Accelerating Sparse Linear Solvers on Intelligence Processing Units

Tim Noack
Embedded Systems and Applications
TU Darmstadt
Darmstadt, Germany
noack@esa.tu-darmstadt.de

Louis Krüger
Fluid Mechanics and Aerodynamics
TU Darmstadt
Darmstadt, Germany
krueger.l@sla.tu-darmstadt.de

Andreas Koch
Embedded Systems and Applications
TU Darmstadt
Darmstadt, Germany
koch@esa.tu-darmstadt.de

***Abstract*—This paper presents an open-source framework for solving large, sparse linear systems on Intelligence Processing Units (IPUs). The IPU is a massively parallel architecture consisting of thousands of independent cores connected by an all-to-all communication fabric.**

We introduce two Domain-Specific Languages, CodeDSL and TensorDSL, that allow us to express complex algebraic algorithms like solvers and preconditioners close to their mathematical notation. Our framework symbolically executes these DSLs to generate the dataflow graph, execution schedule, and the codelets required by IPU's programming model.

The IPU's cacheless design allows us to reorder matrices without performance penalties. We present a novel matrix reordering strategy that enables efficient blockwise halo exchanges with the IPU's all-to-all communication fabric.

Our framework implements a suite of parallel linear solvers and preconditioners, including the Preconditioned Bi-Conjugate Gradient Stabilized (PBiCGStab) method and the Incomplete LU Factorization (ILU). The IPU's lack of native double-precision support poses a challenge for solving large, sparse systems. We address this using the novel combination of the Mixed-Precision Iterative Refinement (MPIR) method and double-word arithmetics to achieve high-precision solutions without native double-precision support.

We evaluate our framework's performance on matrices from the SuiteSparse collection and synthetic Poisson problems, demonstrating near-ideal strong and weak scaling on Sparse Matrix-Vector Multiplications (SpMVs). Comparative benchmarks show that our framework outperforms state-of-the-art CPU and GPU implementations by up to 150x and 36x, respectively, for SpMVs and iterative solvers at a comparable energy consumption level. Our results highlight the potential of IPUs for high-performance sparse linear algebra computations in scientific and engineering applications.

***Index Terms*—Sparse Linear Solvers, Domain Specific Language (DSL), Intelligence Processing Unit (IPU), Double-Word Arithmetics**

I. Introduction and motivation

Solving large, sparse linear systems of equations is a fundamental task in scientific computing, with applications ranging from computational fluid dynamics to structural analysis. The increasing complexity and scale of these systems demand evermore efficient computational methods and hardware architectures. While traditional CPU and GPU architectures have long been the workhorses for such computations, emerging specialized hardware presents new opportunities and challenges.

The Intelligence Processing Unit (IPU), developed by GraphCore, is a novel massively parallel architecture designed for machine learning applications. With its unique design featuring thousands of independent cores, high-bandwidth on-chip memory, and an all-to-all communication fabric, the IPU offers intriguing potential for accelerating sparse linear algebra computations.

However, we found that the IPU's programming model, centered around constructing a dataflow graph and an execution schedule, is not naturally suited to expressing complex algebraic algorithms. Additionally, the IPU lacks native support for double-precision floating-point operations, which are often crucial for achieving the required accuracy in scientific computations. Efficiently distributing and solving sparse linear systems across the IPU's thousands of relatively small cores while managing inter-core communication presents significant algorithmic challenges.

This paper presents an open-source framework [1] designed to address these challenges and enable efficient solution of large, sparse linear systems on IPU architectures. Our key contributions include:

1) Two Domain-Specific Languages (DSLs) that allow programmers to express complex algebraic algorithms close to their mathematical notation
2) A novel combination of the Mixed-Precision Iterative Refinement (MPIR) method with double-word arithmetics to achieve high-precision solutions without native double-precision support
3) A novel matrix reordering strategy that allows for efficient blockwise halo exchanges on IPUs
4) A suite of parallel linear solvers and preconditioners, including Preconditioned Bi-Conjugate Gradient Stabilized (PBiCGStab) method and the Incomplete LU Factorization (ILU)
5) A comprehensive evaluation demonstrating our framework's performance across various matrices, showcasing significant speedups over CPU and GPU implementations for certain operations.

By addressing these challenges, our framework opens up new possibilities for accelerating sparse linear algebra computations on specialized hardware. This work demonstrates the

potential of IPUs for scientific computing beyond their original machine learning focus.

II. Background

A. Intelligence Processing Unit

The Intelligence Processing Unit (IPU) is a novel massively parallel computing architecture developed by GraphCore specifically for machine learning applications. As the IPU is less familiar than traditional architectures like CPUs and GPUs, we provide a brief overview of the architecture and its programming model [2].

The current generation of IPU chips (*Mk2*) consists of 1,472 processor cores, referred to as tiles. For optimal utilization, each tile must execute six worker threads in parallel. In contrast to GPU architectures, worker threads on the IPU are fully independent and can execute different instructions.

Each tile is equipped with approximately 612 kB of local SRAM memory, totaling 900 MB per chip. This memory is tightly coupled to its respective processor core, enabling high-bandwidth, low-latency access without a cache hierarchy. The aggregate memory bandwidth reaches an impressive 47.5 TB/s per chip. However, each SRAM block is exclusively accessible by its associated processor core.

To program the IPU, GraphCore provides the Poplar software development framework, which includes the Poplar compiler. Programmers describe a program in the Poplar framework by constructing a dataflow graph, an execution schedule, and codelets. Codelets are individual computational operations, similar to CUDA kernels, that are programmed in C++. The dataflow graph consists of vertices representing compute operations on tensors, connected by edges depicting data flow. Parallel-executable vertices are grouped into compute sets. The execution schedule is a Directed Acyclic Graph (DAG), with vertices representing program steps. Program steps can execute a compute set, copy tensors, and perform control flow operations like loops and branches.

Multiple IPU chips can be interconnected via stateful, packaged inter-chip communication links (*IPU-Links*). Tiles on the same chip are connected to each other via a stateless, packageless, all-to-all communication fabric. The communication programs are generated before execution by the Poplar compiler. On-chip communication programs are cycle-precise; that is, each transfer's start and end cycle count relative to the communication program is fixed. This is possible because the Poplar compiler knows all the timing characteristics of the communication fabric, and all involved tiles synchronize before communication, following the Bulk Synchronous Parallel (BSP) paradigm. In the BSP model, computation is divided into supersteps separated by global synchronization barriers, ensuring predictable communication patterns and simplifying reasoning about parallel execution.

Notably, the IPU does not support double-precision floating-point operations. We overcome this limitation by using a combination of the Mixed-Precision Iterative Refinement (MPIR) method (Section V-B) and double-word arithmetics (Section III-D).

B. Sparse Linear Solvers on Parallel Architectures

Solving large, sparse linear systems of equations is a fundamental task in scientific computing, particularly in fields such as computational fluid dynamics, structural mechanics, and electromagnetic simulations. These systems often arise from discretizations of partial differential equations using Finite Element or Finite Volume Methods. Due to the scale and complexity of these problems, parallel computing architectures are often essential.

The parallelization of sparse linear solvers typically involves two levels of parallelism:

1) Distributed Memory Parallelism: On distributed memory systems - like clusters of CPUs and GPUs - the matrix describing the system of equations is distributed across multiple computation nodes. This distribution is typically done row-wise, where each node handles a subset of the matrix rows. During the solution process, nodes must exchange information about overlapping regions of their submatrices. This inter-node communication is commonly implemented using the Message Passing Interface (MPI) [3], [4].

2) Shared Memory Parallelism: Within each computational node, fine-grained parallelism can be exploited to utilize multi-core CPUs or many-core accelerators like GPUs. For example, NVIDIA's cuSPARSE library employs various techniques to parallelize sparse matrix operations across thousands of GPU threads, such as graph coloring, level-set scheduling, and iterative methods [5].

Our framework combines both techniques: We distribute the matrix row-wise across all tiles and use level-set scheduling to parallelize sequential solvers to all six worker threads of each tile.

C. Sparse Matrix Format

Various storage formats have been proposed to represent sparse matrices efficiently. Due to its simplicity and efficiency, the Compressed Row Storage (CRS) format (also called Compressed Sparse Row (CSR) format) is widely adopted for sparse matrix representation. This format employs three arrays to store the matrix [6]:

1) A value array contains all non-zero elements.
2) A column index array indicates the column position of each non-zero element.
3) A row pointer array marks each row's start in the value and column index arrays.

Our framework uses a modified CRS format that stores diagonal elements separately from the off-diagonal entries in a dense array. Although non-singular matrices do not inherently guarantee non-zero diagonal entries, in many practical applications – such as matrices arising from Finite-Element and Finite-Volume methods – or after applying an appropriate row permutation, the diagonal entries are typically non-zero. By storing these diagonal values in a dense array rather than embedding them in the CRS structure, we avoid recording their column indices, thereby reducing the memory footprint. Moreover, this modification benefits certain solvers, such as

the Gauss-Seidel method, by providing direct access to each row's diagonal element.

Several sparse matrix formats have been developed to enhance the vectorizability of operations, particularly for SpMVs commonly encountered in iterative solvers. Notable among these are the ELLPACK format, initially proposed in the ITPACKV software package [7], and its variant, the Sliced ELLPACK (SELL) format [8].

Both ELLPACK and SELL store matrix coefficients in fixed-length blocks. While these formats can significantly benefit processors with large vector units and memory caches, they add complexity to the implementation. However, the IPU has limited support for wide vector operations, with most single-precision floating-point instructions restricted to two-element wide vectors [2]. Furthermore, some operations, such as division and reciprocal calculations, do not support vectorization at all. The IPU's cache-less design eliminates most potential benefits from the contiguous memory layout of ELLPACK and SELL formats.

Furthermore, the IPU can execute conditional branches with single-cycle latency, allowing for an efficient selection of two-element vector and scalar instructions based on the number of coefficients left in the matrix row. Given these architectural considerations, we anticipate that the performance gains typically associated with ELLPACK and SELL formats would be small on IPUs. We leave the exploration of these formats on IPUs for future work.

III. Domain-Specific Languages (DSLs) for Algebraic Programs on the IPU

The programming model of Poplar requires the programmer to construct a data-flow graph, an execution schedule and C++ codelets. We found this to be cumbersome for developing complex algebraic programs like solvers. To address this, we developed two DSLs that work hand-in-hand: *CodeDSL* and *TensorDSL*. Both languages are embedded in C++ and dynamically typed. They allow the programmer to write expressive code that is close to the mathematical notation.

CodeDSL is a description language for codelets. Algorithms in CodeDSL are written from a tile-centric perspective, meaning they can only access parts of tensors that are mapped to the executing tile. CodeDSL supports most operations that are possible in bare C++.

TensorDSL operates on tensors mapped across one or multiple tiles, providing a global perspective on entire tensors, regardless of their distribution across multiple tiles or IPUs. This language supports elementwise operations, reductions, broadcasting, and copying tensors to and from remote and host memory, but it does not allow manipulating individual elements of a tensor. TensorDSL uses CodeDSL internally to generate codelets for tensor operations.

Many algorithms in our framework combine both languages. For instance, the Gauss-Seidel solver uses TensorDSL to calculate the residual and its vector norm, and CodeDSL to perform the smoothing step.

```
// Create a TensorDSL tensor
Tensor x(Type::FLOAT32, {10000});

// Fill the tensor with the Leibniz
// sequence using CoreDSL
Execute({x}, [](Value x) {
 // Iterate over the elements
 For(0, x.size(), 1, [&](Value i) {
  x[i] =
   ((i % 2 == 0) ? 1 : -1) / (2 * i + 1);
 });
});

// Calculate pi from the Leibniz sequence
// using TensorDSL
Tensor pi = x.reduce() * 4;

If(Abs(pi - 3.141f) < 0.001f,
 [] { Print("We found pi!") });
```

Fig. 1: Example of an algorithm written with TensorDSL.

Both DSLs support control flow operations, including conditionals and loops, with intuitive syntax. Although the syntax is consistent across both languages, the internal implementation of control flow operations differs fundamentally. While CodeDSL simply emits C control flow statements (eg. `if()`, `while()`) into the generated codelets, TensorDSL uses a *control flow stack* to generate the execution schedule for the dataflow graph (detailed in Section III-B).

Figure 1 shows an example of an algorithm that uses both languages. The example code uses CodeDSL to construct the Leibniz sequence and TensorDSL to calculate the sum of the sequence and multiply it by four. The result is a scalar tensor with a value close to the mathematical constant π.

A. Compilation and Execution

The compilation of our DSLs to a graph program and its execution consists of four steps:

1) **Compilation for the CPU**: The DSLs are compiled to a CPU executable using a standard C++20 compiler.
2) **Symbolic execution on the CPU**: The program is executed on the CPU, generating a dataflow graph, execution schedule, and codelets. We call this step *symbolic execution* because, during this step, tensors have symbolic values that correspond to edges in the generated data flow graph. TensorDSL operations like materializing expressions or copying tensors extend the dataflow graph and execution schedule. CodeDSL operations generate C++ codelets that are compiled and scheduled for execution.
3) **Graph compilation for the IPU**: The Poplar compiler optimizes the dataflow graph and execution schedule. It then generates communication schedules and machine code for the IPU.

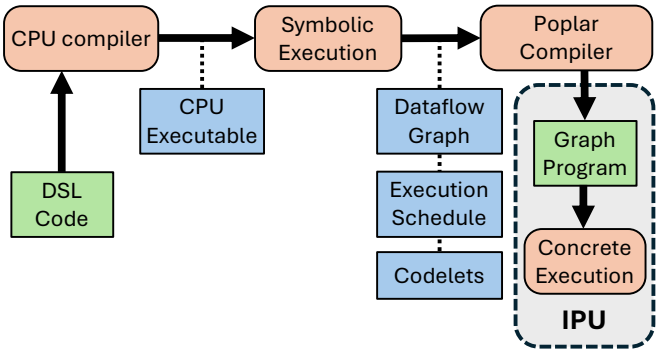


Fig. 2: Compilation and Execution of DSL code with our framework.

4) **Concrete execution on the IPU**: The generated graph program is executed on the IPU or in a simulator. CPU callbacks can be used for data transfer or to mix CPU and IPU calculations. For example, we use CPU callbacks to inform the user about the solver's progress. The simulator is part of the Poplar framework and allows the programmer to test their algorithms on the CPU before deploying it to the IPU.

The translation process is illustrated in Figure 2.

B. Control Flow in TensorDSL

Poplar's programming model requires developers to manually construct an execution schedule for the dataflow graph. The execution schedule consists of sequential program steps for executing codelets, copying tensors, performing control flow operations, and other tasks.

Our framework *automatically* generates the execution schedule based on a *control-flow stack*. The stack is managed by control functions provided in TensorDSL, such as `If` and `While`. For each branch, these functions push a new program step onto the stack and populate it by symbolically executing the code provided for the branch. The code is passed to the control functions as lambda functions, allowing for a clean and expressive syntax. After a branch is executed, the program step is popped from the stack again. With this approach, the program step at the top of the stack always represents the current state of the symbolically executed program.

C. Expressions and Materialization in TensorDSL

When evaluating a TensorDSL expression like `x * 4` during symbolic execution, our framework does not modify the Poplar dataflow graph or execution schedule directly. Instead, the mathematical operation returns an *expression object*. Expression objects can be combined to form more complex expressions. Only when the value of an expression is actually needed, the expression is *materialized*. Expressions are materialized by generating CodeDSL codelets from the expression tree for all involved tiles. The codelets are then compiled and their execution is scheduled in the current program step. When performing algebraic operations on tensors of different shapes, tensors are automatically broadcast according to NumPy's broadcasting rules. Broadcasting is done internally by the generated codelets, which avoids expanding broadcasted tensors in memory prior to the operation.

We found that delaying the materialization of expressions as long as possible has two advantages: First, it allows the C++ compiler to optimize the generated codelet, for example by performing a partial-redundancy elimination. This is not possible otherwise, because Poplar compiles each codelet to machine code in isolation. Second, it reduces the number of vertices and program steps in the dataflow graph and execution schedule. We found this to be important to reduce the compile time of the Poplar graph compiler.

D. Extended Precision

The IPU does not support double-precision floating point instructions natively. This presents a challenge for solving large sparse systems of equations, where single-precision is often insufficient to achieve acceptable residuals. To address this, our DSLs supports two additional data types: A double-precision type, which is emulated in software, and a double-word type, which uses double-word arithmetics to provide extended precision. Both types can be used with the Mixed-Precision Iterative Refinement (MPIR) method, detailed in Section V-B, to improve the precision of single-precision solvers at no significant cost, as we demonstrate in Section VI-C.

In double-word arithmetics, a number is represented as the sum of two floating-point values. The larger value can be seen as the rounded number, and the smaller value can be seen as the rounding error. For instance, 1.00000001 cannot be represented as a single-precision floating point number, but it can be written as the sum of $1 \cdot 10^{0}$ and $1 \cdot 10^{-8}$. The sum of these two numbers can represent a number with twice the precision of the main part alone. Double-word arithmetics take advantage of the fact that many floating-point operations can determine their own rounding error. It is worth noting that double-word arithmetics do not extend the range of representable numbers, only their precision.

The double-word arithmetics proposed by JOLDES ET AL. [9] and LANGE AND RUMP [10] provide algorithms for basic arithmetic operations (addition, subtraction, multiplication, and division) on double-word numbers and between double-word and single-word numbers. To the best of our knowledge, there was no existing C/C++ library for double-word arithmetics with single-precision floating-point numbers. Therefore, we implemented and open-sourced the TWOFLOAT library [11]. The library implements both arithmetics and calculates all required constants during compilation, enabling its use with any underlying floating-point type. CodeDSL uses TWOFLOAT internally to materialize double-word expressions.

While LANGE AND RUMP's approach offers faster computation by omitting normalization steps, we prioritize the more precise, albeit slower, arithmetic by JOLDES ET AL. for the MPIR method, as we found numerical stability to be crucial for overall solver performance. Depending on the operation,

Operation	Single-Precision	Double-Word	Double-Precision
Algorithm	*native*	JOLDES ET AL.	compiler-rt
Decimal digits	7.2	13.3 to 14.0	16.0
Range	10^{-38} to 10^{38}	10^{-38} to 10^{38}	10^{-308} to 10^{308}
Addition	6 cycles	132 cycles	ca. 1080 cycles
Multiplication	6 cycles	162 cycles	ca. 1260 cycles
Division	6 cycles	240 cycles	ca. 2520 cycles

TABLE I: Comparison of the floating-point types supported by our DSL and the cycle count required for basic arithmetic operations on the IPU. The exact precision of the double-word arithmetic depends on the operation; the exact cycle count for the emulated double-precision depends on whether normalization of the result is required.

the arithmetic by JOLDES ET AL. requires 20 to 34 floating-point operations to perform a double-word operation, while the arithmetic by LANGE AND RUMP requires only 7 to 25 operations. However, the precision decreases with consecutive operations, which is a concern for the Iterative Refinement (IR) method.

Table I compares the floating-point types supported by our DSL. Operations in the double-word arithmetic are significantly faster than emulated double-precision, at the cost of two to three fewer decimal digits of precision and a smaller range.

IV. HALO EXCHANGE

Our framework decomposes and distributes the matrix across multiple tiles. However, this decomposition introduces a challenge: Computations on one tile require data owned by neighboring tiles at the boundaries of each tile's domain.

For this, each tile maintains *halo values* - a buffer containing copies of boundary data from neighboring tiles. These values must be updated regularly during the solution process to ensure all tiles have access to the most current data required for their local computations.

While conventional CPU and GPU systems benefit from matrix row reordering to improve cache locality, the cacheless architecture of IPUs renders such optimizations irrelevant. Instead, our framework reorders the matrix to facilitate efficient blockwise transfers of halo regions, speeding up halo exchanges and reducing the size of the compiler-generated communication programs.

Our approach conceptualizes the matrix as a mesh of cells, with each cell representing a matrix row and coefficients describing neighboring relationships. The matrix partitioning corresponds to subdividing this mesh into subdomains assigned to different IPU tiles. Figure 3(a) illustrates this concept with an 8x8 mesh partitioned across four tiles.

We categorize cells into three types:

- Interior cells (blue): Owned and required only by the current tile.
- Separator cells (yellow): Owned by the current tile but also required by neighboring tiles.
- Halo cells (red): Owned by neighboring tiles but also required by the current tile.

Separator and halo cells are grouped into regions, where each separator region has one or multiple corresponding halo regions. During halo exchange, halo regions are updated with the values of their corresponding separator regions.

Our strategy establishes a consistent ordering of cells within each separator region and its corresponding halo regions. This means that the cells are ordered identically in the source (separator) region and all of its corresponding destination (halo) regions on neighboring tiles. By maintaining this consistent ordering, we enable data to be copied directly from a separator region's memory to all corresponding halo region memories in a broadcast manner, eliminating the necessity for reordering.

We define a region as the largest possible group of cells for which a consistent cell ordering can be established across all involved tiles. To determine the regions, each region is characterized by a unique set of *involved tiles* - for separator regions, these are the neighboring tiles requiring the region's values; for halo regions, it's determined by the corresponding separator region's neighbors.

Our strategy follows these steps:

1) Identify separator cells and the neighboring tiles requiring their values.
2) Group separator cells with identical sets of neighboring tiles into regions.
3) Create corresponding halo regions based on the separator regions.
4) Establish the same cell order within each separator region and all its corresponding halo regions.

Figure 3(a) visualizes the resulting regions, with purple arrows indicating regions sent from Tile 1 to its neighbors and green arrows showing regions received by Tile 1. Figure 3(b) displays the resulting memory layout of a solution vector (x) on Tile 1, with regions labeled by their involved tiles.

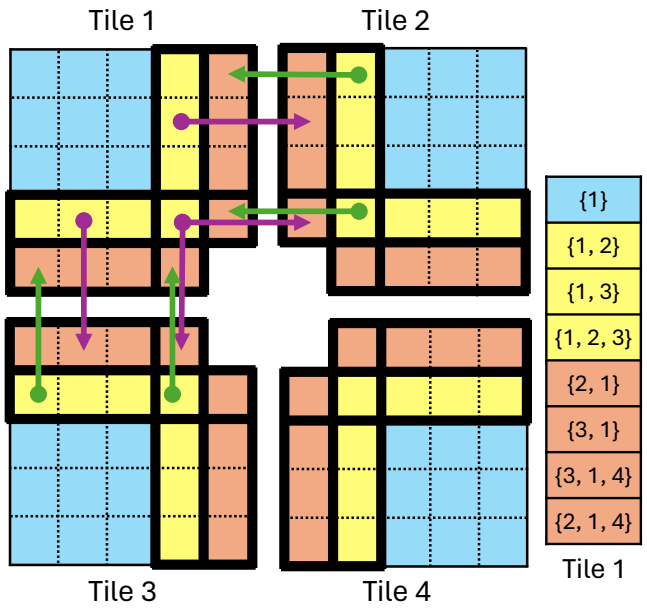
This reordering strategy enables efficient, blockwise halo exchanges without local reordering operations. This is beneficial for two reasons:

- It minimizes the size of communication programs generated by the compiler. Instead of issuing a communication instruction for each separator cell, the compiler can issue a single instruction for each region.
- It utilizes the IPU's unique communication fabric, which connects all tiles in an all-to-all manner. If multiple neighboring tiles require a region, it can be broadcast to all neighbors in a single blockwise transfer.

It is worth mentioning that our method is inspired by BURCHARD ET AL. [12]. The authors propose three communication schemes for unstructured mesh simulations on IPUs, but all of which communicate unused data and require reordering.

V. SOLVERS

Our framework offers a versatile suite of solvers for linear systems of equations. A key feature is the modular design, which allows for nested solver configurations - any solver can serve as a preconditioner for another, enabling highly customizable solution strategies.



(a) Mesh partitioning and region transfers

(b) Memory layout

Fig. 3: (a) An 8x8 mesh partitioned across four tiles, showing interior (blue), separator (yellow), and halo (red) cells. Purple arrows indicate regions sent from Tile 1, while green arrows show regions received by Tile 1. (b) Resulting memory layout of the solution vector on Tile 1, with regions labeled by involved tiles.

The solver hierarchy and associated parameters are easily configured through a JSON file. This approach allows users to quickly adapt the solver setup to their specific problem requirements without modifying the underlying code.

A. Level-Set Scheduling

Level-Set Scheduling is a parallelization technique for sequential solvers, initially proposed by ANDERSON AND SAAD [13] and later by SALTZ [14]. We use this method to parallelize the Gauss-Seidel and Incomplete LU (ILU) solvers to each tile's six worker threads, but it can be applied to any algorithm that iterates over matrix rows sequentially and then updates the solution vector for each row based on previously updated values.

The core idea of Level-Set Scheduling is to analyze the data dependencies in the lower triangular part of the matrix, which represent dependencies on updated solution vector values. The method generates a DAG of these dependencies, in which nodes represent matrix rows and edges indicate dependencies between rows. This graph is then clustered into the levels. Each level contains rows that depend on the updated values from previous levels, allowing for parallel processing of rows within the same level. The levels can be determined using graph search algorithms such as modified topological sort or breadth-first search [15]–[17]. When the levels are processed in order, the method yields the same result as the sequential algorithm and thus the same convergence rate.

Implementing Level-Set Scheduling requires all worker threads to process the same level concurrently, followed by a synchronization step to ensure updated values are available for the subsequent level. Initially, we achieved this synchronization by adding one compute set per level to Poplar's data-flow graph. Poplar automatically inserts a synchronization point before each compute set, ensuring all worker threads have completed the previous level before starting the next.

However, the large number of compute sets added to the dataflow graph led to unacceptably long graph compilation times. To address this issue, we developed and open-sourced the IPUTHREADING library [18]. Our library allows us to use a single compute set that spawns and synchronizes worker threads for every level. The library generates the startup code for worker threads and makes use of the `run`, `runall`, and `sync` instructions.

The amount of parallelism that Level-Set Scheduling can exploit depends on the sparsity pattern of the matrix. We found that the method can often fully utilize all six worker threads per tile. This is opposed to GPUs, where the method may struggle to fully utilize all of the thousands of threads available.

B. Mixed-Precision Iterative Refinement

Mixed-Precision Iterative Refinement (MPIR) is a method proposed by MOLER [19] for improving the accuracy and stability of floating-point solvers [20]. In the 2000s, hardware emerged in which single-precision floating point was much faster than double-precision floating point [21]. This motived first LANGOU ET AL. to use the MPIR method to enhance the performance of dense linear algebra calculations [22], and later BUTTARI ET AL. for sparse linear algebra calculations [23]. Only recently have high-performance architectures emerged in the context of machine learning accelerators that do not support double-precision floating point operations at all, like the IPU.

In this context, we revisit the MPIR method and combine it with one of two extended precision methods: Double-word arithmetics (detailed in Section III-D) and software emulated double-precision. To the best of our knowledge, we are the first to use double-word arithmetics for the extended precision calculations in the MPIR method.

The method involves a three-step iterative process:

1) Compute the residual $r^{(m)} = b - A \cdot x^{(m)}$ in extended precision.
2) Calculate a correction vector c by solving $A \cdot c^{(m+1)} = r^{(m)}$ in working precision.
3) Update the solution $x^{(m+1)} = x^{(m)} + c^{(m+1)}$ in extended precision.

The method's convergence rate is proportional to the matrix's condition number, and it typically achieves working precision accuracy for matrices that are not too ill-conditioned [20]. In our framework, we utilize either the double-word arithmetic proposed by JOLDES ET AL. [24] or software emulated

```
Tensor rA0 = b - A * x;
Tensor pA = rA0;
While(!terminate, [&] {
 rA0rA = (rA0 * rA).reduce();
 Tensor beta = (rA0rA / rA0rAold) *
               (alpha / omega);
 pA = rA + beta * (pA - omega * AyA);
 Tensor yA = preconditioner.solve(pA);
 AyA = A * yA; // SpMV
 alpha = rA0rA / (rA0 * AyA).reduce();
 Tensor sA = rA - alpha * AyA;
 Tensor zA = preconditioner.solve(sA);
 Tensor tA = A * zA; // SpMV
 omega = (tA * sA).reduce() /
         (tA * tA).reduce();
 x = x + alpha * yA + omega * zA;
 rA = sA - omega * tA;
 rA0rAold = rA0rA;
 terminate = ...;
});
```

Fig. 4: Condensed DSL of the PBiCGStab solver. Our actual implementation contains additional code for setup, residual calculation, early exits due to convergence or singularity, statistics and optional verbose output.

double-precision for steps 1 and 3, while step 2 employs any of the solvers presented in this section in native single precision. As detailed in Section VI-C, this approach effectively mitigates the performance impact of the IPU lacking hardware double-precision support.

C. Preconditioned BiCGStab (PBiCGStab)

The Preconditioned Bi-Conjugate Gradient Stabilized (PBiCGStab) method is a Krylov subspace solver designed for nonsymmetric and symmetric linear systems [25]. It combines elements from the Bi-Conjugate Gradient (BiCG) and Conjugate Gradient Squared (CGS) methods to enhance stability and convergence rates. Preconditioning significantly accelerates the method's convergence, making it a popular choice for solving large sparse systems of equations in computational fluid dynamics, structural analysis, and other fields. Any solver presented in this section can be used as a preconditioner for our PBiCGStab implementation. The method's inherent parallelism allows it to execute across all six worker threads on the IPU without modifications.

Our implementation follows the algorithm presented in [25, p. 638] and is shown in Figure 4. As can be seen, the implementation closely resembles the mathematical formulation of the method.

D. Gauss-Seidel

The Gauss-Seidel method is a relatively simple iterative solver for linear systems. While it guarantees convergence for diagonally dominant or symmetric positive definite matrices [26], it typically converges slowly. However, its smoothing properties make it valuable as a standalone solver in finite volume methods and as a smoother in multigrid algorithms [27].

The method updates the solution vector components sequentially:

$$x_i^{(m+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(m+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(m)} \right) \quad (1)$$

To overcome the inherent sequentiality of Gauss-Seidel, we employ Level-Set Scheduling (detailed in Section V-A) to parallelize the method across all six worker threads on the IPU.

E. ILU and DILU Preconditioners

Incomplete LU Factorization (ILU) and its variant, Diagonal-based Incomplete LU Factorization (DILU), are preconditioners that approximate the LU decomposition of a matrix while maintaining its sparsity pattern. These methods involve two distinct phases: The factorization process and the substitution process.

The factorization process computes approximate lower triangular (L) and upper triangular (U) matrices such that $A \approx LU$. This process follows the standard LU decomposition algorithm but discards fill-in elements to preserve the original sparsity pattern [28]. DILU, a simplified variant, only computes the diagonal elements of U, reducing the computational cost and memory footprint.

Once the L and U matrices are computed, they can be used to solve systems of the form $Ax = b$. This substitution step involves two steps:

1) Forward substitution: Solve $Ly = b$ for y
2) Backward substitution: Solve $Ux = y$ for x

The effectiveness of the (D)ILU preconditioners stems from the fact that the factorization must be calculated only once for the matrix and can be reused for all iterations in iterative solvers as long as the matrix coefficients remain unchanged.

We parallelize the factorization and substitution steps using Level-Set Scheduling (Section V-A) to all six worker threads of each tile of the IPU. Both the factorization and the substitution steps are performed on the IPU.

It is worth noting that variants of the Level-Set Scheduled ILU and DILU preconditioners are patented in the US by NVIDIA [29], [30].

VI. Evaluation

This section presents a comprehensive evaluation of our framework's performance. We begin by detailing the experimental setup in Section VI-A. Section VI-B examines strong and weak scaling characteristics of SpMVs in our framework. In Section VI-C, we analyze the impact of the Mixed-Precision Iterative Refinement method. The evaluation concludes in Section VI-D with a performance comparison between our framework and solvers on traditional CPU and GPU architectures.

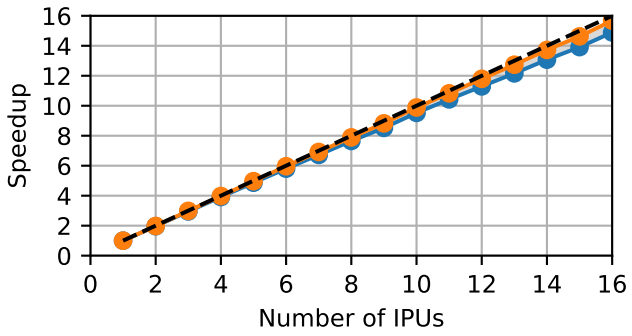


Fig. 5: Strong scaling behaviour of an SpMV with a poisson matrix with 58 M entries (blue dotted line). The orange dotted line represents the speedup of the compute part only.

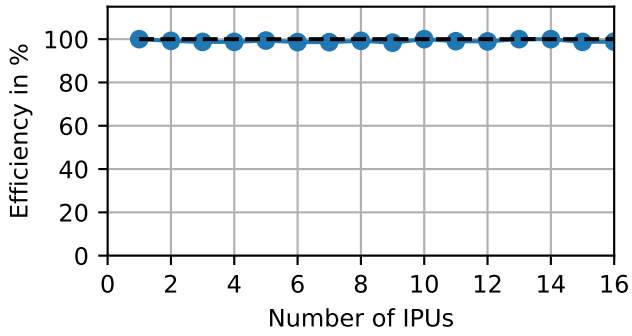


Fig. 6: Weak scaling behaviour of an SpMV with a poisson matrix with 58 M to 890 M entries. The influence of the halo exchange is too small to be visible.

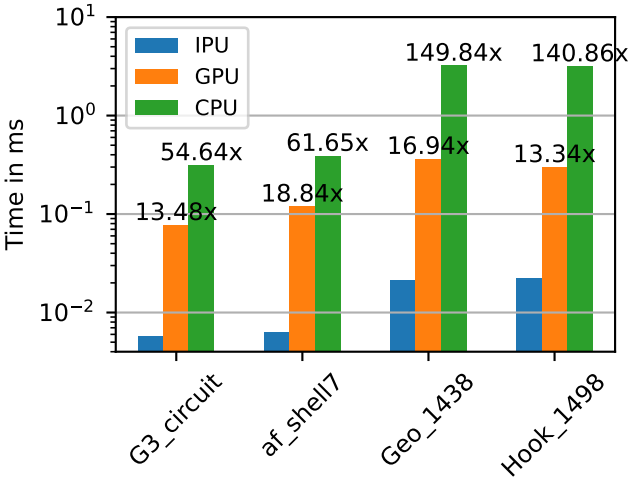


Fig. 7: Execution times of SpMV on different platforms and matrices.

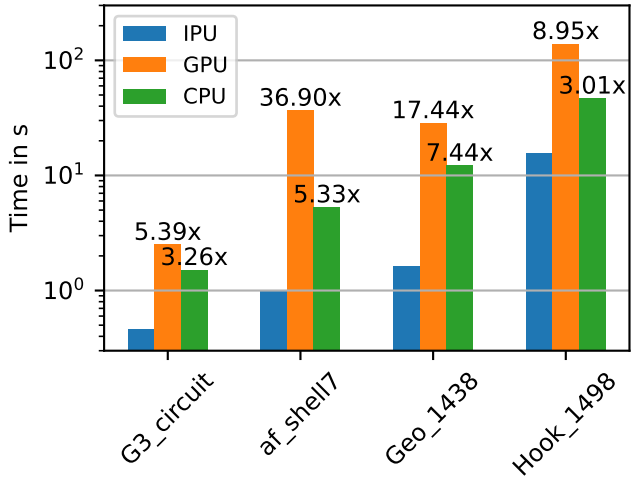


Fig. 8: Time for a IR-PBiCGStab+ILU(0) solver to converge to a relative residual of 10^{-9} on different platforms and matrices.

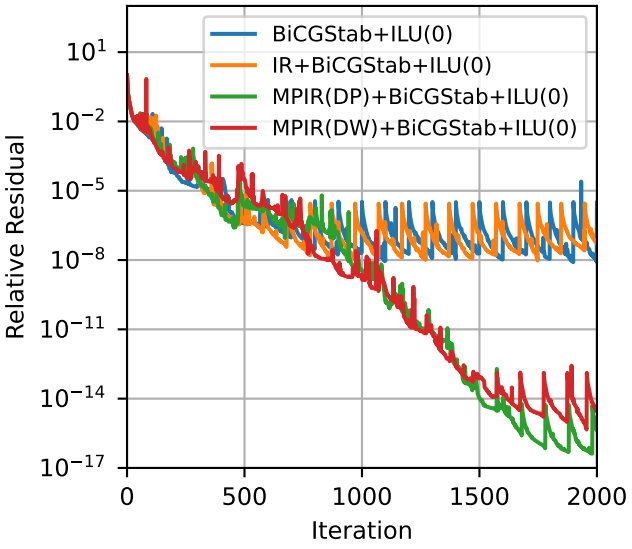


Fig. 9: Convergence plot of different solver configurations for the Geo_1438 matrix.

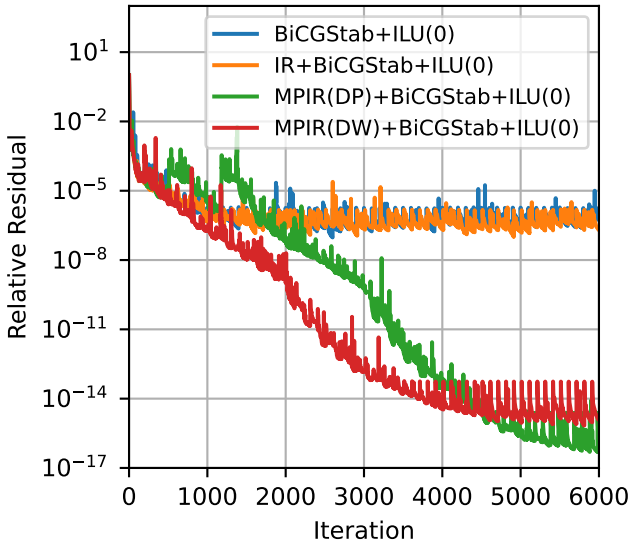


Fig. 10: Convergence plot of different solver configurations for the af_shell7 matrix.

A. Methodology

We conducted performance evaluations of our framework on an IPU-POD16 system, comprising four GraphCore M2000 machines interconnected via IPU-Links. Each M2000 unit contains four IPUs, collectively providing 5,888 tiles, and is equipped with 256 GB of DDR4 RAM. For comparative benchmarking against other architectures, we compared a single M2000 machine to an Intel Xeon Platinum 8470Q CPU and an NVIDIA H100 SXM GPU. Table III presents an overview of these architectures. GraphCore does not specify the TDP for a single IPU chip. Instead, they list 1.100 W for the AC power consumption of the whole M2000 unit. This includes four IPUs, the IPU-Gateway, BMC, RAM, power conversion losses, fans, etc. The maximal power consumption of four IPUs on an M2000 measured with the onboard power sensors during any of our benchmarks was 420 W.

For CPU and GPU benchmarks, we employed the HYPRE solver framework. On NVIDIA GPUs, HYPRE leverages cuSPARSE for fundamental operations such as SpMVs, vector operations, and the ILU preconditioner. cuSPARSE offers both direct and iterative variants of the ILU preconditioner; our testing revealed the direct variant to be consistently faster across all benchmark matrices. While HYPRE provides OpenMP support for certain operations, the ILU preconditioner cannot exploit OpenMP parallelism. Consequently, we opted for the MPI version of HYPRE in all benchmarks, though we observed the OpenMP version to be approximately 20 % faster for SpMVs.

We compiled CPU and GPU code using GCC 13.1.0 and CUDA 12.5, with the flags `-O3 -march=native -mtune=native`. This configuration enables the compiler to exploit all available instructions and optimizations specific to the target CPU.

When measuring execution times of individual operations, like SpMVs, on the CPU or GPU, we warm up their cache by performing 1,000 operations and measure the total execution time of the subsequent 1,000 operations. For the IPU, we use Poplar's profiling feature to measure the required number of cycles. Due to the determinism of the IPU and its constant clock speed, the execution time is the same for every invocation.

We used four matrices from the SuiteSparse Matrix Collection [31] in our benchmarks, all of which are real, symmetric, and positive definite. Table II details these matrices. For scaling benchmarks, we generated matrices by discretizing the Poisson equation on a regular, cubic 3D grid with a 7-point stencil.

B. Strong and weak scaling

To evaluate the strong scaling capabilities of our framework for SpMV operations, we measured execution times as we increased the number of IPUs while keeping the problem size constant. Our test case used a sparse matrix derived from discretizing the Poisson equation on a 200×200×200 grid, containing approximately 58 million non-zero entries. We measured performance both with and without halo exchanges to assess communication impact. As shown in Figure 5, our framework achieves remarkably near-ideal strong scaling for SpMV operations.

The small deviation from perfect linear scaling can be attributed to a fundamental property of domain decomposition. As we distribute a fixed-size problem across an increasing number of IPUs, each processor handles fewer matrix rows, which alters the computation-to-communication ratio. This occurs because the surface-to-volume ratio of the decomposition increases—when a domain is partitioned into smaller subdomains, the proportion of boundary elements relative to interior elements grows. Consequently, the communication volume of halo exchanges increases non-linearly with the number of IPUs, as more elements must be shared across processor boundaries.

For our weak scaling analysis, we employed matrices resulting from discretizing the Poisson equation, ranging from 58 million to 890 million non-zero entries. We maintained a constant computational load per IPU by adjusting the discretization grid size to ensure each tile processed the same number of rows. Figure 6 demonstrates our framework's ideal weak scaling performance for SpMV operations.

While the total communication volume increases linearly with the number of IPUs, the time required for halo exchange remains constant. This is made possible by the IPU's all-to-all communication fabric, which allows all tiles to exchange separator regions with their neighbors simultaneously without incurring additional latency as the system size grows.

C. Mixed-Precision Iterative Refinement

The MPIR method enables our framework to achieve high-precision solutions for large, sparse linear systems on the IPU, which does not natively support double-precision floating-point operations.

To demonstrate this, we compared the convergence behavior of a PBiCGStab+ILU(0) solver in four configurations on the Geo_1438 and af_shell7 matrices: without Iterative Refinement (IR), with IR, and with Mixed-Precision Iterative Refinement (MPIR) in combination with the double-word arithmetic (DW) and soft-float double-precision (DP). For each configuration, the solver performed 100 iterations before either calculating an IR step or restarting directly. Figures 9 and 10 reveal that IR alone, without mixed-precision, did not improve convergence behavior. Both non-MPIR configurations converged only to a relative residual of 10^{-6}, which is inadequate for many applications. In contrast, the MPIR methods achieved convergence to a relative residual of 10^{-13} (double-word) and 10^{-15} (double-precision) for both matrices, demonstrating a significant improvement in the solver's performance.

To assess MPIR's computational overhead, we profiled the execution of the MPIR+PBiCGStab+ILU(0) solver using the double-word arithmetic on the G3_circuit matrix. For this experiment, we chose 10 iterations for the PBiCGStab method. As shown in Table IV, the double-word arithmetic operations (Steps 1 and 3 in the MPIR method) required merely 2 % of the total execution time. This minimal overhead is attributed

Matrix	Rows	Entries
G3_circuit	1.6 M	7.7 M
af_shell7	0.5 M	17.6 M
Geo_1438	1.4 M	63.1 M
Hook_1498	1.5 M	60.9 M

TABLE II: Benchmark matrices

Architecture	Cores	Memory	TDP	General purpose FLOPs
Intel Xeon 8470Q	52 CPUs	208 GB DDR5	350 W (CPU only)	2.3 teraFLOPS FP64
NVIDIA H100 SXM	14592 FP32 CUDA cores	80 GB HBM3	700 W (GPU only)	34 teraFLOPS FP64
GraphCore M2000 (4x Mk2 IPU)	5888 tiles	3.6 GB SRAM + 256 DDR4	1.100 W incl. peripherals, 420 W measured IPUs only	11 teraFLOPS FP32

TABLE III: Benchmark architectures

Operation	Double-Word	Double-Precision
ILU(0) Solve	75%	66%
SpMV	7%	6%
Reduce	12%	11%
Elementwise Ops	4%	3%
Extended-Precision Ops	2%	14%

TABLE IV: Relative computation times of all parts of the MPIR+PBiCGStab+ILU(0) solver with different extended precision methods on G3_circuit. The BiCGStab method performs 10 iterations before calculating an IR step.

to the inner solver (PBiCGStab+ILU(0), Step 2) dominating execution time with its 10 iterations.

These findings demonstrate that combining MPIR and double-word arithmetics can effectively enable high-precision solutions of large sparse linear systems on hardware lacking native double-precision support, while incurring negligible computational overhead.

D. Comparison with CPUs and GPUs

To bring the performance of our framework into relation, we compared the execution times of SpMVs and the total time for the IR-PBiCGStab+ILU(0) solver to converge on the IPU, CPU and GPU architectures. SpMVs are fundamental operations in many solvers and can be seen as a representative benchmark for an architecture's performance in sparse linear algebra [32].

1) SpMV: We evaluated SpMV execution times across the IPU, CPU, and GPU using various matrices. Figure 7 illustrates that the IPU outperformed the GPU by a factor of 13x to 19x and the CPU by a factor of 55x to 150x. This substantial performance gain on the IPU can be attributed to three primary factors: First, the IPU's all-to-all communication fabric is exceptionally well-suited for the communication required for SpMVs. Second, the IPU's cache-less architecture eliminates cache miss penalties, allowing full utilization of its processor performance. And third, the IPUs two-pipeline processor architecture allows for the simultaneous execution of a floating-point instruction and a load, store or integer instruction, which we found particularly beneficial for SpMVs.

2) MPIR+PBiCGStab+ILU(0): We compared the time required for the (MPIR-)PBiCGStab+ILU(0) solver to converge to a relative residual of 10^{-9} (euclidean norm) across different platforms and matrices. Due to the lack of native double precision, the IPU uses the MPIR method in combination with double-word arithmetics. The CPU and GPU uses native double precision without MPIR. Figure 8 demonstrates that the IPU outperformed the GPU by a factor of 5x to 36x and the CPU by a factor of 3x to 7x.

Interestingly, the CPU performs significantly better in this test than in the SpMV benchmark. This improvement is likely because the ILU preconditioner is particularly well-suited to the CPU architecture and not as advantageous for IPUs as initially anticipated. Decomposing the domain across such a large number of small subdomains has a substantial negative impact on the effectiveness of the ILU preconditioner, as it completely disregards halo values.

This effect could potentially be compensated for by employing methods such as the Schur complement [33], which requires solving an additional linear system of equations comprising only the halo values of all tiles. However, this approach is not currently implemented in our framework and would likely necessitate a multi-step process [34], as the resulting additional matrix would likely be too large to be solved on a single tile.

VII. Related Work

While the IPU is designed mainly for machine learning applications, research has also explored its potential for various non-ML tasks. These include DNA and protein sequence alignment [35], breadth-first search (BFS) [36], particle physics simulations [37], and skewed matrix multiplication [38]. However, to the best of our knowledge, our work is the first to solve sparse linear systems on IPUs.

Two studies are particularly relevant to our work because they focus on mesh or grid computations, which could be represented as sparse matrices. Still, none of them solves a linear system of equations.

Buchard et al. investigated unstructured mesh computations on IPUs, using cardiac simulations as an example [12]. Their approach involves explicitly integrating an ODE on an unstructured mesh, which is performed by just two hand-written codelets.

Luow and McIntosh-Smith examined the use of IPUs for traditional HPC applications [39]. Their study included stencil computations and a Lattice-Boltzmann fluid simulation, albeit on structured grids. Notably, they observed that "Expressing our chosen HPC problems in Poplar was not always straightforward compared with familiar HPC technologies such as OpenMP, MPI, and OpenCL". This observation aligns with our experience and motivated our development of CodeDSL and TensorDSL for improved ease-of-use.

VIII. Summary

This paper presented a novel, open-sourced framework [1] for solving large sparse linear systems on IPUs. We introduced two DSLs, CodeDSL and TensorDSL, that enable the

expression of complex algebraic algorithms in a notation close to mathematical formalism. These DSLs are symbolically executed to generate the dataflow graph, execution schedule, and codelets required by the IPU's programming model.

We addressed the IPU's lack of native double-precision support by combining the MPIR method with double-word arithmetics, achieving high-precision solutions without compromising performance. Our framework implements a suite of parallel linear solvers and preconditioners, including PBiCGStab and ILU, optimized for the IPU's unique architecture.

Our novel matrix reordering strategy enables efficient block-wise halo exchanges utilizing the IPU's all-to-all communication fabric. This strategy and the IPU's cache-less design contribute to the framework's exceptional scaling behavior.

Our comprehensive evaluation demonstrated near-ideal strong and weak scaling for SpMVs. Comparative benchmarks revealed that our framework outperforms state-of-the-art CPU and GPU implementations by up to 150x and 36x, respectively, for SpMVs and iterative solvers at similar power draw.

These results highlight the significant potential of IPUs for high-performance sparse linear algebra computations in scientific and engineering applications. Our open-source framework opens new avenues for leveraging specialized hardware in computational science, extending the utility of IPUs beyond their original machine-learning focus.

IX. Acknowledgements

We gratefully acknowledge support by the Federal Ministry of Education and Research (BMBF) under Grant No. 01IS22091.

References

[1] T. Noack, *Graphene Linear Algebra Framework*, Aug. 12, 2024. [Online]. Available: https://github.com/esa-tu-darmstadt/graphene.

[2] *Tile Vertex ISA, IPU2*, GraphCore, Mar. 2022. [Online]. Available: https://docs.graphcore.ai/projects/isa/en/latest/_static/Tile-Vertex-ISA_1.2.3.pdf.

[3] M. Naumov *et al.*, "Amgx: A library for gpu accelerated algebraic multigrid and preconditioned iterative methods," *SIAM Journal on Scientific Computing*, vol. 37, no. 5, S602–S626, 2015. DOI: 10.1137/140980260. eprint: https://doi.org/10.1137/140980260. [Online]. Available: https://doi.org/10.1137/140980260.

[4] *hypre Documentation, Release 2.31.0*, Lawrence Livermore National Laboratory, Feb. 2024. [Online]. Available: https://hypre.readthedocs.io/_/downloads/en/latest/pdf/.

[5] *Incomplete-LU and Cholesky Preconditioned Iterative Methods, Release 12.6*, NVIDIA Corporation, Apr. 2024. [Online]. Available: https://docs.nvidia.com/cuda/pdf/Incomplete_LU_Cholesky.pdf.

[6] J. Scott and M. Tůma, "An introduction to sparse matrices," in *Algorithms for Sparse Linear Systems*. Cham: Springer International Publishing, 2023, pp. 1–18, ISBN: 978-3-031-25820-6. DOI: 10.1007/978-3-031-25820-6_1. [Online]. Available: https://doi.org/10.1007/978-3-031-25820-6_1.

[7] D. R. Kincaid, T. C. Oppe, and D. M. Young, "Itpackv 2d user's guide," May 1989. DOI: 10.2172/7093021. [Online]. Available: https://www.osti.gov/biblio/7093021.

[8] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009, pp. 1–11. DOI: 10.1145/1654059.1654078.

[9] M. Joldes, J.-M. Muller, and V. Popescu, "Tight and rigorous error bounds for basic building blocks of double-word arithmetic," *ACM Trans. Math. Softw.*, vol. 44, no. 2, Oct. 2017, ISSN: 0098-3500. DOI: 10.1145/3121432. [Online]. Available: https://doi.org/10.1145/3121432.

[10] M. Lange and S. M. Rump, "Faithfully rounded floating-point computations," *ACM Trans. Math. Softw.*, vol. 46, no. 3, Jul. 2020, ISSN: 0098-3500. DOI: 10.1145/3290955. [Online]. Available: https://doi.org/10.1145/3290955.

[11] T. Noack, *TwoFloat: Double-Word Arithmetics in C++*. Aug. 12, 2024. [Online]. Available: https://github.com/esa-tu-darmstadt/twofloat.

[12] L. Burchard, K. Hustad, J. Langguth, and X. Cai, "Enabling unstructured-mesh computation on massively tiled ai processors: An example of accelerating in silico cardiac simulation," *Frontiers in Physics*, vol. 11, 2023. DOI: https://doi.org/10.3389/fphy.2023.979699.

[13] E. Anderson and Y. Saad, "Solving sparse triangular linear systems on parallel computers," *Int. J. High Speed Comput.*, vol. 1, no. 1, pp. 73–95, Apr. 1989, ISSN: 0129-0533. DOI: 10.1142/S0129053389000056. [Online]. Available: https://doi.org/10.1142/S0129053389000056.

[14] J. H. Saltz, "Aggregation methods for solving sparse triangular systems on multiprocessors," *SIAM J. Sci. Stat. Comput.*, vol. 11, no. 1, pp. 123–144, Jan. 1990, ISSN: 0196-5204.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd. The MIT Press, 2009, ISBN: 0262033844.

[16] T. A. Davis, *Direct Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2006. DOI: 10.1137/1.9780898718881. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9780898718881. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9780898718881.

[17] R. K. Ghosh and G. P. Bhattacharjee, "A parallel search algorithm for directed acyclic graphs," *BIT*, vol. 24, no. 2, pp. 133–150, Jun. 1984, ISSN: 0006-3835. DOI:

10.1007/BF01937481. [Online]. Available: https://doi.org/10.1007/BF01937481.

[18] T. Noack, *IPU Threading Library*. Aug. 12, 2024. [Online]. Available: https://github.com/esa-tu-darmstadt/ipu-thread-sync.

[19] C. B. Moler, "Iterative refinement in floating point," *J. ACM*, vol. 14, no. 2, pp. 316–321, Apr. 1967, ISSN: 0004-5411. DOI: 10.1145/321386.321394. [Online]. Available: https://doi.org/10.1145/321386.321394.

[20] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, Second. Society for Industrial and Applied Mathematics, 2002. DOI: 10.1137/1.9780898718027. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9780898718027. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9780898718027.

[21] N. J. Higham and T. Mary, "Mixed precision algorithms in numerical linear algebra," *Acta Numerica*, vol. 31, pp. 347–414, 2022. DOI: 10.1017/S0962492922000022.

[22] J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, and J. J. Dongarra, "Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems)," *ACM/IEEE SC 2006 Conference (SC'06)*, pp. 50–50, 2006. [Online]. Available: https://api.semanticscholar.org/CorpusID:2482611.

[23] A. Buttari, J. J. Dongarra, J. Kurzak, P. Luszczek, and S. Tomov, "Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy," *ACM Trans. Math. Softw.*, vol. 34, 17:1–17:22, 2008. [Online]. Available: https://api.semanticscholar.org/CorpusID:3047581.

[24] M. Joldes, J.-M. Muller, and V. Popescu, "Tight and rigorous error bounds for basic building blocks of double-word arithmetic," *ACM Trans. Math. Softw.*, vol. 44, no. 2, Oct. 2017, ISSN: 0098-3500. DOI: 10.1145/3121432. [Online]. Available: https://doi.org/10.1145/3121432.

[25] H. A. van der Vorst, "Bi-cgstab: A fast and smoothly converging variant of bi-cg for the solution of non-symmetric linear systems," *SIAM Journal on Scientific and Statistical Computing*, vol. 13, no. 2, pp. 631–644, 1992. DOI: 10.1137/0913035. eprint: https://doi.org/10.1137/0913035. [Online]. Available: https://doi.org/10.1137/0913035.

[26] T. Sauer, *Numerical Analysis*, 2nd. USA: Addison-Wesley Publishing Company, 2011, ISBN: 0321783670.

[27] M. Adams, M. Brezina, J. Hu, and R. Tuminaro, "Parallel multigrid smoothing: Polynomial versus gauss–seidel," *Journal of Computational Physics*, vol. 188, no. 2, pp. 593–610, 2003, ISSN: 0021-9991. DOI: https://doi.org/10.1016/S0021-9991(03)00194-3. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0021999103001943.

[28] Y. Saad, *Iterative Methods for Sparse Linear Systems*, Second. Society for Industrial and Applied Mathematics, 2003. DOI: 10.1137/1.9780898718003. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9780898718003. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9780898718003.

[29] P. Castonguay and R. Strzodka, "System and method for multi-color dilu preconditioner," Patent US9798698B2, Oct. 2017.

[30] R. Strzodka, J. Demouth, and P. Castonguay, "Parallel multicolor incomplete lu factorization preconditioning processor and method of use thereof," Patent US9600446B2, Mar. 2017.

[31] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011, ISSN: 0098-3500. DOI: 10.1145/2049662.2049663. [Online]. Available: https://doi.org/10.1145/2049662.2049663.

[32] K. Asanovic *et al.*, "A view of the parallel computing landscape," *Commun. ACM*, vol. 52, no. 10, pp. 56–67, Oct. 2009, ISSN: 0001-0782. DOI: 10.1145/1562764.1562783. [Online]. Available: https://doi.org/10.1145/1562764.1562783.

[33] Y. Saad, "Schur complement preconditioners for distributed general sparse linear systems," in *Domain Decomposition Methods in Science and Engineering XVI*, O. B. Widlund and D. E. Keyes, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 127–138, ISBN: 978-3-540-34469-8.

[34] S. Kocak and H. Akay, "Parallel schur complement method for large-scale systems on distributed memory computers," *Applied Mathematical Modelling*, vol. 25, no. 10, pp. 873–886, 2001, ISSN: 0307-904X. DOI: https://doi.org/10.1016/S0307-904X(01)00019-1. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0307904X01000191.

[35] L. Burchard, M. X. Zhao, J. Langguth, A. Buluç, and G. Guidi, "Space efficient sequence alignment for sram-based computing: X-drop on the Graphcore IPU," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '23, Denver, CO, USA: Association for Computing Machinery, 2023, ISBN: 9798400701092. DOI: 10.1145/3581784.3607094. [Online]. Available: https://doi.org/10.1145/3581784.3607094.

[36] L. Burchard, J. Moe, D. Schroeder, K. Pogorelov, and J. Langguth, "Ipug: Accelerating breadth-first graph traversals using manycore graphcore ipus," in Jun. 2021, pp. 291–309, ISBN: 978-3-030-78712-7. DOI: 10.1007/978-3-030-78713-4_16.

[37] S. Maddrell-Mander *et al.*, "Studying the potential of graphcore® ipus for applications in particle physics," *Computing and Software for Big Science*, vol. 5, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:257096474.

[38] S. K. Shekofteh, C. Alles, N. Kochendorfer, and H. Froning, "On performance analysis of graphcore ipus: Analyzing squared and skewed matrix multiplication,"

ArXiv, vol. abs/2310.00256, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:263333982.

[39] T. Louw and S. McIntosh-Smith, "Using the Graphcore IPU for traditional hpc applications," English, 3rd Workshop on Accelerated Machine Learning : Co-located with the HiPEAC 2021 Conference, AccML ; Conference date: 18-01-2021 Through 18-01-2021, Dec. 2020. [Online]. Available: http://workshops.inf.ed.ac.uk/accml/.