng* vol. 40 (Oct. 2012), pp. 2243–2254.

[BCL21] Burchard, L., Cai, X., and Langguth, J. "iPUG for Multiple Graphcore IPUs: Optimizing Performance and Scalability of Parallel Breadth-First Search". In: *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC).* 2021, pp. 162–171.

[BP00] Bern, M. and Plassmann, P. "Chapter 6–Mesh Generation". In: *Handbook of Computational Geometry.* Ed. by Sack, J.-R. and Urrutia, J. North-Holland, 2000, pp. 291–332.

[Bul+16] Buluç, A. et al. "Recent Advances in Graph Partitioning". In: *Algorithm Engineering: Selected Results and Surveys.* Vol. 9220. Lecture Notes in Computer Science. Springer, 2016, pp. 117–158.

[Bur+21] Burchard, L. et al. "iPUG: Accelerating Breadth-First Graph Traversals Using Manycore Graphcore IPUs". In: *International Conference on High Performance Computing.* Springer. 2021, pp. 291–309.

[Cla+11] Clayton, R. H. et al. "Models of cardiac tissue electrophysiology: Progress, challenges and open questions". In: *Progress in Biophysics and Molecular Biology* vol. 104 (Jan. 2011), pp. 22–48.

[Fas+21] Fasi, M. et al. "Numerical behavior of NVIDIA tensor cores". In: *PeerJ Computer Science* vol. 7 (2021), e330.

[GKY79] Grimes, R. G., Kincaid, D. R., and Young, D. M. *ITPACK 2.0 User's Guide*. Center for Numerical Analysis, University of Texas, 1979.

[Hus19] Hustad, K. G. "Solving the monodomain model efficiently on GPUs". MA thesis. http://urn.nb.no/URN:NBN:no-74080: University of Oslo, 2019.

[Håp21] Håpnes, S. "Solving Partial Differential Equations by the Finite Difference Method on a Specialized Processor". MA thesis. http://urn.nb.no/URN:NBN:no-92505: University of Oslo, 2021.

[IEE08] IEEE. "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2008* (2008), pp. 1–70.

[JAM22] Jacquelin, M., Araya-Polo, M., and Meng, J. *Massively scalable stencil algorithm*. arXiv, 2022.

[Jia+19] Jia, Z. et al. "Dissecting the Graphcore IPU architecture via microbenchmarking". In: *arXiv preprint arXiv:1912.03413* (2019).

[KK98] Karypis, G. and Kumar, V. "A fast and high quality multilevel scheme for partitioning irregular graphs". In: *SIAM Journal on Scientific Computing* vol. 20, no. 1 (Aug. 1998), pp. 359–392.

[Lan+15a] Langguth, J. et al. "Parallel performance modeling of irregular applications in cell-centered finite volume methods over unstructured tetrahedral meshes". In: *Journal of Parallel and Distributed Computing* vol. 76 (Feb. 2015), pp. 120–131.

[Lan+15b] Langguth, J. et al. "Scalable Heterogeneous CPU-GPU Computations for Unstructured Tetrahedral Meshes". In: *IEEE Micro* vol. 35, no. 4 (2015), pp. 6–15.

[LM21] Louw, T. R. and McIntosh-Smith, S. N. "Using the Graphcore IPU for Traditional HPC Applications". In: 3rd Workshop on Accelerated Machine Learning: Co-located with the HiPEAC 2021 Conference, AccML; Conference date: 18-01-2021. 2021.

[Mar+19] Martinez-Navarro, H. et al. *Repository for modelling acute myocardial ischemia: simulation scripts and torso-heart mesh*. https://ora.ox.ac.uk/objects/uuid:951b086c-c4ba-41ef-b967-c2106d87ee06. 2019.

[Mes21] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*. June 2021.

[Mir+12] Mirin, A. A. et al. "Toward real-time modeling of human heart ventricles at cellular resolution: simulation of drug-induced arrhythmias". In: *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE. Nov. 2012.

[Moh+20] Mohan, L. R. M. et al. “Studying the potential of Graphcore IPUs for applications in particle physics”. In: *arXiv preprint arXiv:2008.09210* (2020).

[Moh+21] Mohammed, T. et al. “DIESEL: A novel deep learning-based tool for SpMV computations and solving sparse linear equation systems”. In: *The Journal of Supercomputing* vol. 77, no. 6 (2021), pp. 6313–6355.

[MZS12] Marsh, M. E., Ziaratgahi, S. T., and Spiteri, R. J. “The secrets to the success of the Rush–Larsen method and its generalizations”. In: *IEEE Transactions on Biomedical Engineering* vol. 59, no. 9 (June 2012), pp. 2506–2515.

[Nie+11] Niederer, S. A. et al. “Verification of cardiac tissue electrophysiology simulators using an N-version benchmark”. In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* vol. 369, no. 1954 (2011), pp. 4331–4351.

[Ort+20] Ortiz, J. et al. “Bundle Adjustment on a Graph Processor”. In: *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, 2020, pp. 2413–2422.

[RL78] Rush, S. and Larsen, H. “A practical algorithm for solving dynamic membrane equat ions”. In: *IEEE Transactions on Biomedical Engineering*, no. 4 (1978), pp. 389–392.

[Roc+20] Rocki, K. et al. “Fast Stencil-Code Computation on a Wafer-Scale Processor”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’20. Atlanta, Georgia: IEEE Press, 2020.

[ten21] ten Tusscher, K. H. W. J. *Source code second version Human Ventricular Cell model*. http://www-binf.bio.uu.nl/khwjtuss/SourceCodes/HVM2/ [Accessed on July 23, 2021]. 2021.

[TP06] Tusscher, K. H. ten and Panfilov, A. V. “Cell model for efficient simulation of wave propagation in human ventricular tissue under normal and pathological conditions”. In: *Physics in Medicine & Biology* vol. 51, no. 23 (2006), p. 6141.

[Una+17] Unat, D. et al. “Trends in Data Locality Abstractions for HPC Systems”. In: *IEEE Transactions on Parallel and Distributed Systems* vol. 28, no. 10 (2017), pp. 3007–3020.

[Val90] Valiant, L. G. “A bridging model for parallel computation”. In: *Communications of the ACM* vol. 33, no. 8 (1990), pp. 103–111.

Authors’ addresses

Luk Burchard Simula Research Laboratory, Kristian Augusts gate 23, 0164 Oslo, Norway, luk@simula.no

Kristian Gregorius Hustad Simula Research Laboratory, Kristian Augusts gate 23, 0164 Oslo, Norway, kghustad@simula.no

Johannes Langguth Simula Research Laboratory, Kristian Augusts gate 23, 0164 Oslo, Norway, langguth@simula.no

Xing Cai Simula Research Laboratory, Kristian Augusts gate 23, 0164 Oslo, Norway, xingca@simula.no

Appendix II.A Algorithmic Description of Three Separator Partitioning Methods

The description in Algorithm 2 corresponds to Section II.4.1. The three schemes are for reordering the separator cells and generating outgoing ranges over segments of memory (see Figure II.2 in Section II.4.1). The *deps* parameter is a multiset where each $deps_j$ is a set that indicates for a tetrahedron $tets_j$ the partitions p_i with index $i \in deps_j$ that requires the value of the tetrahedron j. We denote the subrange of a multiset X from the n-th to m-th element inclusive with $X(n, m)$.

Appendix II.B Technical Information of Graphcore IPUs

II.B.1 Architectural Overview

Developed as an accelerator for machine intelligence, the Graphcore IPU is a massively parallel, multiple-instruction-multiple-data (MIMD) processor that consists of a large number of independent units called *tiles*. Each tile features a core and a small amount of private SRAM used as scratchpad memory. Although the entire device has access to slower DRAM, the SRAM of the individual tiles jointly make up the primary device memory, thereby eschewing the need for a traditional cache hierarchy.

In this paper, we use the second-generation IPU, named GC200. It has in total 1472 tiles organized into *islands* and *columns*, as shown in Figure II.7. Each tile has 624 KB of SRAM, for a total of 918 MB per IPU. Both cores and SRAM run at 1.33 GHz, giving the device an aggregate memory bandwidth of 47.5 TB/s. However, data that is not local to a core must be communicated between the tiles via the IPU exchange network. In previous work, architectural details of the first-generation IPU called GC2 was exhaustively studied and benchmarked [Jia+19].

A tile is capable of sending and receiving 4 bytes per cycle, which amounts to 5.3 GB/s for a total of 7.83 TB/s for all 1472 tiles. Moreover, the IPU alternates between computation and communication in a bulk-synchronous parallel (BSP) [Val90] manner with no overlap between the phases.

Each core runs six concurrent threads in a fine-grained *temporal multithreading* scheme for a total of 8832 threads. Unlike simultaneous multithreading, which is commonly used in modern CPU and GPU designs, IPU threads are scheduled

Algorithm 2 Three schemes for reordering the separator on a tile and generating outgoing ranges

```
 1: procedure AssignFullRange($tets, deps$)          ▷ dependencies for each tet.
 2:     for $dest \in \bigcup_{j=1}^{N} \bigcup_{j'=1}^{|deps_j|} deps_{j,j'}$ do
 3:         $p_{dest} \leftarrow tets(0, N)$
 4:     end for
 5: end procedure
 6:
 7: procedure AssignRanged($tets, deps$)
 8:     for $dest \in \bigcup_{j=1}^{N} \bigcup_{j'=1}^{|deps_j|} deps_{j,j'}$ do
 9:         $L \leftarrow min\{i : dest \in deps_i\}$
10:         $U \leftarrow max\{i : dest \in deps_i\}$
11:         $p_{dest} \leftarrow tets(L, U)$
12:     end for
13: end procedure
14:
15: procedure AssignMixed($tets, deps$)
16:     $mixed \leftarrow \{\}$
17:     $singleton \leftarrow \{\{\}, \ldots, \{\}\}, |singleton| = |\bigcup_{j=1}^{N} \bigcup_{j'=1}^{|deps_j|} deps_{j,j'}|$
18:     for $t \in tets$ do
19:         if $|dest_t| = 1$ then
20:             $singleton \leftarrow singleton \cup \{t\}$
21:         else
22:             $mixed \leftarrow mixed \cup \{t\}$
23:         end if
24:     end for
25:     for $dest \leftarrow \bigcup_{j=1}^{N} \bigcup_{j'=1}^{|deps_j|} deps_{j,j'}$ do
26:         $p_{dest} \leftarrow singleton_{dest}$
27:         $p_{dest} \leftarrow mixed$
28:     end for
29: end procedure
```

consecutively in a fixed order. For that reason, the design is also referred to as a *barrel processor*. IPU instructions, including loads and stores from the local tile memory, take exactly 6 cycles. Thus, individual threads do not experience latency since they execute one instruction per cycle in which they are scheduled. Similar to the tensor cores on recent NVIDIA GPUs, the IPU has *Accumulating Matrix Product* units which enable a peak performance of 62.5 TFLOPS/s in 32-bit precision for dense matrix multiplications. Other floating-point operations are performed significantly slower though. With one FMA (fused-multiply-add) operation per clock cycle per tile, the IPU reaches approximately 4 TFLOPS/s, which is substantially slower than comparable GPUs.

Four IPUs are deployed together in an IPU-M2000 blade, which in turn can be combined into larger systems called IPU PODs. Each M2000 contains the

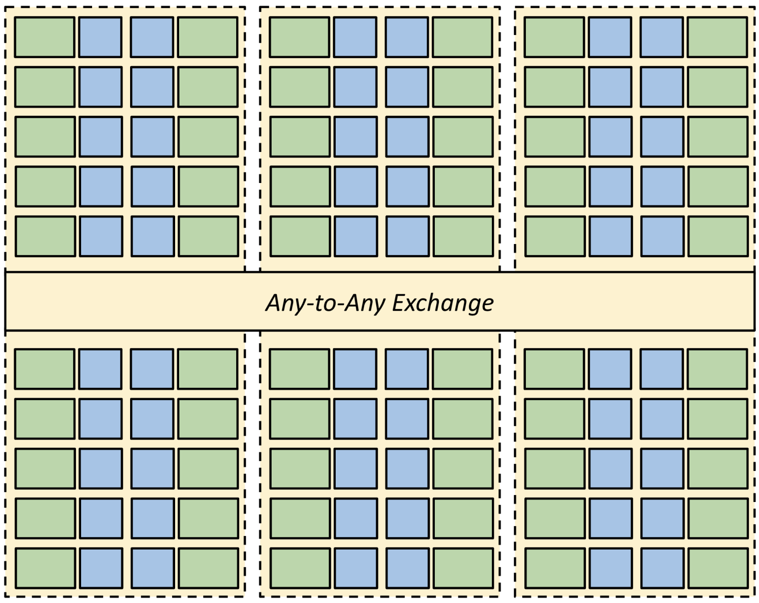

Figure II.7: Tile layout on the GC200 IPU processor. Each of the 1472 tiles (green/blue) has 624 KB of directly accessible memory and a core with 6 threads, temporarily scheduled through a barrel processor. All tiles are connected through 7.83 TB/s interconnect.

Gateway Link, a network adapter that connects the M2000 to a CPU host that controls the IPU system. The Gateway link has a nominal bandwidth of 32 GB/s to each IPU, and the network connecting the Gateway to the host has 100 Gb/s (12.5 GB/s). The M2000 also contains up to 448 GB of DRAM memory, which it can access at a speed of about 20 GB/s.

Between the IPUs, data is transferred via the IPU-link, which performs both intra-blade and inter-blade communication. The IPU-link thus corresponds to both PCIe and Infiniband in CPU/GPU systems (or alternatives such as NVIDIA NVLink and CRAY Shasta). Each IPU has 10 IPU-links with a total bandwidth of 320 GB/s. Pairs of IPUs are connected with 12 links, which amount to a bandwidth of 192 GB/s. Therefore, 8 links are left in the system to connect to other IPUs. These connections use double-link cables. Thus they operate at 64 GB/s. Up to 32 such pairs can be connected in a ladder configuration with a bisection bandwidth of 128 GB/s, see Figure II.8 for an example. The ladder can be closed to form a torus, which doubles the bisection bandwidth. Note that a single IPU has 150 W TDP, which is approximately half of a competitive GPU. Thus, with respect to power consumption, each IPU pair is comparable to one powerful GPU, such as the NVIDIA V100 or A100.

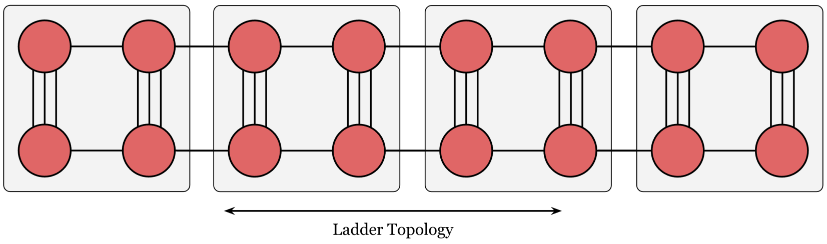

Figure II.8: A single IPU is denoted as a red circle, and the lines represent IPU-links each with a bandwidth of 64 GB/s. A double-IPU configuration is connected via three 64 GB/s lanes, with an aggregate bandwidth of 192 GB/s. Each box denotes a blade and multiple blades are connected with two 64 GB/s cables, extending horizontally in a ladder configuration.

II.B.2 Execution Latency

The IPU is a time-multiplexed barrel processor, which hides instruction latency by round-robin cycling through 6 hardware contexts, each taking six cycles to execute. Thus, the true execution latency is six cycles; normalized, a hardware context has a frequency of $\frac{1.33GHz}{6} \approx 221$ MHz. In the following section, we refer to time-multiplexed latencies as one cycle even though it takes six global cycles to execute them.

A tile on the IPU has two execution pipelines, one for integer and management instructions (main), the other for floating-point operations (aux). These two pipelines run together and can issue instructions on both pipelines during the same cycle through *instruction-bundles*, which contain two instructions.

All instructions issued in the main pipeline have a execution time of a single cycle. The main pipeline executes all memory load and store instructions up to quad-register wide loads of 128 bit. However, the platform does not have speculative execution or other non-deterministic optimizations often found with CPUs. Furthermore, branches and conditional execution are also executed within one cycle.

The aux pipeline, used for floating-point operations, executes instructions of different latencies. Most instructions only require a single cycle, but instructions commonly used in scientific computing can take up to 6 cycles. The one-cycle lower bound for these instructions is an in-hardware-handled special case, e.g., calling a math function with a special operand such as `exp((float32) 0.0)`. In particular, our measurements show that `div`, `exp2`, and `exp` normally require 3 cycles on the Graphcore IPU. Moreover, `sqrt` and `tanh` require 5 cycles, whereas `log` requires 6 cycles.

II.B.3 Execution Dependency

Pipelining is a commonly used technique for processors to hide the latencies of instructions by simultaneously executing multiple instructions (at different stages) such that they are interleaving each other. Interleaving instructions can be trivially done when there are no dependencies between them. In the common scenario, the IPU does not need interleaving as all instructions are completed within one cycle. However, some instructions (see Section II.B.2) require more time to execute; therefore a form of pipelining would be beneficial.

To determine if the IPU is capable of instruction interleaving, we can choose an instruction that runs over multiple cycles and can be interleaved. We then measure the cycles this instruction needs. A form of simple interleaving can occur in two ways:

1. The same AUX pipeline could be used for a different instruction bundle.
2. Only the MAIN pipeline is capable of interleaving while the AUX pipeline is busy.

For case 1, we added a one-cycle AUX instruction with independent registers after the long-running instruction. We observed that the cycle count proportionally increased. Therefore, we can conclude that no interleaving on the AUX pipeline exists. For case 2, a one-cycle MAIN instruction with independent registers after the long-running instruction also increased the cycle count.

This experiment shows that a long-running instruction bundle blocks both the MAIN and AUX pipelines. Therefore, we can count the instructions to determine the runtime under the assumption that the operand values are known. This is used in Section 4.3 where we develop performance models of the monodomain simulator when executed on IPU.

II.B.4 Poplar Programming

Programming can be done through Poplar, the lowest level C++ library to interact with the IPU. Poplar adheres to the BSP structure of the underlying hardware. Following machine learning frameworks like TensorFlow or Pytorch [AIM17], Poplar also adopts a dataflow architecture with mutable state. The underlying hardware will then schedule this dataflow graph in a BSP fashion.

Formally there are three elements in such a dataflow graph: *tensors* that hold data in an n-dimensional array, *compute-verticies* that specify the computations, and *edges* that connect the tensors with their vertices. Compute vertices can modify an incoming tensor or create a new one. Poplar calls the underlying code to a compute vertex a *codelet*.

All *tensors* and *compute-verticies* are mapped to tiles on the physical chip with the assigned tile-local memory. Therefore, when an edge in the dataflow graph is created between *tensors* and a *compute-vertex*, the data will be transferred to the tile executing the *compute-vertex*. Globally, this can create duplicated data.

It is possible to view the dataflow graph as a two-colored level-wise alternating directed graph through time. Each layer either contains data in the form of *tensors*, and the next layer contains the *compute-verticies*. We can map a pair of these layers into a BSP-superstep, where we synchronize the computation before exchanging the data between the *tensors* and *compute-verticies* in the next BSP-superstep.

It is also possible to have control flow within the dataflow graph. Operations, such as branches, operate on the value of a *tensor* and are executed as a BSP-superstep, which is introduced by the compiler. A variety of basic operations are available such as `WHILE` and `IF`. The control-flow operations are globally synchronized, which get more expensive when more devices are connected, as the synchronization becomes more costly. However, when a decision variable is known on all IPUs, it is possible to make this IPU-local without requiring a global synchronization phase.

The Poplar programming model makes the data exchanges implicit by defining a *tensor* as an input to a *compute-vertex* mapped to a different tile. The data-exchange code is statically generated by the compiler before runtime. All exchanges happen simultaneously and take place after a synchronization step. This static graph compilation makes it impossible to do dynamic data exchanges or sparse communication. One workaround is to create different compute groups of compute vertices, all using a different amount of data and switching between them. However, this strongly assumes that sparsity occurs evenly throughout all partitions and that the reduced communication time offsets the additional synchronization phase for a global branch. The compiler can optimize simple cases, for example, merging multiple BSP-supersteps into one if no outer-tile exchange occurs.

Writing a user-defined function for a *compute-vertex* happens in the form of a C++ *codelet*. The *codelet* is defined as a templated C++ class. The user can define multiple class member variables. The compiler will fill in these member variables, and appropriate fields will be created on the *compute-vertex* to connect *tensors* in the computational graph. The *tensors* in the *codelet* are accessible in a row-major C style array. Once the *compute-vertex* in the graph gets called, an entry function in the *codelet* is executed, which can then modify data available during the execution of the *codelet*. After completion, the data from the output *tensors* are copied to their respective owning tile.

Paper IV

Space Efficient Sequence Alignment for SRAM-Based Computing: X-Drop on the Graphcore IPU

Luk Burchard[†], Max Xiaohang Zhao[†], Johannes Langguth, Aydın Buluç, Giulia Guidi

To appear in *SC '23: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*.

Abstract

Dedicated accelerator hardware has become essential for processing AI-based workloads, leading to the rise of novel accelerator architectures. Furthermore, fundamental differences in memory architecture and parallelism have made these accelerators targets for scientific computing.

The sequence alignment problem is fundamental in bioinformatics; we have implemented the X-Drop algorithm, a heuristic method for pairwise alignment that reduces search space, on the Graphcore Intelligence Processor Unit (IPU) accelerator. The X-Drop algorithm has an irregular computational pattern, which makes it difficult to accelerate due to load balancing.

Here, we introduce a graph-based partitioning and queue-based batch system to improve load balancing. Our implementation achieves 10× speedup over a state-of-the-art GPU implementation and up to 4.65× compared to CPU. In addition, we introduce a memory-restricted X-Drop algorithm that reduces memory footprint by 55× and efficiently uses the IPU's limited low-latency SRAM. This optimization further improves the strong scaling performance by 3.6×.

The work of the first author was partially supported by the European High-Performance Computing Joint Undertaking under grant agreement No. 955495 and the Research Council of Norway under contract 329032. The work of the third author was partially supported by the European High-Performance Computing Joint Undertaking under grant agreement No. 956213 and the Research Council of Norway under contracts 303404 and 329017. The research presented in this paper has benefited from the Experimental Infrastructure for Exploration of Exascale Computing (eX3), which is financially supported by the Research Council of Norway under contract 270053.

[†]Shared first author, both authors contributed equally.

Contents

IV.1 Introduction

Today's architectures are complex but often suboptimal for modern irregular computation, being overprovisioned for arithmetic computation and challenging the programmer to cope with the high cost of moving data. A clear insight into this problem is provided by the Top500 list, in which the world's 10 fastest machines achieve peak performance of up to 83% in the computationally intensive LINPACK benchmark but no more than 3% peak performance in the High-Performance Conjugate Gradient (HPCG) benchmark, which involves irregular computation [Meu+01].

In the last decade, the Graphics Processing Unit (GPU) has emerged as a leading architecture for high-performance computing (HPC) challenges involving dense linear algebra and scientific computing [VN14]. However, GPUs are single instruction multiple data (SIMD) architectures, and this can be a limitation for computational challenges that suffer from high load imbalance, as is often the case with data analytics and general computation. They require regular data access and work pattern to reach their theoretical peak performance [Lee+10]. A general-purpose processor such as a CPU is better suited for non-uniform data access but does not provide the high instruction throughput achieved by GPUs, since CPUs are optimized for latency rather than throughput. Therefore, new architectures are needed for HPC that can provide more flexible acceleration like CPUs while providing high throughput like GPUs.

Recently, the Graphcore Intelligence Processing Unit (IPU), a massively parallel multiple instruction multiple data (MIMD) SRAM-based processor designed as an AI accelerator, has emerged as a potential solution to irregular computation by combining fine-grained memory access with wide parallelism [Jia+19]. While processors connected to external RAM are constrained by the von Neumann bottleneck, SRAM-based computing eschews complex memory hierarchies by providing sufficient SRAM storage on the processing chip to fit a problem instance [Mit+21]. IPUs were developed for AI applications but showed potential for other applications, such as for the breadth-first search algorithm, stencil computations, and cardiac simulation [Bur+21; Bur+23; LM21]. The question we seek to answer in this work is if new SRAM-based architectures can improve performance on a wider range of emerging HPC challenges, such as bioinformatics.

In bioinformatics, pairwise sequence alignment is a fundamental technique used in many scenarios, such as genome assembly, phylogenetic analysis, protein structure prediction based on homology, and searching for similar sequences in databases [AG98]. Long-read sequencing technologies are increasingly available, producing sequences with an average length of about $15,000$-$30,000$ base pairs (bp). Longer sequences allow for more precise genome assembly [Ama+20] but they come with increased algorithmic complexity and computational cost. Therefore, there is a need for efficient algorithms that can handle long-read sequencing data.

The Needleman-Wunsch (NW) algorithm is used for finding the best global alignment, while the Smith-Waterman (SW) algorithm is used for finding the best local alignment. There is also a version of pairwise alignment called semi-global, where one side of the sequences is forced to align, but the other is not. However, finding the optimal solution for these algorithms requires quadratic time as a function of sequence length, which is inefficient for long sequences. The critical role that sequence alignment plays in understanding protein and DNA sequences has made it a focal point in attempts to optimize both algorithms and hardware [Als+21]. In practice, assumptions can be made based on the input data and the type of computation desired to create heuristic algorithms with subquadratic runtimes.

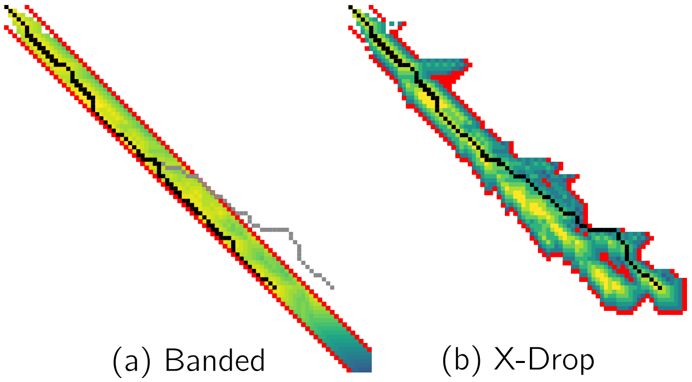

Figure IV.1: On the left, the alignment is forced to stay within the banded area regardless of the score, missing the optimal alignment (gray). On the right, when the score (yellow-blue) X goes below the current best score, the search is terminated (red boundary), and the optimal alignment (black) is returned.

A popular heuristic that we target in this work is called X-Drop. The X-Drop algorithm is a heuristic for restricting the search space of a semi-global alignment algorithm. It reduces the quadratic cost by dynamically searching only for a high-quality alignment and stopping the computation early when a good alignment is impossible. This allows for a more dynamic fit to the data than a static search space (Figure IV.1). It is a promising algorithm for long-read sequencing data because a good alignment can be found in nearly linear time. Genomic pipelines already use X-Drop and its variants Y-Drop and Z-Drop due to their good alignment quality and fast runtime [Alt+90; Gui+20; Li18; Sel+20].

In the literature, we only found one implementation of the X-Drop algorithm on GPUs [Zen+20], and one on Field Programmable Gate Arrays (FPGAs) [Zen+21], which can only run on DNA sequences (no protein sequences). However, X-Drop is widely implemented on CPUs [Alt+90; Li18; Rei+17]. In this paper, we present an implementation of the X-Drop algorithm on a novel AI accelerator hardware, the Graphcore IPU, that is suitable for both DNA and protein alignment. The proposed IPU-based approach provides a competitive solution for accelerating X-Drop on a wider range of problem instances compared to traditional CPUs and GPUs. Here we demonstrate the practicality of our implementation by integrating it into two distributed-memory pipelines with high alignment volumes: ELBA, a *de novo* long-read genome assembler [Gui+20; Gui+22], and PASTIS, a protein similarity search engine [Sel+20]. Our implementation is tested on a variety of real-world data, reporting speedup over state-of-the-art CPU and GPU implementations. Our work thus demonstrates the potential of AI architectures to accelerate the pairwise alignment of long sequences and their suitability for irregular scientific computations.

IPUs and other SRAM-based devices, such as Cerebras hardware [Lau21], rely on the MIMD paradigm and a large SRAM, making them more effective for handling irregular computation. In this work, we not only highlight the advantages of SRAM-based computing for scientific computing and the speedup achieved for the X-Drop algorithm but also show where these novel devices need progress to be widely deployed. For example, improving the IPU interconnect would lead to significantly cheaper host-to-device transfer times.

Our contributions are as follows:

- First, we demonstrate the first application of a cluster of Graphcore IPUs for high-performance processing of irregular genomic data.
- Then, we present a memory-restricted version of X-Drop which reduces the required memory by a factor of up to $55\times$. This enables the algorithm to run in the IPU's SRAM memory.
- To solve the host-device communication bottleneck, to the best of our knowledge, we are the first to treat sequence comparisons as a graph and perform graph partitioning to reduce data transfer.
- Finally, we integrate our algorithm into ELBA and PASTIS, two state-of-the-art bioprocessing pipelines.

IV.2 Background

In this section, we first describe the architectural features of the Graphcore IPU and give a definition of the X-Drop pairwise alignment problem. Then, we briefly describe two biological pipelines that use x-drop alignment and into which we have integrated our approach.

IV.2.1 Graphcore IPU

The Graphcore IPU is a massively parallel multiple instruction multiple data (MIMD) processor consisting of a large number of independent units called *tiles*. Each of these tiles has a core and a small amount of SRAM memory. Instead of serving as a cache, the SRAM memories of the individual tiles together form the device memory, so no traditional cache hierarchy is required.

IV.2.1.1 Hardware

Three IPU generations have been released so far, called GC2, GC200, and BOW. Functionally, the latter differs from its predecessor only in its clock frequency. In this paper, we make use of both the GC200 and the BOW. Both recently released IPU models consist of 1472 tiles, each of which contains a core and 624 KB of SRAM which run at 1.33 GHz on the GC200 and at 1.85 GHz on the BOW. Each IPU core runs six concurrent threads in *temporal multithreading*, meaning that they are scheduled consecutively in a fixed order. The majority of IPU instructions, including loading and storing from local tile memory, take exactly six cycles. Once issued, instructions are completed the next time a thread is scheduled. Thus, the 8832 threads can be viewed as independent cores running at one-sixth of the original clock frequency without instruction or memory latency. The total SRAM per IPU is 918 MB which can be read with an aggregate memory bandwidth of 46.9 TB/s (GC200) or 65.2 TB/s (BOW). However, data that is not local to a core must be communicated between the tiles via the IPU exchange network, which has an aggregate bandwidth of 7.83 TB/s (GC200) or 10.9 TB/s (BOW). The IPU alternates between computation and communication in a bulk-synchronous parallel (BSP) [Val90] manner with no overlap between phases.

As the IPU is an accelerator that does not run its operating system, it is dependent on a host machine. Unlike GPUs, whose CPU host is typically contained inside the same machine, a group of IPUs is connected to the host node via 100 Gb/s Ethernet. This means that the number of IPUs per host can vary widely.

Our GC200 test system contains 64 IPUs, but only one dual-socket Xeon-based server. Consequently, host-to-device transfers can become a bottleneck. Four IPUs are used together in an IPU-M2000 blade, which in turn can be combined into larger systems called IPU PODs. The M2000 also contains up to 448 GB of DRAM memory, which it can access at a rate of about 20 GB/s. While this memory is too slow for most computations, it can be used to buffer data from host-to-device transfers. The IPUs themselves are connected in a ladder topology with a bisection bandwidth of 128 GB/s. In terms of power consumption, two IPUs are comparable to a powerful GPU like the NVIDIA A100 or a pair of moderately powerful CPUs.

IV.2.1.2 Programmability

Unlike other hardware accelerators, the IPU is a distributed memory system consisting of multiple tiles that use a direct memory write technique for communication.

Poplar is the C++ framework used to program the IPU at the lowest level. It is inspired by TensorFlow and the dataflow programming model, as high-level program flow is defined as a dataflow graph, where we can define a state as *Tensor* and transfer functions as *Vertex*. Code that executes in a vertex is called a codelet. One can think of a codelet as analogous to a CUDA kernel. To synchronize computation and data accesses, the IPU hardware supports the Bulk-Synchronous Parallel (BSP) programming model, which divides algorithm execution into level-synchronous supersteps with three phases: Compute, Exchange, and Synchronize.

In Poplar, each *Tensor* and *Vertex* must be mapped to a tile. The programmer must define input and output *Tensor* for the *Vertex*. The compiler uses the dataflow graph and vertex mapping to create a synchronized data exchange following the BSP pattern. A higher-level control flow can be introduced to select the next BSP superstep to execute. Unlike MPI, the data exchange does not need to be explicitly programmed.

IV.2.2 X-Drop Pairwise Alignment

The comparison of biological sequences is important for a deeper understanding of the role and function of genetic areas and protein structures, but also for the construction of the genomic sequence itself. The genome consists of strings of nucleotides (adenine, thymine, guanine, cytosine), which code for protein sequences and contain additional regulatory information. Genomes cannot be sequenced in their entire length; current sequencing technologies can only read and output sequences that are significantly shorter than the entire genome. Therefore, we need sequence alignment to reconstruct whole genomes. For short-read technologies such as Illumina, the average sequence length is 100-250 nucleotides (or base pairs, bp). In newer long-read technologies such as Pacific Biosciences and Oxford Nanopore, the average read length can be more than 20,000 bp and up to several megabases, enabling the generation of highly continuous bacterial genomes [Ser+22]. Long-read technologies are highly promising as they can further improve our understanding of genomic structure [Nur+22]. Yet, they also present new computational challenges due to their longer length and higher error rates.

The optimal sequence alignment between two sequences can be found in quadratic time and linear space [Hir75; MM88] if we use the classical Smith-Waterman or Needleman-Wunsch algorithm for local and global alignment,

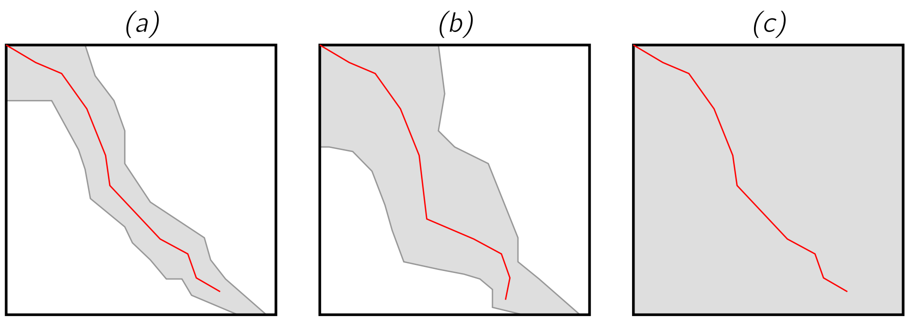

Figure IV.2: The red path is the optimal alignment, the gray area is calculated values, and the white area is non-calculated values. Due to the X-Drop condition, the white nonzeros contain a score of $-\infty$. Panel (a) shows an iteration with $X = 10$, (b) with $X = 20$, and (c) with $X = \infty$.

respectively. The sequence alignment problem is defined as follows. Given two sequences $\mathcal{H} = h_1, h_2, \cdots, h_m$, $\mathcal{V} = v_1, v_2, \cdots, v_n$, with $|\mathcal{H}| = m, |\mathcal{V}| = n$ we want to find the best scoring set of changes to transform sequence $\mathcal{H}$ into $\mathcal{V}$. If we assume that the sequences are homologous, i.e. that they are evolutionarily related, the number of resulting changes is small. The alignment is done by dynamic programming, where we define a dense scoring matrix $S(i, j)$, with $i \leq n, j \leq m$. The matrix S is filled from the upper left corner and extended to the lower right corner. In each nonzero, we store the best score for the alignment of each two-symbol pair (v_i, h_j). This score is computed based on the match of v_i and h_j and the history of alignment for the previous three scores, as defined in the following rule:

$$S(i,j) = \begin{cases} S(i-1, j-1) + Sim(v_i, h_j) & \text{if } i > 0, j > 0, \\ S(i, j-1) + \text{gap} & \text{if } j > 0 \\ S(i-1, j) + \text{gap} & \text{if } i > 0 \end{cases}$$

In the above definition, $Sim(v_i, h_j)$ is an arbitrary scoring function used to quantify the degree of similarity between a pair. In the case of DNA, $Sim(v_i, h_j)$ is a positive value if v_i and h_j match (i.e., no change is required), or a negative value if they do not, and gap is also a negative value, meaning that either v_i or h_j has a symbol inserted or deleted at that position: The goal is to find a path of changes in S that maximizes the score and is optimal for aligning $\mathcal{H}$ and $\mathcal{V}$. The scoring function assigns higher scores to likely biologically related sequences and lower scores to less likely related sequences.

In real-world scenarios, we can often make a reasonable assumption about where to find the optimal alignment on the two sequences, and this can lead to heuristics that can significantly reduce time and space complexity.

First, in one-to-many and many-to-many sequence alignment, the number of sequences to be compared can be reduced by first identifying common contiguous subsequences of fixed length k (i.e., k-mers). The k-mer information reduces the number of sequences to be compared in large-scale computation but also gives us an indication of where in the sequences the optimal alignment might be found. This can lead to a semi-global alignment approach, where each pairwise alignment is divided into the left and right extension of the k-mer match.

The semi-global approach forces the alignment to start at one of the two extremities of the sequences (similar to the Needleman-Wunsch algorithm [NW70]), but leaves the other extremity free. This is a common approach first introduced by BLAST [Alt+90], and it can lead to shorter sequences, but it scales with $\mathcal{O}(mn)$, where m and n are the lengths of the sequences involved since we still need to compute the entire matrix S.

The second key insight is that for sequences that have some similarity, the optimal alignment is often found on the diagonal of the S matrix, with the antidiagonal extremities storing low (i.e., bad) scores because the number of mismatches is high when moving away from the center of the diagonal, as shown in Figure IV.1. A common approach is to restrict the search to a predefined band region of S around the diagonal (left in Figure IV.1). This heuristic significantly reduces the time and space complexity but may restrict the search space too much and result in the failure to find the correct optimal alignment. This problem is particularly severe for sequences generated with long-read technologies, as these technologies are more prone to insertion and deletion of nucleotides (which can lead to a long gap sequence that moves the optimal alignment away from the diagonal) than to mismatches (which keep the optimal alignment mostly on the diagonal), which was the case with short-read technologies.

This provides the motivation for the X-Drop condition. The X-Drop strategy can be viewed as a dynamic band approach, where the search space is dynamically bounded by the score values rather than by a predefined band width. The X-Drop algorithm defines a threshold X that removes nonzeros (and the resulting possible path) from the S matrix that are worse than a path with the current best score. The assumption is that a path that is significantly worse (where significant is defined by X) than the current best score is unlikely to lead to the optimal alignment. Let us denote the current best score as T. The X-Drop condition states that if $S(i,j) < T - X$ in the current iteration of the dynamic program, then $S(i,j) = -\infty$.

The first iteration of the fill process for the X-Drop algorithm initiates in the upper left corner and initializes the matrix with $S(0,0) = 0$. To avoid starting the alignment with a gap or gap sequence at a location in the matrix other than the upper left corner (i.e., where both sequences start), we define the off- matrix access as $S(i,j) = -\infty, i < 0 \vee j < 0$. This is because we want to perform a semi-global alignment and force one side of the extremities (from the k-mer match heuristic) to align. If X is large, the computation approaches filling the entire dynamic programming matrix S, whereas when X is small, the non-zeros in S are pruned as we move away from the optimal result, and these non-zeros become $-\infty$. In Figure IV.2 we show the impact of different X values on the search space of the scoring matrix.

The X-Drop algorithm has been widely implemented and variants used for commonly used long-read alignment software such as minimap2 [SK18].

In this work, we implemented the original X-Drop algorithm as formulated by Zhang [ZBM98; Zha+00]. Their implementation traverses the dynamic programming matrix S in an antidiagonal fashion. They fill the non-zeros along the antidiagonal line from the upper right corner to the lower left corner of the dynamic programming matrix S. This sweeping anti-diagonal approach is initiated at $S(0, 0)$ and progresses until the lower right corner is reached. To fill a cell in an antidiagonal of the matrix, only the scores of the adjacent cells (top, top-left, and left) are needed. These are stored in the antidiagonals that were filled in the previous two phases. Earlier literature [ZBM98; Zha+00] used this insight to store only three antidiagonals: two for the previous phases and one for the current phase. This approach is popular for SIMD parallelism because the dependencies for input and output non-zeros are well aligned. But this is not strictly necessary. Gotoh [Got82] stated that storing two antidiagonals is sufficient. However, it is not often used in practice because SIMD parallelism is difficult to achieve. Despite that, in this work, we choose to store only two antidiagonals as the IPU is a MIMD architecture, and we aim to reduce the memory footprint.

IV.2.3 ELBA

ELBA is a long-read assembler implemented for distributed-memory parallelism that uses sparse matrices as the main data structures, mapping the *de novo* assembly process onto sparse matrix computation [Gui+22]. ELBA is composed of five main stages which comply with the Overlap-Layout-Consensus paradigm for assembling long-read sequencing data. In the first step, k-mer counting, the input sequences are parsed to extract subsequences of fixed length k and count their frequency. This produces a 1D distributed hash table of the k-mers and their frequencies and sequences of origin. This hash table is then transformed into a 2D $|k-mers|$-by-$|sequences|$ sparse matrix that we call $\mathbf{A}^\mathsf{T}$. In the second step, called overlap detection, ELBA multiplies $\mathbf{A}$ by its transpose $\mathbf{A}^\mathsf{T}$ to detect overlapping k-mer matches between input sequences. In this way, a $|sequences|$-by-$|sequences|$ matrix $\mathbf{C}$ is obtained in which the non-zeros represent such matches and their position on the sequences. Then, for each non-zero of $\mathbf{C}$, the X-Drop pairwise alignment algorithm is run, starting from the k-mer march position, to obtain similarity values and remove false matches from the matrix. In the fourth and fifth stages, ELBA simplifies the matrix, that is the assembly graph, to extract contiguous areas of the genomes (i.e., *contigs*), which are the result of the assembly process.

IV.2.4 PASTIS

PASTIS [Sel+20] similar to ELBA computes protein homology searches as a distributed sparse matrix multiplication. PASTIS computes the k-mer count and $\mathbf{A}\mathbf{A}^\mathsf{T}$, but then must perform additional matrix multiplication with the matrix S to produce the output $|sequences|$-by-$|sequences|$ matrix. The S matrix is called the substitution matrix and is used to find quasi-exact k-mer matches because

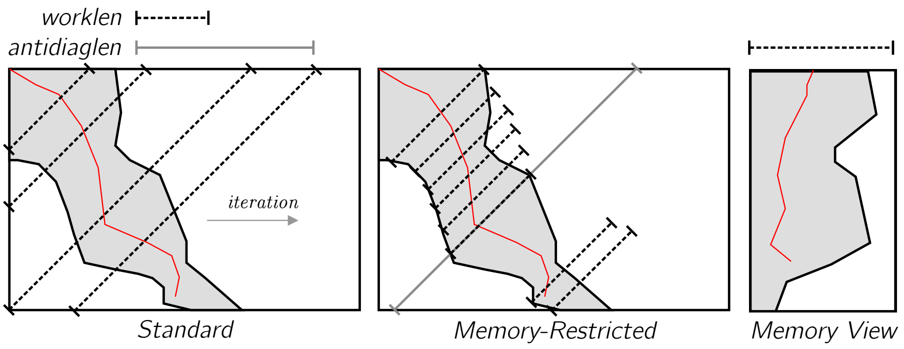

Figure IV.3: The antidiagonal length is $\delta = min(|\mathcal{H}|, |\mathcal{V}|)$. The memory-restricted version allocates work memory of $\max_k |U_k - L_k| \leq \delta_b \leq \delta$. The left panel illustrates the standard algorithm (3δ memory). The middle one illustrates our algorithm ($2\delta_b$ memory), while the right one is the pattern of our memory usage over time.

it has been shown that strictly enforcing exact matches in protein homology searches can lead to a significant loss of accuracy. Thus, the overlap detection phase has the form of $\mathbf{ASA}^\mathsf{T}$. Once the output matrix is formed, PASTIS computes an alignment step on each non-zero, similar to ELBA. PASTIS has two alignment modes: seed-and-extend with X-Drop and Smith-Waterman alignment. Using X-Drop, PASTIS initiates the alignment from the k-mer match.

Both PASTIS and ELBA defer implementation of X-Drop to the Library for Sequence Analysis (SeqAn) C++ library for CPU [Rei+17]. ELBA also provides support for the GPU-based X-Drop alignment called LOGAN [Zen+20]. LOGAN does not support protein alignment.

IV.3 Algorithm

In this section, we describe the algorithm we implemented on the Graphcore IPU and the algorithmic changes we made to make the computation more suitable for the IPU.

One of the major challenges in implementing sequence alignment on specialized hardware is the memory requirement since storing the entire dynamic matrix can exceed the available memory. In Section IV.2.2, we described how it is possible to reduce the memory footprint of the scoring matrix S by storing only three antidiagonal phases (the previous two phases and the current phase k) to traverse S, instead of storing the entire matrix. It can be observed that an antidiagonal can never become larger than $\delta = min(|\mathcal{H}|, |\mathcal{V}|)$, where $\mathcal{H}$ and $\mathcal{V}$ are the sequences involved in the alignment. To limit the workload, it is common to use a lower L_k and an upper bound U_k for the antidiagonal, where $|U_k - L_k|$ is the length of the antidiagonal in iteration k. These boundaries are derived from the number of scores in S that are not yet $-\infty$ (i.e., scores that have not triggered the X-Drop termination condition).

To store three antidiagonal phases, we need 3δ of memory for each alignment run. This memory requirement is too high for the IPU. Therefore, we address this problem with a two-step approach. First, we reformulate the algorithm using the technique found in [Got82] to store only two antidiagonal phases. This is possible by using a temporary variable since the values in the antidiagonal k and $k-2$ are one iteration offset accessed and written. In addition, we propose to use a band in the iteration, which is different from the classical banded algorithm shown in Figure IV.1 on the left, because the band is not static in space (i.e., it does not remain fixed around the diagonal), but is constantly realigned to the active iteration position that stores the best score. It is possible to observe that even though the antidiagonal is fully allocated (δ), only a small part of it is accessed during each phase k, since $|U_k - L_k| \leq \delta$. Therefore, in our implementation, we assume a bound length δ_b, which is the total working length $w = \max_k |U_k - L_k|$ of the antidiagonal to keep $w \leq \delta_b \leq \delta$. Thus, we use the restricted δ_b to constrain the algorithm in memory by placing antidiagonals in the active working area of the algorithm, resulting in a memory allocation of $2\delta_b$. Figure IV.3 on the left illustrates the antidiagonal length (black dashed line) for the original algorithm, while the middle one illustrates the antidiagonal length for our proposed memory-restricted version. The gray area is part of the scoring matrix S filled by the X-Drop algorithm. The right panel in Figure IV.3 illustrates a reinterpretation of the iteration space of the working memory region. The choice of an appropriate δ_b value is related to the error rate of the sequence and the X-Drop factor. Both high error rates and large X increase the working length w, as shown in Section IV.6.1.

Algorithm 3 describes our memory-restricted algorithm using only two antidiagonals A_1, A_2 of length δ_b. T is the best score found by the algorithm, while L, U are the lower and upper iteration boundaries, respectively. It is worth remembering that the algorithm for aligning X-Drop is semi-global; one side of the extremities of the two sequences is forced to align while the other side is left free. This is the case because each alignment results from splitting two sequences into four sequences (i.e., two for the left extension and two for the right extension) using the k-mer seed match information. To perform the forward alignment (right extension), we can access the sequences in a natural access pattern from left to right. For the backward alignment (left extension), we use an index transformation $op(\cdot)$ that produces either forward or backward accesses to $\mathcal{H}$ and $\mathcal{V}$. This way, we do not have to completely reverse the sequences to perform the alignment with the left extension. The current diagonal iteration is given by k. The algorithm terminates when L and U converge, i.e., when no values greater than $-\infty$ remain in the working set of the algorithm.

IV.4 Implementation

In this section, we describe the implementation of the memory-restricted X-Drop algorithm on the Graphcore IPU accelerator.

Algorithm 3 The memory-restricted X-Drop algorithm.

```
1: $L, U, T', T, k \leftarrow 0$
2: $L1_{inc}, L2_{inc} \leftarrow 0$
3: $A_1, A_2 \leftarrow \{-\infty, \ldots, -\infty\}$
4: $A_1[0] \leftarrow 0$
5: while $L \leq U + 1$ increase $k$ by 1 and do
6:     $W_2 \leftarrow A_2 + (-L + L2_{inc})$                        ▷ C-style array offsetting
7:     $W_1 \leftarrow A_1 + (-L + L2_{inc} + L1_{inc})$
8:     $W'_1 \leftarrow A_1 + (-L)$
9:     $w_{last} \leftarrow W_1[L-1]$                               ▷ Instead of a third anti-diagonal
10:    while $i \in (L, \ldots, U+1)$ do
11:        $j \leftarrow k - i - 1$
12:        $w_{new} \leftarrow W_1[i]$
13:        $score \leftarrow max \left\{ \begin{array}{r} W_2[i] - gap \\ W_2[i-1] - gap \\ w_{last} + sim(\mathcal{H}[op(i)], \mathcal{V}[op(j)])) \end{array} \right\}$
14:        $w_{last} \leftarrow w_{new}$
15:        if $score < T - X$ then
16:            $score \leftarrow - - \infty$
17:        end if
18:        $W'_1[i] \leftarrow score$
19:        $T' \leftarrow max\{T', score\}$
20:    end while
21:    $L_{prev} \leftarrow L$                                        ▷ zero t1 shifted values
22:    $L \leftarrow max(k + 1 - N, argmin(W'_1 \neq -\infty))$
23:    $U \leftarrow min(|\mathcal{H}| - 1, argmax(W'_1 \neq -\infty) + 1)$
24:    $L1_{inc} \leftarrow L - L_{prev}$
25:    $T \leftarrow T'$
26:    $swap(A_1, A_2)$
27:    $swap(L1_{inc}, L2_{inc})$
28: end while
```

Our implementation focuses on large sequences (both protein and DNA) whose length is in the range of 1K to 25K. This raises two challenges that we address in this work. First, the memory requirement for each alignment is large, given that a single pairwise alignment is executed on a single tile of the IPU, where that tile has 624 KB of addressable memory and six threads, each of which requires space for the algorithm. So we need to be able to allocate 6× the amount of working memory during the alignment on one tile. Second, due to the IPU's BSP (bulk synchronous parallel) architecture, we need to create a load-balanced problem (i.e., with equal runtime for each tile) to use all tiles equally. If a single tile takes more time, all other tiles must wait, resulting in poor utilization of hardware resources.

Table IV.1: Optimizations implemented and described throughout Section IV.4.4.

	Optimization	Time [ms]	GCUPS	To Prev.	Total
15% error	Single tile	493907	5.00		
	Scale to 1472 tiles	414	6034	1194×	1193.8×
	Use 6 threads	87	28705	4.76×	5679.4×
	LR splitting	85	29163	1.02×	5768.0×
	Work-stealing	85	29084	1.00×	5765.8×
	Dual issue	65	37933	1.30×	7504.9×
ELBA Ecoli	Single tile	4180499	14.52		
	Scale to 1472 tiles	6939	7302	602×	602.4×
	Use 6 threads	2707	14860	2.56×	1543.9×
	LR splitting	2470	16828	1.10×	1692.3×
	Work-stealing	1713	21935	1.44×	2440.4×
	Dual issue	1268	28587	1.35×	3296.8×

IV.4.1 Kernel Architecture

The X-Drop kernel was written as a Poplar vertex (codelet) in C++, where Poplar is the equivalent of CUDA for IPUs. Our implementation focuses on using the six hardware threads of each IPU tile. In the absence of synchronization instructions such as atomics or mutexes, we implemented a data-parallel implementation with throughput in mind. It is possible to perform tile-local coarse-grained synchronization by combining the hardware threads into a single supervised thread, which can use the entire set of six threads on a single alignment. However, this would result in context switches for each synchronized part of the algorithm, which would degrade performance. Therefore, we choose to have each thread perform a single performance alignment, and for this reason we have a sixfold memory footprint on each tile.

The IPUs inability for random external memory access and the limitation of a single tile to 624 KB of local memory forces us to choose a memory minimizing algorithm that enables many input sequences to be stored on a tile in order to maximize the number and size of sequences that can be processed on a single tile. To optimize the use of limited local memory, our tile architecture employs several techniques that allow all six threads to be used during the execution of the X-Drop algorithm. By reducing the amount of memory needed to implement the memory-restricted algorithm and efficiently using the available processing resources, we can achieve better performance and efficiency in sequence alignment on the IPU. The optimizations we implemented and their relative improvement are summarized in Table IV.1, measured for real-world and synthetic data.

IV.4.1.1 Tile Data Structures

The tile receives as input a set of sequences `seqs` and a list of seeds for these sequences to be computed. The seed matches are tuples containing a pointer to

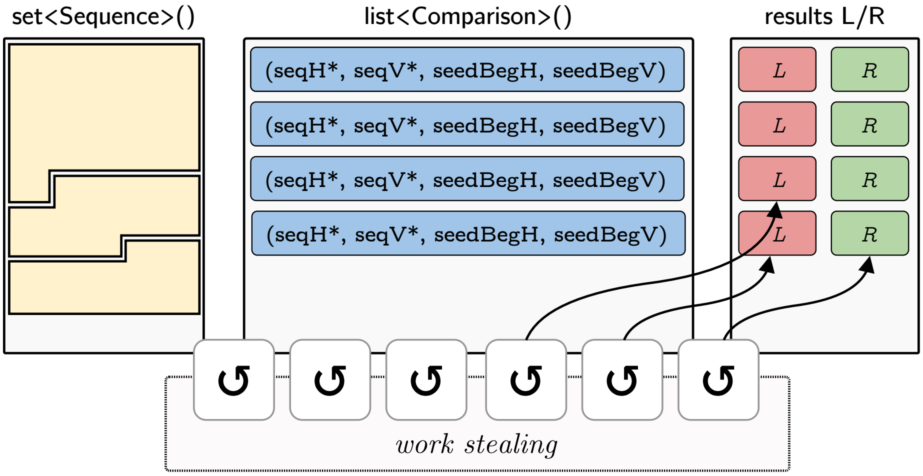
set<Sequence>()
list<Comparison>()
results L/R
(seqH*, seqV*, seedBegH, seedBegV)
(seqH*, seqV*, seedBegH, seedBegV)
(seqH*, seqV*, seedBegH, seedBegV)
(seqH*, seqV*, seedBegH, seedBegV)
L
R
work stealing

Figure IV.4: Tile structure with six worker threads filling in the output for left and right seed extension, using work stealing. The input is sequences and a list of seed extension information.

two sequences in `seqs`, and the position of the seeds on them to avoid having to split the sequences on the host, as shown in Figure IV.4. The output array stores a list of tuples for each left and right extension of a seed. Our representation has many advantages over state-of-the-art seed extension representation, as no preprocessing has to be done on the host device. Our algorithm can operate in reverse on continuous memory by providing a suitable *op* function for Algorithm 3. Furthermore, in real-world pipelines, a pair of sequences often must be aligned considering multiple seed matches, which would lead to the retransmission of the same sequences, negatively affecting performance. Thanks to the detached structure for sequences and seed alignment information we introduced, we can transfer multiple seed matches and sequences at once. This optimization saves $\mathcal{O}(\#seeds)$ in data transfer from the host to the device. The use of the *op* function is useful because the sequences can be truncated at different positions for the left and right extension, creating $\#matches \times 4\ \mathcal{H}_L, \mathcal{H}_R, \mathcal{V}_L, \mathcal{V}_R$ individual sequences.

IV.4.1.2 Left and Right (LR) Extension Splitting

Scaling from one thread to six threads per tile, we expect a speedup of 6×. However, the observed speedup is only 4.7×. This is because only 5 comparisons with 10 unique sequences have the memory to accommodate large sequences of length $10,000$ bp on a single tile, leaving one of six threads without work. To use all threads, we introduce a finer distribution of work by having threads work individually on the left and right extensions of the seed matches rather than assigning both extensions to the same thread. This doubles the number of work units to be distributed so that each thread is used for large sequences instead of

Table IV.2: Data sets for comparisons with CPU and GPU implementations with distribution for the left and right extensions.

Name	Cmp Count	Seqlen Avg	Seqlen P10 L	Seqlen Avg L	Seqlen P90 L
simulated85	40 000	9 992	9 992	9 992	9 992
ecoli	568 208	7 319	832	7 322	13 684
ecoli100	15 611 769	3 631	431	3 705	8 319
elegans	16 794 715	7 346	1 184	7 347	13 375
Name		Complexity Avg	Seqlen P10 R	Seqlen Avg R	Seqlen P90 R
simulated85		99 830 072	9 991	9 991	9 991
ecoli		45 870 449	823	7 317	13 675
ecoli100		12 524 999	388	3 557	8 087
elegans		52 763 834	1 179	7 345	13 380

leaving one thread idle. Our synthetic data, whose sequences are generated to be of equal length, does not benefit from this optimization because if we have 5 uniform workloads (i.e., sequence comparisons), they are split into 10 uniform workload units. This leaves four threads with two workload units, as is the case even without this optimization since we rely on BSP synchronization, which does not benefit from an unbalanced workload since the bottleneck is caused by the longest-running process. Nevertheless, this optimization can lead to a significant improvement in real-world workload due to a larger variance in seed position and sequence length.

IV.4.1.3 Eventual Work Stealing

Despite the LR optimization to increase workload granularity, we can still observe a large variance in thread runtime due to sequence length variance in real data. Using a simple round-robin workload allocation will leave one or more threads without work, resulting in load imbalance. Since synchronization is not possible on the IPU tiles, except for coarse thread joining, we initially resorted to statically assigning work to individual threads. However, since a single-seed extension has a relatively high runtime, we decided to implement *Eventual Work Stealing*. A work-stealing approach makes it possible for an idle thread to take a unit of work from the globally stored list of seed extensions and work on it locally. Since no mutexes were available to ensure that only one thread at a time could access the seed structure, we resorted to globally swapping a value. This does not avoid race conditions, but in this case, we would only compute a seed extension multiple times and not skip over it. Since instruction latencies are deterministic, two threads stealing the same unit of work will perpetually continue to do so. We introduced a small thread-unique busy wait loop to create variance, eventually avoiding this race condition and a possible perpetual joint execution. This loop reduces the race conditions from 16K to 18 for a total number of 1.13M alignments performed.

IV.4.1.4 Dual Instruction Issuing

The tiles implement a Very Long Instruction Word (VLIW) ISA with dual instruction issuance on two lock-synchronous pipelines, one integer and one floating point. The instructions require a single cycle to retire, except for certain floating-point operations such as `exp`, `log`, `sqrt`, which are not used in our code. Thus, we are not concerned with lock synchronicity. The integer pipeline is responsible for memory and branching operations, while the floating point pipeline is only responsible for floating point arithmetic.

We analyzed the generated assembly and found that registers in the integer pipeline in the inner loop of the X-Drop algorithm spilled when traversing the antidiagonal. Therefore, we reformulated our similarity function Sim (Section IV.2) to return floating-point values, forcing a floating-point representation of our scores to make use of additional floating-point registers. In addition, we used a compiler hint to use the built-in floating-point max instruction.

IV.4.2 Batching

Before the kernel is executed, the sequences (in pairs) must be distributed among the individual tiles. This distribution can be modeled as a k-partitioning problem, where each comparison task is assigned to a particular tile, and the total number of tiles is k. The size of each comparison is equal to the sum of the lengths of the two sequences involved. Given the BSP pattern of the IPU, it is important to minimize the longest-running tile runtime, which may cause other tiles to wait.

Estimating computational complexity is difficult because perfectly matched sequences have a smaller search space (they do not deviate too much from the diagonal) than sequences with higher mismatch rates. However, completely mismatching sequences run faster because the computation terminates early due to the X-Drop condition triggered by rapidly increasing bad scores. Therefore, we use the maximum running time, which is quadratic to the lengths of the sequences involved, for each comparison as an estimate of the computation time.

IV.4.3 Graph Based Sequence Partitioning

A single seed extension e_c with comparison index c for two sequences $\mathcal{H}, \mathcal{V} \in \Omega$, where Ω is the total set of input sequences Ω, contains the information of $e_c := (\mathcal{H}_i, \mathcal{V}_j, \mathcal{H}_s, \mathcal{V}_s)_c, \in \mathcal{C}$ for a seed of given length and two seed initial positions $\mathcal{H}_s, \mathcal{V}_s$ for $\mathcal{H}, \mathcal{V}$, respectively. In previous work, the relationship of pairs of sequences to each other was not considered (e.g., when two identical sequences have multiple k-mer matches), but these c-tuples were considered as single sequence extensions to be computed.

In this work, we propose to interpret the set of seed extensions as a graph partitioning problem to reduce the number of data transfers. The idea is to reduce the transmission of the same sequence from Ω, which is shared between multiple e_c used in the same batch sent to the IPU. As memory is not shared between tiles and communication has to be determined at compile time, we are limited to reusing sequences on a single tile. Dynamically compiling IPU exchanges at runtime takes too much time. Therefore, we can not create dynamic sequence exchanges, keeping Ω entirely on the IPU.

Many-to-many sequences reuse is enabled through our tile data structures storing the tile's local set of sequences $\omega_i \subset \Omega$ detached from the set of seed extensions in a tile with index i. The set of seed extensions only keeps a reference to sequences in ω_i.

Real-world bioinformatic pipelines that rely on many-to-many sequence comparisons store the information of which sequences need to be aligned against each other, which our optimization can use. Both ELBA and PASTIS offer this information in sparse matrix representation containing planned sequence alignments.

We propose a simple graph partitioning algorithm to increase data reuse. Given a graph $G(V, E)$, with $V \subseteq \Omega$, V the set of vertices, we want to distribute the set of edges $e_j \in E$, which is representative of the seed extensions $\mathcal{C}$. The graph has an edge between two vertices where a comparison uses the sequences represented in V. We distribute our graph in partitions p_i containing a set of edges and its set of sequences unique to the tile. The partitions are constrained to hold sequences of total size less or equal to available tile memory.

To avoid spending lots of time in this immediate optimization step, we partition the graph using a greedy strategy to stay within a thigh time regime, of usually less than a second for our tested data sets. The greedy strategy is given as follows: Take a vertex in the graph and linearly walk through the edge list. Add the start vertex to the partition and the adjacent vertex of the edge. Continue to walk through the edges, adding the adjacent vertex to the partition, until adding a new vertex would create exceed the partitions memory constraint; start a new partition. For simplicity, we leave the batching of our partitions to our batching algorithm.

Given equal-length sequences, this optimization approaches a sequence reuse effectiveness of $2\times$, as for each new comparison on a tile, only one new sequence needs to be transmitted, as the other is already ω_i. However, real-world sequence lengths are more inhomogeneous in size. Thus we observed for *E coli* and *C elegans*, that we could pack up to 41 smaller sequences into a single large sequence, drastically improving the transmission performance.

IV.4.4 Multi-IPU Support

The utilization of multiple IPUs is a crucial consideration in achieving optimal performance for many computational tasks. We are presented with multiple options for scaling up our algorithm. One such option is the combined multi-device approach, which exposes a virtual, seemingly homogenously extended IPU with a greater number of tiles. However, it should be noted that using this approach may result in global synchronization and necessitate larger batches, leading to suboptimal parallelization efficiency. Therefore, we have opted for a different approach that employs multiple single IPU devices.

Using our load-balancing driver, we can effectively manage load balancing and scheduling comparisons across connected single devices. It is worth noting that the individual devices remain hidden from the user. Such a methodology can help us achieve improved performance while ensuring optimal resource utilization.

Our wrapping driver class manages the Poplar graph and enables execution on multiple IPUs. The driver class wraps the submission of batches and handles internal work distribution across IPUs and their respective tiles. Batches are submitted to a work queue shared between all IPU instances. The shared queue is connected to the input stream of each IPU, which allows for prefetching by the IPU, as all submitted batches are fully preprocessed. Prefetching on the IPU allows for interleaving data transmission with computation allowing us to hide transmission time to an extent.

IV.5 Experimental Setup

Our test setups were run on three large systems. Firstly, we used the Perlmutter supercomputer equipped with a single-socket AMD EPYC 7763 CPU and 256 GB RAM. Furthermore, for running GPU experiments, each node has four NVIDIA A100 (Ampere) GPUs attached to each node. Secondly, the IPU results for the Mk2 IPUs were acquired on the ex3 supercomputer, which has a dual-socket Intel Xeon Platinum 8168 CPU connected to 16 Mk2 blades, containing four IPU GC200 each. Thirdly, our IPU BOW results were collected using a Paperspace cloud instance using a dual-socket AMD EPYC 7742 CPU and 425 GB RAM, connected to 16 BOW IPUS in two blades. All experiments were compiled with native optimizations, and AVX2 was explicitly enabled using GCC 11.2.0.

IV.5.1 Comparison to State-of-the-Art

We compared our implementation to CPU-based implementations SeqAn [Rah+18; Rei+17], ksw2 [Li18; SK18], libgaba [SK18] and genometools [GSK13]. All implementations were integrated into our benchmark runner program, which implements parallel processing of alignments using OpenMP [DM98]. Multiple data sets (Table IV.2) extracted from ELBA and a synthetic data set were tested with $X \in \{5, 10, 15, 20\}$.

For our performance measurements, we define Giga cell updates per second (GCUPS) as a metric for assessing the performance of aligner data sets on a given data set. Cells are defined as the number of fields in the dynamic programming matrix S, corresponding to the theoretical number of cells. The time of performing a full alignment by computing S is measured with the total time t. Heuristics, such as a X factor or banding, reduce the number of actually computed cells. We define our metric as: GCUPS $= \frac{|\mathcal{H}| \times |\mathbf{V}|}{t}$. Execution time was measured for our IPU implementation based on the cycle count necessary for the computation of the alignment on the device. The number of cycles for the execution of a given program is deterministic, given identical inputs and configuration parameters. Using the tile's frequency $f = 1.33 \times 10^9$ for the IPU

Mk2 and $f = 1.85 \times 10^9$ for the IPU Bow, the total on-device execution time can be derived by $t = {}^{\text{cycles}}/_{f}$. On the GPU, on-device execution performance was measured by timing kernel execution time without data transfer in LOGAN. On the CPU, the execution time for the alignments is measured without the preparation time necessary for loading sequences and comparison metadata.

IV.5.1.1 Strong Scaling

Using *E coli 100x* and *C elegans* data defined in Table IV.2, we investigated the strong scaling performance of our approach, scaling a single BOW IPU to 32 IPUs in exponential steps. Each scaling experiment was performed with graph-based partitioning (Section IV.4.3) enabled and disabled. Total execution time was measured after the alignments were generated, excluding sequence load times.

IV.5.2 Data Set

For the standalone experiments, we generated synthetic data and extracted realistic data from the ELBA pipeline to evaluate the performance of our own implementation as a function of certain properties of the data sets. All data sets, including distribution characteristics, used to compare the different X-Drop implementations are listed in Table IV.2.

The distribution of our data indicates a lower sequence length of 5 kb for *E coli 100x* in comparison to 15 kb for *E coli*, and *C elegans*. The seed position is equally distributed across sequences throughout all data sets but with a higher skew towards the center and edges in the *E coli* and *C elegans* data sets. For *E coli 100x* we observed that left and right extensions are skewed towards lower complexity alignments. The synthetic data sets were generated with equal sequence length and fixed read similarity. Mismatches were generated by uniform-randomly mutating individual bases outside the seed position. ELBA data sets were based on alignments generated during processing of PacBio SMRT HiFi read data from *E coli* (29x and 291x) and *C elegans* (40x) [Zen+20] in the alignment step of the pipeline with a seed length of 17 in all data sets.

IV.5.3 Real-World Application

For the two real pipelines PASTIS (Section IV.2.4) and ELBA (Section IV.2.3), we perform the IPU experiments on the IPU BOW system. Scaling to multiple IPUs was transparently enabled by setting our library's `NUMBER_IPUS` parameter. Therefore, no further code optimization was required in either pipeline.

Both pipelines create sparse *overlap* matrices to determine the sequences to be compared. We interpret these matrices as adjacency matrices for our graph partitioning scheme presented in Section IV.4.3.

To compare the system performance of our systems, we focus only on the *alignment* step since the other nodes are not equipped with equivalent hardware. The remaining and leading times of the pipelines are unaffected by the device on which the alignment step is performed. Any speedup that one method provides over the other contributes to the speedup of the entire pipeline in proportion to the percentage of time that the alignment step originally took.

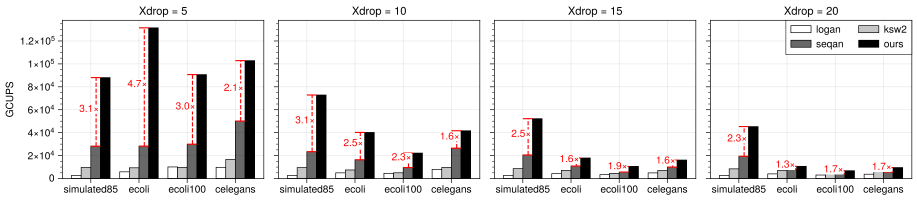

Figure IV.5: Normalized performance of our IPU implementation on 4 data sets (Table IV.2) in comparison to CPU implementations SeqAn, ksw2 and GPU implementation LOGAN. The relative speedup to the second fasted implementation SeqAn is given.

IV.5.3.1 PASTIS

Our integration is based on git-commit `fced0f2`, in which we replaced the SeqAn library X-Drop alignment algorithm with our own implementation of X-Drop. PASTIS does not provide a X-Drop GPU algorithm because, to our knowledge, no GPU X-Drop algorithm supports protein alignment. For PASTIS, the largest data set we could run was a uniformly subsampled protein database from the metaclust [SS18] data set, containing 500 k protein sequences. We used an X-Drop factor of $X = 49$ and a gap penalty of -2 and used BLOSUM62 [HH92] as our similarity matrix, as described by Selvitopi et al. [Sel+20]. Further, we choose a k-mer length of 6, with two required seeds per overlap.

IV.5.3.2 ELBA

The results are based on the GPU branch's git-commit *40c1b3a*. We used the same input data provided by Guidi et al. [Gui+22] to measure performance. The comparisons were made using the *E coli*, *C elegans* data sets. We compared the runtime of the alignment kernel in the ELBA bioinformatics pipeline. All experiments were run with X-Drop factors of $\{10, 15, 20\}$ and a k-mer length of 31, with two required seeds per overlap.

IV.6 Experimental Results

IV.6.1 Selection of δ_b

To evaluate the validity of our algorithm's assumption that δ_w is significantly smaller than δ, we conducted experiments on various synthetic data sets with decreasing similarity rates from 100% down to 0% (Figure IV.6) on sequence lengths of 20000 base pairs. We observed that the working band δ_w is smallest for perfectly matching sequences, except for sequences that are entirely mismatched

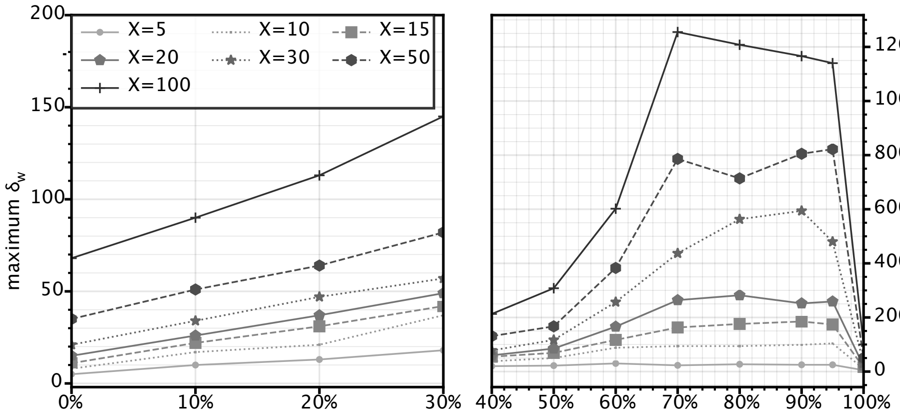

Figure IV.6: Find the maximum spread of the upper and lower pointers of the anti-diagonal δ_w for error rates ranging from 0% to 100% symbol mismatches with varying X-Drop factors.

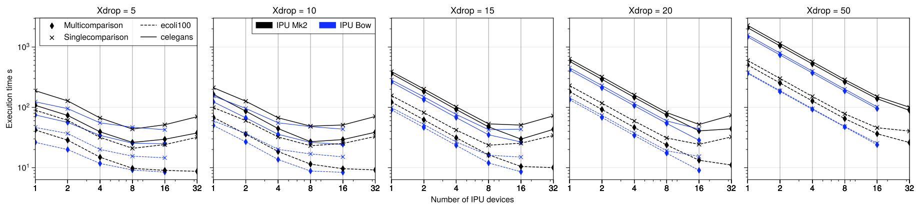

Figure IV.7: Scaling performance measured in alignment execution time on the *E coli 100x* and the *C elegans* data sets using 1 to 32 IPU devices. *Multicomparison* enables the usage of graph-based partitioning of comparisons, which allows for sequence reuse.

at 0% similarity. For perfectly matching sequences, the computed band is small. The highest score is always located on the diagonal, and the search is aborted close to the diagonal based on X.

When the similarity decreases to 80%, the working band doubles for small X and only increases 13% for $X = 100$. All investigated X-Drop values reach a maximum bandwidth when the sequence mismatch is around 70%. Further decreasing similarity, the bandwidth decreases again as the computation terminates early due to the X-Drop condition. For fully mismatched sequences of similarity 0%, the area of computed cells is restricted by X to a distance to the start of both sequences depending on mismatch and gap penalties.

For real-world *E coli* data δ_w values were $\{176, 339, 656\}$ for realistic values of X of $\{10, 15, 30\}$, respectively. Compared to the longest sequence length required for δ, we can choose a $\delta_b \geq \delta_w$, saving up to 98.2% of memory for a realistic X value of 15.

IV.6.2 Comparison to State-of-the-Art

On all data sets and X, the IPU implementation shows better performance, while the smallest difference was observed with *E coli* and $X = 20$. Single IPU on-device performance reaches 102 844 GCUPS using *C elegans* with $X = 5$, that is 2.05× faster compared to SeqAn's (50 084 GCUPS) and 10.54× faster than LOGAN (9761 GCUPS) on a single GPU. With a higher $X = 20$ the IPU implementation is 1.68× faster than SeqAn and 2.55× faster than LOGAN.

Of the CPU implementations, the SeqAn X-Drop implementation consistently outperforms ksw2 because ksw2 [Li18] penalizes long gaps less, leading to a larger search space. Further, tests on an Intel Xeon Platinum 8360Y showed consistently worse performance than the AMD EPYC 7763; thus, we omitted the results here. We observed that LOGAN does not perform well on HiFi data sets. We attribute this to the higher sequence similarity and more unbalanced search lengths under smaller values of X. For our implementation, we note that larger X values expand computational work on similar sequences while dissimilar sequences still terminate early. This leads to faster-decreasing performance on more dissimilar inputs. In comparison, LOGAN's SIMT implementation gains performance over other hardware through a larger search space.

Strong scaling from 1 up to 32 IPU devices for $X \in \{5, 10, 15, 20, 50\}$ on the *E coli 100x* and *C elegans* data sets showed near linear scaling behavior for up to 16 IPUs on larger $X \geq 15$ (Figure IV.7) with up to 15× speedup on 16 devices on the *C elegans* data set ($X = 50$). Aside from a constant factor speed-up provided by the higher clock frequency of the IPU Bow, scaling properties do not strongly differ between the tested IPU Bow and IPU Mk2 systems. If not otherwise specified, scaling results refer to the IPU Mk2 system. Graph partitioning of comparisons allows for the reuse of sequences for multiple comparisons. This decreases the amount of sequence data that needs to be transferred to the IPU and increases the number of comparisons that can be performed on a single IPU tile. On the *E coli 100x* data set, the number of batches is reduced by −52% (816 to 387) and on the *C elegans* data set by −44% (1723 to 972). On both data sets, this increases performance. Using $X = 10$, for *E coli 100x* 1.46× on a single device and up to 3.59× using 32 devices and on the *C elegans* data set 1.29× on a single device and 1.83× on 32 devices. For higher $X = 20$ and $X = 50$, scaling is linear up to 16 and 32 devices both using single- or multi-comparisons. For $X = 50$ using multi-comparisons is 1.18× faster on the *E coli 100x* data set using 1 device and 1.55× faster using 32 devices. This suggests a higher computational load per batch, allowing for the full utilization of more IPUs before the interconnect to the IPUs is saturated.

IV.6.3 Real World Pipelines

The CPU and GPU results were collected on AMD EPYC 7763 nodes, while the IPU results were collected on the IPU BOW system with an AMD EPYC 7742.

IV.6.3.1 ELBA

For *E coli*, we measured 7.4 s in the alignment phase with $X = 15$ using a single IPU. We observed good scaling up to 8 IPUs which took the alignment time down to 2.2 s. The CPU system took 11.61 seconds, using a single node, while the GPU code was run with up to 4 GPUs spending 52.14 s in the alignment phase.

We used the *C elegans* data set, the largest data set we could run, which occupied around 400 *Gb* of the host system's DRAM. As the EPYC 7763 nodes have less memory, we had to compare the IPU results against four nodes for CPU and GPU comparisons. For a four-node CPU setup, we attained $227.5, 340.7$ s for X of $\{15, 20\}$, respectively. The GPU results were measured with a total of 16 GPUs resulting in 1068 s for $X = 15$. Compared to four CPUs and 16 GPUs the IPUs alignment phase took 255.6 and 401.9 s for X of $\{15, 20\}$, respectively. On $X \geq 15$, we observed scaling up to 16 IPUs. The total alignment runtime was brought down to 46.5 s for $X = 15$, which is equivalent to a speedup of 22.3× to a cluster with 16 GPUs and a 4.7× speedup to a four-node CPU cluster.

IV.6.3.2 PASTIS

We measured 44.9 s for the alignment step on the CPU, while the IPU took 9.6 s, which is equivalent to a 4.7× speedup over the CPU. For larger inputs with more than 500 k sequences, experienced segmentation faults, which made larger scale experiments not possible.

IV.7 Related Work

A large number of works have focused on accelerating the Smith-Waterman and Needleman-Wunsch algorithm for sequence alignment [Awa+20; Fen+19; LWS13; Mül+22], which without additional heuristics computes the entire dynamic programming matrix for the alignment. Other work has focused on accelerating specific tools and applications, including BLAST [LB10; VS11; Ye+17] and BWA [Liu+12], both of which implement heuristic alignment strategies. The data intensive computations patterns in sequence alignment have lead to the development of memory-centric processor architectures including processing in memory (PIM) [Gup+19; Mut+22; Xu+23; ZZJ18] and near-memory computing [Sin+21] systems. Edit-distance algorithms consider only the number of changes necessary to transform one sequence into another, with each modification, either a insertion, deletion or substitution having the same cost. This considerably more constrained formulation of an alignment problem has similar complexity properties of $O(nm)$. The Bitap-Algorithm [Döm64] uses a bitmask and bitwise operations on a constrained alphabet to compute the edit distance between a pattern and a queried string. It has been adapted with a greedy windowing heuristic for long sequences and parallelized execution in GenASM and Scrooge [Cal+20; Lin+23], reducing the alignment complexity to $O(n + m)$, while allowing for non-optimal alignment results.

The X-Drop alignment algorithm has rarely been the target of hardware acceleration work. Recent efforts have included GPU [Zen+20] and FPGA [Zen+21], which outperformed then state-of-the-art CPU implementations.

IV.8 Conclusion

The processing power of modern CPUs has far outpaced the speed improvement of persistent and random access memory (DRAM), widening the gap between memory and processor performance. This discrepancy is masked by a highly hierarchical cache system in traditional CPUs. The Graphcore IPU's single level of large, low-latency SRAM reflects the recent trend toward shifting computation to memory, in both PIM and near-memory computing approaches.

In this work, we implement the X-Drop sequence alignment algorithm on the Graphcore IPU. A massively parallel MIMD AI accelerator with a single level of low latency SRAM for storage. Our contributions include algorithmic updates to the X-Drop algorithm to adapt it to the memory-constrained IPU architecture. Our dynamic band restriction algorithm reduces memory usage without compromising alignment computation with real data. Our formulation of graph-based sequence partitioning enables the reuse of sequences in many-to-many sequence alignment settings common in genome assembly and protein cluster pipelines.

Our implementation of X-Drop sequence alignment outperforms current state-of-the-art implementations on CPU and GPU for both DNA and protein alignment for realistic X values. Furthermore, we demonstrate near-linear strong scaling properties on common IPU host configurations. In two real-world pipelines, ELBA and PASTIS, we demonstrate significant speedup using our IPU implementation as the algorithm for the X-Drop aligner.

Finally, we note that the low bandwidth of host-device communication and the rigidity of the BSP paradigm, as well as the lack of atomic operators for thread-level cooperative multitasking, are the major limitations of the Graphcore IPU system. Our implementation mitigates these issues, but future SRAM-based architecture should be improved to enable the widespread use of SRAM-based computing for more data-intensive computation.

In summary, the IPU has significant potential for accelerating irregular computations where low-level parallelism is difficult to exploit on highly instruction-parallel architectures, such as GPUs.

References

[AG98] Apostolico, A. and Giancarlo, R. “Sequence Alignment in Molecular Biology”. In: *Journal of Computational Biology* vol. 5, no. 2 (Jan. 1998), pp. 173–196.

[Als+21] Alser, M. et al. “Technology Dictates Algorithms: Recent Developments in Read Alignment”. In: *Genome Biology* vol. 22, no. 1 (Aug. 2021), p. 249.

[Alt+90] Altschul, S. F. et al. “Basic Local Alignment Search Tool”. In: *Journal of Molecular Biology* vol. 215, no. 3 (Oct. 1990), pp. 403–410.

[Ama+20] Amarasinghe, S. L. et al. “Opportunities and Challenges in Long-Read Sequencing Data Analysis”. In: *Genome Biology* vol. 21, no. 1 (Feb. 2020), p. 30.

[Awa+20] Awan, M. G. et al. “ADEPT: A Domain Independent Sequence Alignment Strategy for Gpu Architectures”. In: *BMC Bioinformatics* vol. 21, no. 1 (Sept. 2020), p. 406.

[Bur+21] Burchard, L. et al. “iPUG: Accelerating Breadth-First Graph Traversals Using Manycore Graphcore IPUs”. In: *International Conference on High Performance Computing.* Springer. 2021, pp. 291–309.

[Bur+23] Burchard, L. et al. “Enabling Unstructured-Mesh Computation on Massively Tiled AI-Processors: An Example of Accelerating In-Silico Cardiac Simulation”. In: *Frontiers in Physics* vol. 11 (2023), p. 105.

[Cal+20] Cali, D. S. et al. *GenASM: A High-Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis.* Sept. 2020. arXiv: arXiv:2009.07692.

[DM98] Dagum, L. and Menon, R. “OpenMP: An Industry Standard API for Shared-Memory Programming”. In: *IEEE Computational Science and Engineering* vol. 5, no. 1 (Jan. 1998), pp. 46–55.

[Döm64] Dömölki, B. “An Algorithm for Syntactical Analysis”. In: *Computational Linguistics* vol. 3, no. 29-46 (1964), p. 151.

[Fen+19] Feng, Z. et al. “Accelerating Long Read Alignment on Three Processors”. In: *Proceedings of the 48th International Conference on Parallel Processing.* Kyoto Japan: ACM, Aug. 2019, pp. 1–10.

[Got82] Gotoh, O. “An Improved Algorithm for Matching Biological Sequences”. In: *Journal of Molecular Biology* vol. 162, no. 3 (Dec. 1982), pp. 705–708.

[GSK13] Gremme, G., Steinbiss, S., and Kurtz, S. “GenomeTools: A Comprehensive Software Library for Efficient Processing of Structured Genome Annotations”. In: *IEEE/ACM transactions on computational biology and bioinformatics* vol. 10, no. 3 (2013), pp. 645–656.

[Gui+20] Guidi, G. et al. *Parallel String Graph Construction and Transitive Reduction for De Novo Genome Assembly.* Oct. 2020. arXiv: arXiv:2010.10055.

[Gui+22] Guidi, G. et al. “Distributed-Memory Parallel Contig Generation for De Novo Long-Read Genome Assembly”. In: *Proceedings of the 51st International Conference on Parallel Processing.* 2022, pp. 1–11.

[Gup+19] Gupta, S. et al. “RAPID: A ReRAM Processing in-Memory Architecture for DNA Sequence Alignment”. In: *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. July 2019, pp. 1–6.

[HH92] Henikoff, S. and Henikoff, J. G. “Amino Acid Substitution Matrices from Protein Blocks”. In: *Proceedings of the National Academy of Sciences of the United States of America* vol. 89, no. 22 (Nov. 1992), pp. 10915–10919.

[Hir75] Hirschberg, D. S. “A Linear Space Algorithm for Computing Maximal Common Subsequences”. In: *Communications of the ACM* vol. 18, no. 6 (June 1975), pp. 341–343.

[Jia+19] Jia, Z. et al. “Dissecting the Graphcore IPU architecture via microbenchmarking”. In: *arXiv preprint arXiv:1912.03413* (2019).

[Lau21] Lauterbach, G. “The Path to Successful Wafer-Scale Integration: The Cerebras Story”. In: *IEEE Micro* vol. 41, no. 6 (Nov. 2021), pp. 52–57.

[LB10] Ling, C. and Benkrid, K. “Design and Implementation of a CUDA-compatible GPU-based Core for Gapped BLAST Algorithm”. In: *Procedia Computer Science*. ICCS 2010 vol. 1, no. 1 (May 2010), pp. 495–504.

[Lee+10] Lee, V. W. et al. “Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU”. In: *Proceedings of the 37th Annual International Symposium on Computer Architecture*. ISCA ’10. New York, NY, USA: Association for Computing Machinery, June 2010, pp. 451–460.

[Li18] Li, H. “Minimap2: Pairwise Alignment for Nucleotide Sequences”. In: *Bioinformatics* vol. 34, no. 18 (Sept. 2018), pp. 3094–3100.

[Lin+23] Lindegger, J. et al. “Scrooge: A Fast and Memory-Frugal Genomic Sequence Aligner for CPUs, GPUs, and ASICs”. In: *Bioinformatics* (Mar. 2023), btad151. arXiv: 2208.09985 `[cs, q-bio]`.

[Liu+12] Liu, C.-M. et al. “SOAP3: Ultra-Fast GPU-based Parallel Alignment Tool for Short Reads”. In: *Bioinformatics (Oxford, England)* vol. 28, no. 6 (Mar. 2012), pp. 878–879.

[LM21] Louw, T. and McIntosh-Smith, S. *Using the Graphcore IPU for traditional HPC applications*. Tech. rep. EasyChair, 2021.

[LWS13] Liu, Y., Wirawan, A., and Schmidt, B. “CUDASW++ 3.0: Accelerating Smith-Waterman Protein Database Search by Coupling CPU and GPU SIMD Instructions”. In: *BMC Bioinformatics* vol. 14, no. 1 (Apr. 2013), p. 117.

[Meu+01] Meuer, H. et al. *Top500 supercomputer sites*. 2001.

[Mit+21] Mittal, S. et al. “A Survey of SRAM-based in-Memory Computing Techniques and Applications”. In: *Journal of Systems Architecture* vol. 119 (Oct. 2021), p. 102276.

[MM88] Myers, E. W. and Miller, W. “Optimal Alignments in Linear Space”. In: *Bioinformatics* vol. 4, no. 1 (Mar. 1988), pp. 11–17.

[Mut+22] Mutlu, O. et al. *A Modern Primer on Processing in Memory.* Aug. 2022. arXiv: arXiv:2012.03112.

[Mül+22] Müller, A. et al. “AnySeq/GPU: A Novel Approach for Faster Sequence Alignment on GPUs”. In: *Proceedings of the 36th ACM International Conference on Supercomputing.* ICS ’22. New York, NY, USA: Association for Computing Machinery, June 2022, pp. 1–11.

[Nur+22] Nurk, S. et al. “The Complete Sequence of a Human Genome”. In: *Science* vol. 376, no. 6588 (Apr. 2022), pp. 44–53.

[NW70] Needleman, S. B. and Wunsch, C. D. “A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins”. In: *Journal of Molecular Biology* vol. 48, no. 3 (Mar. 1970), pp. 443–453.

[Rah+18] Rahn, R. et al. “Generic Accelerated Sequence Alignment in SeqAn Using Vectorization and Multi-Threading”. In: *Bioinformatics* vol. 34, no. 20 (Oct. 2018), pp. 3437–3445.

[Rei+17] Reinert, K. et al. “The SeqAn C++ Template Library for Efficient Sequence Analysis: A Resource for Programmers”. In: *Journal of Biotechnology.* Bioinformatics Solutions for Big Data Analysis in Life Sciences Presented by the German Network for Bioinformatics Infrastructure vol. 261 (Nov. 2017), pp. 157–168.

[Sel+20] Selvitopi, O. et al. “Distributed many-to-many protein sequence alignment using sparse matrices”. In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis.* IEEE. 2020, pp. 1–14.

[Ser+22] Sereika, M. et al. “Oxford Nanopore R10.4 Long-Read Sequencing Enables the Generation of near-Finished Bacterial Genomes from Pure Cultures and Metagenomes without Short-Read or Reference Polishing”. In: *Nature Methods* vol. 19, no. 7 (July 2022), pp. 823–826.

[Sin+21] Singh, G. et al. “FPGA-Based Near-Memory Acceleration of Modern Data-Intensive Applications”. In: *IEEE Micro* vol. 41, no. 4 (July 2021), pp. 39–48.

[SK18] Suzuki, H. and Kasahara, M. “Introducing Difference Recurrence Relations for Faster Semi-Global Alignment of Long Sequences”. In: *BMC Bioinformatics* vol. 19, no. 1 (Feb. 2018), p. 45.

[SS18] Steinegger, M. and Söding, J. “Clustering huge protein sequence sets in linear time”. In: *Nature communications* vol. 9, no. 1 (2018), p. 2542.

[Val90] Valiant, L. G. “A bridging model for parallel computation”. In: *Communications of the ACM* vol. 33, no. 8 (1990), pp. 103–111.

[VN14] Véstias, M. and Neto, H. “Trends of CPU, GPU and FPGA for High-Performance Computing”. In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. Sept. 2014, pp. 1–6.

[VS11] Vouzis, P. D. and Sahinidis, N. V. “GPU-BLAST: Using Graphics Processors to Accelerate Protein Sequence Alignment”. In: *Bioinformatics (Oxford, England)* vol. 27, no. 2 (Jan. 2011), pp. 182–188.

[Xu+23] Xu, W. et al. “RAPIDx: High-performance ReRAM Processing in-Memory Accelerator for Sequence Alignment”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2023), pp. 1–1. arXiv: 2211.05733 [cs].

[Ye+17] Ye, W. et al. “H-BLAST: A Fast Protein Sequence Alignment Toolkit on Heterogeneous Computers with GPUs”. In: *Bioinformatics* vol. 33, no. 8 (Apr. 2017), pp. 1130–1138.

[ZBM98] Zhang, Z., Berman, P., and Miller, W. “Alignments Without Low-Scoring Regions”. In: *Journal of Computational Biology* vol. 5, no. 2 (Jan. 1998), pp. 197–210.

[Zen+20] Zeni, A. et al. “LOGAN: High-Performance GPU-Based X-Drop Long-Read Alignment”. In: *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. New Orleans, LA, USA: IEEE, May 2020, pp. 462–471.

[Zen+21] Zeni, A. et al. “The Importance of Being X-Drop: High Performance Genome Alignment on Reconfigurable Hardware”. In: *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. May 2021, pp. 133–141.

[Zha+00] Zhang, Z. et al. “A Greedy Algorithm for Aligning DNA Sequences”. In: *Journal of Computational Biology* vol. 7, no. 1-2 (Feb. 2000), pp. 203–214.

[ZZJ18] Zokaee, F., Zarandi, H. R., and Jiang, L. “AligneR: A Process-in-Memory Architecture for Short Read Alignment in ReRAMs”. In: *IEEE Computer Architecture Letters* vol. 17, no. 2 (July 2018), pp. 237–240.

Authors’ addresses

Luk Burchard Simula Research Laboratory, Kristian Augusts gate 23, 0164 Oslo, Norway, luk@simula.no

Max Xiaohang Zhao Institut für Medizinische Genetik und Humangenetik, Charité, Augustenburger Pl. 1, 13353 Berlin, Germany, max.zhao@charite.de

Johannes Langguth Simula Research Laboratory, Kristian Augusts gate 23, 0164 Oslo, Norway; University of Bergen, Department of Computer Science, Postbox 7803, NO-5020 Bergen, Norway, langguth@simula.no

Aydın Buluç Lawrence Berkeley National Laboratory, 1 Cyclotron Rd, Berkeley, CA 94720, USA, abuluc@lbl.gov

Giulia Guidi Cornell University, 107 Hoy Rd, City of Ithaca, Tompkins, NY 14853, USA, gg434@cornell.edu