HUNIPU: Efficient Hungarian Algorithm on IPUs

Cheng Huang
Aarhus University
cheng@cs.au.dk

Alexander Mathiasen
Graphcore
alexm@graphcore.ai

Josef Dean
Graphcore
josefd@graphcore.ai

Davide Mottin
Aarhus University
davide@cs.au.dk

Ira Assent
Aarhus University
ira@cs.au.dk

***Abstract*— The Hungarian algorithm is a fundamental approach for a number of matching problems that find corresponding elements in two sets. For instance, the optimal alignment of proteins. Due its computational complexity, this algorithm takes several hours for only a few thousand elements on a desktop computer. The most common solution to increase the efficiency is parallelization using GPUs. However, GPUs run the same operation in all the threads in a warp. This constraint is a limitation for the Hungarian algorithm that requires to find, in multiple steps, the best matching among a variable number of candidates.**

In this paper, we introduce HUNIPU, a parallel version of the Hungarian algorithm optimized for a novel architecture called Intelligence Processing Unit (IPU). IPUs is a promising avenue for the Hungarian algorithm since each core can independently perform a distinct operation. To benefit from the computational power of the IPU, we design HUNIPU to exploit the computational model of the IPU. We provide a smooth introduction to the IPU model and its challenges and show its flexibility and potential for numerous algorithmic advances. In our extensive experiments, the HUNIPU outperforms the best GPU algorithm on cutting-edge A100 GPU running $6\times$ faster on synthetic datasets and up to $32\times$ on real datasets for graph alignment.

***Index Terms*—IPU, GPU, Hungarian Algorithm, Linear assignment problem**

I. INTRODUCTION

The Hungarian algorithm [1] is the cornerstone of a number of methods that compute correspondences between two sets of elements. Its widespread applications include disparate problems, such as plant location planning [2], 3D shape matching [3], resource allocation for wireless networks [4], and graph alignment [5]–[7]. The Hungarian algorithm solves the Linear Sum Assignment problem aiming to find the one-to-one assignment between n agents and m tasks that minimizes the overall sum of the costs.

Yet, the Hungarian algorithm is impractical for large data as a result of its computational complexity. The most common strategy to overcome the time impediment is to optimize the algorithm to run in parallel. The most efficient Hungarian-based algorithms run on Graphical Processing Units (GPUs) [8], [9] and achieve up to $20\times$ speedup on the CPU-optimized counterpart in matrices for dense graphs. Yet, all the threads in a GPU warp share the same program counter, constraining the algorithm to run the same operation on all threads. Furthermore, the limitations of shared memory and the relatively slow global memory bandwidth pose challenges when accessing the data. Due to these shortcomings, even the best GPU-optimized Hungarian algorithm underperforms on the steps of the algorithms that require returning the best assignment among variable sets of candidate elements. The most efficient Hungarian algorithm [9] takes more than 20 seconds on a 8000×8000 matrix. In typical applications, such as shape matching, that run the Hungarian algorithm hundreds of times, efficiency becomes a bottleneck.

The *Intelligence Processing Unit (IPU)* [10] is a machine learning computing architecture with massive parallelism that features independent memory for each core. As opposed to the GPU, the IPU enables to run different instructions on each thread with no time loss [10]. This distinctive hardware characteristic mitigates the latency issues evident in GPUs, particularly the inefficiencies caused by the retrieval of data from the main memory [11].

Yet, to achieve the desired speedup, the IPU precompiles the code into a static computation graph that is optimized at runtime. As such, the algorithms need to be completely redesigned to minimize the communication and balance workload among the cores. In spite of this, IPUs have led important efficiency improvements in traditional data analysis problems, such as molecular property prediction [12], link prediction on large-scale knowledge graphs [13], differential privacy [14], prediction models [15], graph nearest neighbors [16], and sequence alignment [17].

Motivated by the interest in the problem and the GPUs shortcomings, we first carefully analyze and systematize the IPU strengths and challenges. In light of these findings, we introduce HUNIPU, an IPU-optimized Hungarian algorithm that returns a valid assignment in just a few seconds on a matrix with more than 64 million elements.

In summary, our contributions are as follows.

- We categorize the IPU's main characteristics including the hardware architecture and the software support and summarize the main challenges faced when designing algorithms on IPUs.
- We revamp the Hungarian algorithm to operate on IPUs, and provide HUNIPU, an efficient IPU version.
- We perform a thorough experimental comparison with both GPU- and CPU-optimized algorithms. HUNIPU achieves $6\times$ speedup on synthetic datasets and up to $32\times$ speedup on real-world datasets than the GPU version and up to $3000\times$ speedup than the CPU version.

The project is funded by the AUFF Nova grant no. AUFF-E-2022-9-37.

II. Linear Sum Assignment

A *bipartite graph* is a triple $G = (P, Q, E)$ where P, Q are sets of nodes, whereby $P \cap Q = \emptyset$, $|P| = n$, $|Q| = m$, and $E \subset P \times Q$ is a set of edges. We are given a *cost matrix* $\mathbf{C} \in \mathbb{R}^{P \times Q}$ assigning a positive real cost $\mathbf{C}_{ij}, i \in P$ and $j \in Q$ on each edge. A *matching* is a 1-to-1 correspondence between the nodes in P and those in Q. We encode a matching with a binary matrix $\mathbf{M} \in \{0,1\}^{n \times m}$ with $\mathbf{M}_{ij} = 1$ indicating a correspondence between node $i \in P$ and node $j \in Q$. A *perfect matching* is a correspondence between all the nodes in P and those in Q.

The **Linear Sum Assignment Problem (LSAP)** aims to find a maximum-cardinality *matching* with the smallest overall cost. Without loss of generality, we assume that the graph G is complete, that is $E = P \times Q$ and that P and Q have the same size $|P| = |Q| = n$. As a consequence, the maximum cardinality matching $\mathbf{M}$ is always a perfect matching.

A. The Hungarian Algorithm

The Hungarian algorithm [1] starts with a complete bipartite graph $G = (P, Q, E)$ and a cost matrix $\mathbf{C}$ and computes an initial assignment. Next, it performs *path augmentation* to improve the matching; finally, it updates an auxiliary structure to maintain the candidate matches.

1) Initial assignment: In the first phase, the Hungarian algorithm computes an initial matching. Assume the easy case in which all the entries in $\mathbf{C}$ are 0-cost; this situation allows for an arbitrary matching as long as we enforce one and only match elements in P to those in Q. As such, we aim to fill the cost matrix with as many 0-cost entries as possible. We observe that if we subtract from each entry a constant δ, the final matching is not affected while the number of 0 entries potentially increases. Equipped with this knowledge, we substitute the cost matrix $\mathbf{C}$ with a slack matrix $\mathbf{S}$ computed in two steps

$$\mathbf{S}_{ij} = \mathbf{C}_{ij} - \underbrace{\min_k \mathbf{C}_{ik}}_{\text{min. cost row } i} \qquad \mathbf{S}_{ij} = \mathbf{S}_{ij} - \underbrace{\min_k \mathbf{S}_{kj}}_{\text{min. cost column } j} .$$

By subtracting the minimum column- and row-cost, the slack matrix contains only non-negative elements.

Subsequently, we choose the initial matching from the zero elements in the slack matrix. We denote the chosen edges (i, j) where $\mathbf{S}_{ij} = 0$ as initial *star edges*. When the algorithm terminates, the star edges are those in the optimal assignment. Yet, the initial assignment might not be the optimum, as some nodes might remain unmatched.

2) Path Augmentation: Path augmentation iteratively finds and refines certain paths in a graph to improve matching. Although node $i \in P$ may be assigned to $j \in Q$ at first, such an assignment may not be the optimal one. To compute the optimal assignment, we first arbitrarily select a 0-cost edge, called the *prime edge*. The prime edge is a candidate to become the star edge in the process of seeking an optimal assignment. We initiate path augmentation iteration with an unmatched node $i \in P$ and select a 0-cost edge (i, j) that becomes a prime edge. Next, we traverse the star edge (k, j) connected to j. The algorithm iterates the selection of a prime edge from k until there is no further star edge in Q to traverse. Since the algorithm terminates on a node in Q, the number of prime edges is one more than that of star edges. At this point, we convert all the prime edges to star edges and discard all the initial star edges. This refinement process yields an augmentation of the total assignment by one.

3) Slack matrix update: After path augmentation, if some node remains unmatched, and there is no 0-cost edge, the algorithm updates the slack matrix as follows. First, we detect the *uncovered edges* (i, j) with the minimum value $\mathbf{S}_{ij}$. Next, we add a constant Δ to the edges having the corresponding row and column covered and subtract Δ from uncovered edges. This operation produces at least one uncovered 0-edge. By repeating this matrix slack update followed by path augmentation that adds one covered edge to the candidate assignment, the algorithm eventually terminates with a perfect matching.

The Hungarian algorithm requires iterative execution of path augmentation and slack matrix update multiple times to compute the minimum sum assignment. Notably, these steps involve checking and updating the entire slack matrix, leading to considerable overhead, requiring the update of candidate assignments on variable numbers of star edges. This variability is ill-suited for GPUs that require running the same operations on multiple threads. We thus propose exploiting the IPU architecture that offers more flexibility.

III. The Intelligence Processing Unit (IPU)

An IPU consists of several *tiles* where each tile contains a processor and dedicated memory [10]. A fast (8TB/s theoretical), all-to-all communication network called *exchange fabric* connects the tiles. On a multi-IPU architecture, the exchange fabric extends to all tiles on all of the IPUs. As an example, the Colossus MK2 GC200 IPU [18] has 1472 tiles, each tile contains one core with six threads and its own local high-bandwidth (47.5TB/s, aggregate) and low latency (6 clock cycles) SRAM memory with 624 KiB [16]. Hence, a single IPU chip has 8832 working threads and 900 MiB in processor memory in total.

Unlike traditional processors like CPUs and GPUs, IPUs do not have global memory and shared memory but only internal tile memory. Moreover, unlike GPUs featuring Single Instruction, Multiple Thread (SIMT) architecture, IPUs employ a Multiple Instruction, Multiple Data (MIMD) architecture, wherein each thread has completely distinct code and execution flow without incurring performance penalties [10]. Furthermore, IPUs have built-in SRAM within the tiles, significantly reducing the latency incurred in GPUs due to data transfer from the global memory.

A. IPU Low-level communication

Here we delve into the low-level communication to the IPU, through the Poplar [19] programming framework. Poplar utilizes a static computation abstraction called the *computational graph*, whereby a vertex is a task and an edge is a data flow.

The data is represented by a *multi-dimensional* tensor; each tensor must explicitly map to the tile's memory. Similarly, each task must map to a number of tiles to specify the execution position. Each operation, including loop and branching, and the tensor dimension must be *defined at compile time*. Due to the static nature of the computational graph, the data exchange among tiles must be defined ahead at compile time.

Poplar follows Valiant's Bulk-Synchronous Parallel (BSP) computational model [20], wherein the execution of a task is split into compiler-optimized steps. Each step consists of three phases namely, the compute, synchronization, and exchange phase. In the compute phase, all the tiles perform independent isolated operations. Next, in the synchronization phase, the compiler ensures that all the tiles completed their task. Finally, in the exchange phase, the tiles share the results of the computation. The compiler enforces the BSP model by grouping operations into *compute sets*. Each compute set executes operations in parallel on distinct tiles. No thread from another tile can modify the tensors within the same compute set. This ensures data integrity and consistency across the parallel computations.

B. Algorithm design on IPUs

IPUs present a significant architectural shift, whereby algorithms need a conceptual revamp to exploit the advantages and cope with the challenges. Here, we describe the main considerations of IPUs for algorithm design that are the core of our IPU-optimized Hungarian algorithm.

(C1) Lack of atomic operations: IPUs do not support atomic operations. As such, multiple threads on a tile share the memory, incurring the risk of race conditions.

(C2) Modest tile memory: Each tile has only 624 KB of memory. As such, partitioning the data and the computational tasks on multiple tiles is paramount. Yet, inadequate task- and data-mapping strategies might cause inefficiencies due to excessive data exchange among the tiles.

(C3) Synchronization: IPUs operate under the BSP model, all tasks must terminate before moving to the next operation. In other words, the time for each phase is determined by the tile that terminates last. As such, task imbalance generates idle IPU-time and waste of computational resources.

(C4) Slow dynamic operations: The static nature of the computational graph poses challenges to dynamic operations. Dynamic slicing of tensors modifies or retrieves the value at a specified index. Since indices are created at runtime, each IPU is unaware of the value of the index in other tiles. As such, the compiler must facilitate internal exchange without prior knowledge of the indices, which therefore consumes a significant portion of memory and renders the computation inefficient.

Next, we describe how to address these challenges in HUNIPU, our efficient IPU-optimized Hungarian algorithm.

IV. HUNIPU

Our IPU-optimized Hungarian Algorithm (HUNIPU) capitalizes on the advantages of the IPU architecture and overcomes its constraints with an intelligent mapping strategy (Section IV-A), a novel matrix compression scheme (Section IV-B), and redesign of the algorithm (Sections IV-C-IV-H) into six steps.

A. Mapping strategy

Mapping data to the IPU tiles in advance requires a careful plan, as the *compiler does not automatically handle the assignment of data to the IPU's memory*.

In the context of matrix-based data mapping, the IPU offers two main strategies, the 1D and the 2D decomposition [21], [22]. In our context, the 1D decomposition selects a subset of the bipartite graph's vertices and their outgoing edges to be assigned to a single tile. The 2D decomposition partitions the matrix along rows and columns and allocates each partition to different tiles.

We observe that in 1D decomposition, each tile encompasses all the outgoing edges of a node; as a result, a single IPU tile has access to the row of the slack matrix. This feature is particularly appealing for the Hungarian algorithm that examines and stores the zero-status for each row. In contrast, the 2D decomposition forces each tile to only access a subset of the outgoing edges. Each tile lands only a part of the row and needs to communicate with other tiles to access the row's zero-status. Therefore, we choose the 1D decomposition mapping strategy. To ensure balanced workloads among all tiles, we enforce an equal number of rows for each tile **(C3)**.

B. Matrix Compression

The Hungarian algorithm operates solely on the zero-elements within the slack matrix; thus, having access to the zero-elements in advance can considerably enhance the efficiency of the algorithm. To this end, we compress the slack matrix to store the zero elements' position and capitalize on the IPU multithreading capabilities.

Even though each IPU tile avails six threads, the IPU does not support atomic operations between the threads **(C1)**. Therefore, it is difficult to collect and count the number of zeros for each row atomically. The naïve solution would use only one thread for each row, but this may be inefficient because the six threads in the IPU tiles execute one instruction at a time in a fixed order. To solve this challenge, we partition each row of the matrix into six segments of approximately equal size to ensure workload balance across threads **(C3)**. The threads work in parallel to count and store the position of zeros within the respective segment.

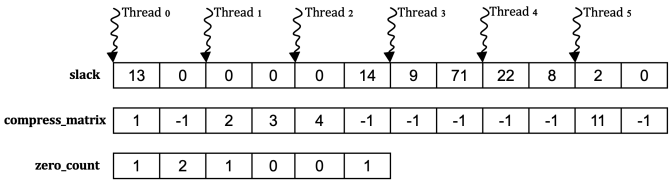

Fig. 1: Compressing the slack matrix (top), recording the 0-element position (middle) and counting the number of zero elements for each segment (bottom).

This compression scheme improves the efficiency by scanning only part of the row to identify uncovered zero elements and only examining zero elements in the segments.

C. Step 1 - Initial Subtraction

Step 1 computes the slack matrix $\mathbf{S}$ from the cost matrix $\mathbf{C}$. To fully harness the IPU power, we apply the Poplar's reduce operation that computes the minimum value in each row, followed by a subtraction operation of the minimum from the respective row in parallel. Subsequently, we repeat the same parallel operations for each column. We further increase the parallelism during the matrix update by dividing each row into six segments, one for each thread. We retrieve and update from the tile's memory two floats at once due to the *IPU efficient handling of two floats at a time.*

D. Step 2 - Initial Matching

Step 2 chooses the initial matching among the 0-elements of the slack matrix $\mathbf{S}$. For each uncovered zero, the algorithm covers the respective row and column and marks the 0-element with a star. We repeat this process for each uncovered zero in the slack matrix.

A zero element is covered if it belongs to a previously covered row or column. Depending on the order of operations of Step 2, this could lead to race conditions in a parallel setting **(C1)**. To solve this challenge, first, we compress the slack matrix as described in Section IV-B. Then, we count the zeros in each row concurrently, and identify the maximum number τ of zeros across all rows using a reduction operation (in Figure 2, τ = 2). Next, we apply Poplar's sort operation to sort all the rows of the compress matrix in descending order in parallel. Finally, we compute the initial matching by scanning only the top τ columns of the sort compress matrix.

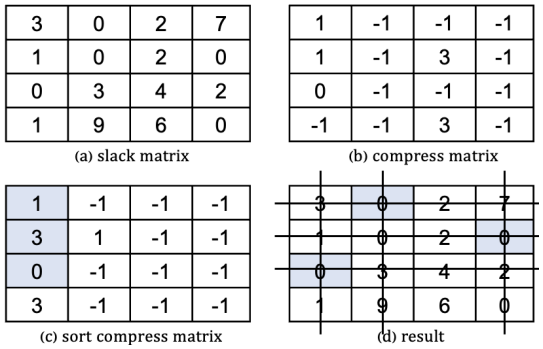

Fig. 2: Example of finding the initial matching from the slack matrix (a) which is compressed (b) and sorted (c), column-wise mark zero elements, cover rows and columns (d).

E. Step 3 - Completion Assessment

Step 3 assesses whether the algorithm found the optimal assignment. Specifically, we update the col_cover storing the cover status for each column according to the col_star storing the position of the star-zero for each column and check whether all the columns are covered. Specifically, we cover the column if it contains a star-zero. If we use a naïve mapping-to-tile strategy for these variables, each execution of Step 3 requires exchanging both variables among tiles, significantly hampering the algorithm performance. To overcome this situation and maintain a balanced workload per tile, we partition col_cover and col_star in segments of 32 elements each[1] and map each segment to a tile **(C3)**. As a result, all tiles can update the col_cover variable in parallel. Finally, we apply a *reduce* operation to the binary tensor col_cover to count the number of covered columns. If any of the columns remain uncovered, the algorithm advances to Step 4.

F. Step 4 - Search for Alternating Path in the Bipartite Graph

Step 4 identifies a non-covered zero and primes it. Yet, scanning the entire slack matrix can be computationally intensive. Note that a row can be in one of these three states: (-1) a row with no uncovered zeros, (0) a row with both an uncovered and a starred zero, or (1) a row with an uncovered zero but no starred zeros. As such, we proceed in two steps. We first assess each row's state and, second, we collect the information for the entire matrix. To this end, we use the $zero\_status$ variable to store the state for each row, using -1, 0 and 1 as flags. We assign each thread a single row of the compress matrix to check and record the status of uncovered zero into the $zero\_status$. As the compressed matrix contains only the positions of zero-elements, this strategy considerably boosts the efficiency of Step 4.

Finally, we perform a reduction operation to determine the maximum value of $zero\_status$. If the maximum is -1, the algorithm proceeds to Step 6 to introduce uncovered zeros; if it is 1, the algorithm advances to Step 5 to find an augmenting path; otherwise, we prime the zero-element, cover its row, uncover its column, and reiterate Step 4.

G. Step 5 - Path Augmentation

Step 5 finds alternative paths between the prime-zero and the star-zero. By traversing from the prime-zero to a star-zero, the algorithm increases the size of the assignment by one. Specifically, we execute Step 5 in two sub-steps. First, we traverse the prime-zero and store it in the $green\_column$ variable. Then we traverse the path in the reverse direction to flag with a star the prime-zero.

The algorithm starts with the uncovered zero from Step 4 and primes it. We update the $green\_column$ to record the position of the prime-zero and check if there is a star-zero in the prime-zero column. If this is the case, as a consequence of Step 4, there must also be a prime-zero in the same row as the star-zero. After detecting the prime-zero, we update the $green\_column$ with its position. We repeat this process until there is no star-zero in the same column with the prime-zero. Figure 3(a) highlights the primed zero in green.

After the first step, we traverse the green elements in reverse order, mark the green zero with a star, and remove any previously starred zeros and primed zeros (Figure 3(b)).

From Step 5, we observe that these two steps require frequent updates or retrieval of tensor values. Moreover, the

[1]The size needs to be decided at compile time. We empirically find that 32 works well regardless of the data and the architecture.

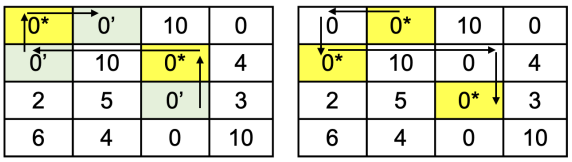

Fig. 3: Example of path augmenting: primed zero in green (a), traverse green elements in reverse order to update (b).

specific position index within the tensor for these operations is computed at runtime, conflicting with the static nature of the computational graph. To address this issue, we propose solutions for dynamic operations on IPUs **(C4)**.

A possible solution to avoid unnecessary data exchange is mapping the entire tensor to a single tile. Yet, this solution easily exceeds tile memory that is limited to 624 KiB **(C2)**.

Our solution instead partitions the tensor and distributes its segments across multiple tiles. Then, in the dynamic update, we carefully record the starting and ending positions of each tensor's segment and ensure each segment maps to the correct tile. With this configuration, all tiles operate concurrently. This partition-and-distribute strategy can also be used in dynamic slicing. Figure 4 shows an example. Let us consider a tensor partitioned into 3 tiles, with each tile containing 3 elements. Given the index 7, the element we intend to dynamically slice is located in tile 2 with column 1. Its position can be calculated as row $\frac{7}{3} = 2$ and column $7\%3 = 1$. To execute the dynamic slicing, the three tiles process the respective segments in parallel to obtain and store in a temporary tensor the elements $2, 6, 10$ from the column 1. Then, we slice the element 2 from the temporary tensor as our final dynamic slice result. Since an IPU contains only 1472 tiles, the maximum length of the temporary tensor is 1472. As such, the slicing of the temporary tensor can be performed in a single tile as we are within the 624 KiB memory boundaries.

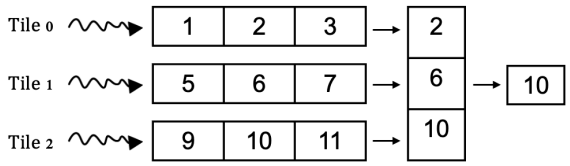

Fig. 4: Example of dynamic slicing of element with index 7.

H. Step 6 - Slack Matrix Update

Step 6 identifies the minimum uncovered value and updates the slack matrix to introduce at least one zero-element.

We segment each row into same size segments and assign each segment to a thread to compute its minimum value in parallel. Consequently, each row yields six values upon completion. We retrieve two float values from tile memory at once which is more efficient on the IPU. We compare each pair, store the smaller value, and return the segment minimum based on the minimum value of the pair. After obtaining the six minimums for each row, we call a reduction operation to determine the minimum in the row. Another reduction computes the overall minimum value in the entire matrix.

After retrieving the minimum uncovered value from the slack matrix, each thread independently processes its segment to update the matrix. We retrieve two float values from tile memory at once, subtract the minimum from the uncovered elements in parallel and then add it to the elements covered in the row and the column. Then we re-compress the slack matrix after the modification of the slack matrix.

V. Experimental evaluation

Experiment Setup. We run HunIPU[2] on a 1.325GHz Mk2 IPU with 900MB in-processor memory and Poplar SDK 3.2.0. The CPU version runs on a high-performance AMD EPYC 7742 2.25GHz 64-Core Processor; the GPU version runs on a modern Nvidia A100 GPU with 40GB VRAM.

Baselines. We compare our HunIPU with

- **CPU**: A fast CPU implementation of the Hungarian algorithm.[3]
- **FastHA** [9]: The state-of-the-art GPU-optimized Hungarian algorithm.[3]

Dataset. We conduct experiments on both real and synthetic datasets. We generate synthetic datasets following Gaussian distributions of different densities. To study the impact of density of values in the cost matrix, we explore *values in the range* $[1, k \times n]$ with $k \in \{1, 10, 100, 500, 1\,000, 5\,000, 10\,000\}$ and n the size of the cost matrix. We generate square cost matrices of size $512, 1\,024, 2\,048, 4\,096, 8\,192$. We set mean $\mu = \frac{k \cdot n}{2}$ and standard deviation $\sigma = \frac{k \cdot n}{6}$.

Use case. Our algorithm and results transfer to any use of the linear assignment problem. Yet, to validate the speedup on a real scenario, we evaluate the methods on the graph alignment problem [5]. We evaluate the methods on three real world datasets HighSchool [23], Voles [24] and MultiMagna [25]. The dataset characteristics are summarized in Table I.

TABLE I: Characteristics of the real graph data in terms of number of nodes n, number of edges m, and network type.

Dataset	n	m	Type
MultiMagna [25]	1 004	8 323	biological
HighSchool [23]	327	5 818	proximity
Voles [24]	712	2 391	proximity

A. HunIPU *vs. Hungarian algorithm on CPU*

We first compare our IPU-optimized algorithm HunIPU with the best CPU implementation. Table II show the relative gain achieved by HunIPU compared to the optimized CPU implementation for Gaussian-distributed data. HunIPU consistently outperforms the CPU version, expediting the calculation by a factor of 6 to 3000. Notably, due to the parallelization during the update of the slack matrix, HunIPU increases the speedup with larger matrices and for larger value ranges. Large value range implies sparser (i.e., less dense) cost matrices, allowing HunIPU to exploit the parallelization more effectively. We observe a similar speedup with uniformly distributed data (omitted in the interest of space).

[2]https://github.com/kouteisang/HunIPU

[3]https://github.com/paclopes/HungarianGPU

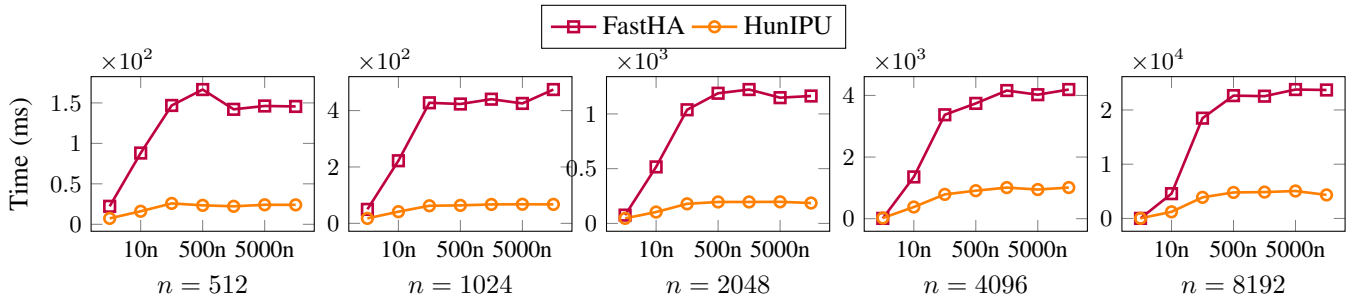

Fig. 5: Runtime of FastHA compared to HunIPU on different matrix size and value range on Gaussian distributed data.

TABLE II: Runtime gain of HunIPU compared to optimized Hungarian algorithm on CPU on Gaussian distributed data.

	1n	**10n**	**100n**	**500n**	**1000n**	**5000n**	**10000n**
512	22.49	51.86	56.73	60.33	64.00	52.59	60.21
1024	56.28	141.79	198.65	194.21	188.68	188.62	204.61
2048	89.46	418.82	525.62	567.65	596.71	531.35	578.33
4096	42.61	927.48	1200.23	1186.28	1155.45	1222.59	1051.89
8192	76.19	1870.44	2902.6	2761.65	2871.69	2880.34	3041.57

B. HunIPU *vs.* FastHA

Here, we compare HunIPU with FastHA [9]. Figure 5 shows the runtime varying the matrix size and the value range for data generated using Gaussian distributions. The results show a consistent advantage over FastHA across distributions and ranges. Even in the case in which the values in the cost matrix are similar, HunIPU maintains a competitive edge. The improvement ranges from $3\times$ to $11\times$ with average speedup of $6\times$. We observe a similar speedup with uniformly data.

C. Use case: Graph Alignment

Graph alignment aims to generate a similarity matrix derived from two graphs' adjacency matrices, representing the pairwise node similarities between the nodes of the two graphs [5]. Given such a similarity matrix, the Hungarian algorithm identifies the pairwise node-to-node correspondence with the maximum similarity.

The evaluation of graph alignment algorithms aligns a graph with its *noisy version*, obtained by modifications of the edges of the input graph [5]. We align the last snapshot of the graph with modified versions featuring different percentages of edges. We evaluate HunIPU and report the average runtime.

We employ the GRAMPA alignment algorithm [26] to compute the similarity matrix. Yet, any choice of the algorithm would suffice, as our method only requires the similarity matrix. GRAMPA features a hyper-parameter, η, which we set to the default recommended value $\eta = 0.2$.

FastHA can only operate on matrix size 2^m, where m is a non-negative integer. To accommodate this, we pad the similarity matrix by filling it with 0-rows and -columns to the nearest 2^m size. After padding, we run FastHA and HunIPU on the similarity matrices and compare their performance.

Table III confirms on real world dataset the runtime results achieved with the synthetic datasets. In particular, HunIPU outperforms FastHA on different noise levels, achieving $5\times$ to $32\times$ speedup. This vindicates our HunIPU as the method of choice in practical applications.

TABLE III: Runtime (ms) on real world graph alignment datasets. HunIPU outperforms the best GPU-optimized Hungarian FastHA algorithms by nearly an order of magnitude.

(a) HighSchool

Edges	**80%**	**90%**	**95%**	**99%**
HunIPU	68.32	68.80	55.69	97.73
FastHA	1258.39	1243.34	1103.90	2541.52

(b) Voles

Edges	**80%**	**90%**	**95%**	**99%**
HunIPU	419.79	332.01	307.96	322.05
FastHA	13251.8	10834.5	8722.55	9896.91

(c) MultiMagna

Edges	**Variant1**	**Variant2**	**Variant3**	**Variant4**	**Variant5**
HunIPU	285.26	382.87	430.44	417.42	422.92
FastHA	1658.74	2024.22	2246.89	2407.45	2461.41

VI. Conclusion

We introduce HunIPU, a solution for the Hungarian algorithm on the IPU, a highly parallel Multiple-Instruction-Multiple-Data (MIMD) architecture for AI/ML tasks that supports simultaneous different operations on its cores, grouped in tiles. As opposed to the GPU, the IPU offers dedicated, low-latency, memory to each tile, increasing the possibilities for parallelism. We provide a characterization of the IPU architecture in terms of its algorithmic potential as well as challenges in its programming model. Concretely, challenges lie in the lack of atomic operations in multi-threading applications, need for tile synchronization, enforcement of static operations, and limited memory. Addressing these challenges, we devise HunIPU, an IPU-optimized Hungarian algorithm for the linear assignment problem that offers dynamic updates of tensors. Our thorough experiments show that HunIPU outperforms state-of-the-art GPU-optimized algorithms by factors of $3\times$ to $32\times$ on different data distributions. These results show that IPUs are also amenable to algorithms beyond standard machine learning tasks, opening for significant potential for novel efficient algorithms.

Acknowledgments

We thank Carlo Luschi and the Graphcore team for their valuable feedback and support in understanding the IPU.

REFERENCES

[1] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval research logistics quarterly*, vol. 2, no. 1-2, pp. 83–97, 1955.

[2] T. C. Koopmans and M. Beckmann, "Assignment problems and the location of economic activities," *Econometrica*, vol. 25, no. 1, pp. 53–76, 1957. [Online]. Available: http://www.jstor.org/stable/1907742

[3] C. N. Vasconcelos and B. Rosenhahn, "Bipartite graph matching computation on GPU," ser. Lecture Notes in Computer Science, vol. 5681. Springer, 2009, pp. 42–55. [Online]. Available: https://doi.org/10.1007/978-3-642-03641-5_4

[4] H. Yin and H. Liu, "An efficient multiuser loading algorithm for ofdm-based broadband wireless systems," in *Globecom '00 - IEEE. Global Telecommunications Conference. Conference Record (Cat. No.00CH37137)*, vol. 1, 2000, pp. 103–107 vol.1.

[5] K. Skitsas, K. Orlowski, J. Hermanns, D. Mottin, and P. Karras, "Comprehensive evaluation of algorithms for unrestricted graph alignment," 2023, pp. 260–272. [Online]. Available: https://doi.org/10.48786/edbt.2023.21

[6] R. Singh, J. Xu, and B. Berger, "Global alignment of multiple protein interaction networks with application to functional orthology detection," *Proceedings of the National Academy of Sciences*, vol. 105, no. 35, pp. 12 763–12 768, 2008. [Online]. Available: https://www.pnas.org/doi/abs/10.1073/pnas.0806627105

[7] J. Hermanns, A. Tsitsulin, M. Munkhoeva, A. Bronstein, D. Mottin, and P. Karras, "Grasp: Graph alignment through spectral signatures." Springer International Publishing, 2021, pp. 44–52.

[8] K. Date and R. Nagi, "Gpu-accelerated hungarian algorithms for the linear assignment problem," *Parallel Comput.*, vol. 57, pp. 52–72, 2016. [Online]. Available: https://doi.org/10.1016/j.parco.2016.05.012

[9] P. A. Lopes, S. S. Yadav, A. Ilic, and S. K. Patra, "Fast block distributed cuda implementation of the hungarian algorithm," *Journal of Parallel and Distributed Computing*, vol. 130, pp. 50–62, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731519302254

[10] Z. Jia, B. Tillman, M. Maggioni, and D. P. Scarpazza, "Dissecting the graphcore IPU architecture via microbenchmarking," *CoRR*, vol. abs/1912.03413, 2019. [Online]. Available: http://arxiv.org/abs/1912.03413

[11] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A performance study of general-purpose applications on graphics processors using CUDA," *J. Parallel Distributed Comput.*, vol. 68, no. 10, pp. 1370–1380, 2008. [Online]. Available: https://doi.org/10.1016/j.jpdc.2008.05.014

[12] H. Helal, J. Firoz, J. A. Bilbrey, M. M. Krell, T. Murray, A. Li, S. S. Xantheas, and S. Choudhury, "Extreme acceleration of graph neural network-based prediction models for quantum chemistry," *CoRR*, vol. abs/2211.13853, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2211.13853

[13] A. Cattaneo, D. Justus, H. Mellor, D. Orr, J. Maloberti, Z. Liu, T. Farnsworth, A. W. Fitzgibbon, B. Banaszewski, and C. Luschi, "BESS: balanced entity sampling and sharing for large-scale knowledge graph completion," *CoRR*, vol. abs/2211.12281, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2211.12281

[14] E. H. Lee, M. M. Krell, A. Tsyplikhin, V. Rege, E. Colak, and K. W. Yeom, "Nanobatch DPSGD: exploring differentially private learning on imagenet with low batch sizes on the IPU," *CoRR*, vol. abs/2109.12191, 2021. [Online]. Available: https://arxiv.org/abs/2109.12191

[15] Z. Zhang and S. Zohren, "Multi-horizon forecasting for limit order books: Novel deep learning approaches and hardware acceleration using intelligent processing units," *CoRR*, vol. abs/2105.10430, 2021. [Online]. Available: https://arxiv.org/abs/2105.10430

[16] L. Burchard, J. Moe, D. T. Schroeder, K. Pogorelov, and J. Langguth, "Ipug: Accelerating breadth-first graph traversals using manycore graphcore ipus." Springer-Verlag, 2021, p. 291–309. [Online]. Available: https://doi.org/10.1007/978-3-030-78713-4_16

[17] 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," *CoRR*, vol. abs/2304.08662, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2304.08662

[18] Graphcore. Ipu - intelligence processing unit. Accessed: November 12, 2023. [Online]. Available: https://www.graphcore.ai/products/ipu

[19] Cambrian AI Research. (2022) Graphcore's ai software stack is now customer-driven. https://cambrian-ai.com/downloads/graphcores-ai-software-stack-is-now-customer-driven/. Online.

[20] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, p. 103–111, aug 1990. [Online]. Available: https://doi.org/10.1145/79173.79181

[21] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek, "A scalable distributed parallel breadth-first search algorithm on bluegene/l," in *SC'05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. IEEE, 2005, pp. 25–25.

[22] A. Buluç, S. Beamer, K. Madduri, K. Asanovic, and D. Patterson, "Distributed-memory breadth-first search on massive graphs," *arXiv preprint arXiv:1705.04590*, 2017.

[23] J. Fournet and A. Barrat, "Contact patterns among high school students," *CoRR*, vol. abs/1409.5318, 2014. [Online]. Available: http://arxiv.org/abs/1409.5318

[24] S. Davis, B. Abbasi, S. Shah, S. Telfer, and M. Begon, "Spatial analyses of wildlife contact networks," *Journal of the Royal Society, Interface / the Royal Society*, vol. 12, 01 2015.

[25] V. Vijayan and T. Milenkovic, "Multiple network alignment via multimagna++," *IEEE ACM Trans. Comput. Biol. Bioinform.*, vol. 15, no. 5, pp. 1669–1682, 2018. [Online]. Available: https://doi.org/10.1109/TCBB.2017.2740381

[26] Z. Fan, C. Mao, Y. Wu, and J. Xu, "Spectral graph matching and regularized quadratic relaxations i: The gaussian model," 2019.