Solving Sparse Assignment Problems on FPGAs

ERLING JELLUM, MILICA ORLANDIĆ, EDMUND BREKKE, TOR JOHANSEN, and TORLEIV BRYNE, Norwegian University of Science and Technology, Norway

The assignment problem is a fundamental optimization problem and a crucial part of many systems. For example, in multiple object tracking, the assignment problem is used to associate object detections with hypothetical target tracks and solving the assignment problem is one of the most compute-intensive tasks. To enable low-latency real-time implementations, efficient solutions to the assignment problem is required. In this work, we present Sparse and Speculative (SaS) Auction, a novel implementation of the popular Auction algorithm for FPGAs. Two novel optimizations are proposed. First, the pipeline width and depth are reduced by exploiting sparsity in the input problems. Second, dependency speculation is employed to enable a fully pipelined design and increase the throughput. Speedups as high as 50× are achieved relative to the state-of-the-art implementation for some input distributions. We evaluate the implementation both on randomly generated datasets and realistic datasets from multiple object tracking.

CCS Concepts: • **Computer systems organization → Embedded systems;**

Additional Key Words and Phrases: Assignment problem, Auction method, FPGA, object tracking

ACM Reference format:
Erling Jellum, Milica Orlandić, Edmund Brekke, Tor Johansen, and Torleiv Bryne. 2022. Solving Sparse Assignment Problems on FPGAs. *ACM Trans. Archit. Code Optim.* 19, 4, Article 55 (December 2022), 20 pages.
https://doi.org/10.1145/3546072

1 INTRODUCTION

The assignment problem is a special type of a linear programming problem, where an optimal one-to-one assignment between a set of *objects* and a set of *agents* is sought. The assignment problem is widely used, for instance, to model network traffic and optimizing routes [10], alignment of protein-protein interaction networks in bioinformatics [11], dynamic task allocation for robots [9, 34, 35], and computer vision tasks [16, 25]. It is formulated as maximum weight matching for bipartite graphs. The assignment problem is of particular importance in **multiple object tracking (MOT)** [8], where, at the data association stage, each object detection is to be associated with a single track. MOT is an NP-hard problem and the assignment problem is the main computational bottleneck [3, 14]. MOT is one of the enabling capabilities for autonomous robotics systems [24, 27] and efficient and scalable solutions are needed.

The Auction algorithm is an effective iterative solution to the assignment problem [5]. There exist multiple optimized implementations of the Auction algorithm for CPUs, GPUs [32], and

Authors' address: E. Jellum, M. Orlandić, E. Brekke, T. Johansen, and T. Bryne, Norwegian University of Science and Technology, Trondheim, Norway, 7030; emails: {erling.r.jellum, milica.orlandic, edmund.brekke, tor.arne.johansen, torleiv.h.bryne}@ntnu.no.

1544-3566/2022/12-ART55
https://doi.org/10.1145/3546072

FPGAs [17, 18, 36]. In Reference [36], an FPGA implementation is proposed that claims 10× performance increase relative to a baseline CPU implementation, whereas a 7× performance increase is reported for the GPU implementation in Reference [32]. However, the proposed implementations fail to address two critical properties of the assignment problems found in, e.g., MOT:

(1) The problems show a high degree of **sparsity**,
(2) The assumption of **data dependencies** between consecutive iterations is often false.

In this article, we propose **Sparse and Speculative (SaS)** Auction, an accelerator generator for the auction algorithm targeting **System-on-Chip (SoC)** FPGAs. It takes advantage of the sparsity often found in the problems, which both reduces the area and depth of the processing pipeline. We also propose a data dependency speculation scheme that reduces the number of pipeline bubbles and improves the throughput. The implementation is evaluated on a SoC FPGA, and the source code is available on Github [20].

The rest of the article is organized as follows: In Section 2, a brief background into the assignment problem, the Auction algorithm, and its application in MOT is given. In Section 3, related work is reviewed. In Section 4, the proposed implementation of the SaS Auction is introduced. In Section 5, the evaluation and results are discussed. The article is concluded in Section 6.

2 BACKGROUND

2.1 The Assignment Problem

The goal of the assignment problem is to find an assignment between a set of objects O and a set of agents A that maximizes a reward function. The assignment problem is also referred to as maximum weight matching of weighted bipartite graphs and is solvable in polynomial time. The assignment problem is characterized by a reward matrix R of size $n \times m$ where r_{ij} is the reward for assigning object j to agent i. The solution to the assignment problem is then finding an optimal assignment matrix B where matrix element $b_{ij} = 1$ if the edge from object j to agent i is picked, and zero otherwise, i.e., b_{ij} is binary.

This optimization problem can be expressed as a linear program as follows:

$$\max \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} r_{ij} \cdot b_{ij} \tag{1}$$

$$\text{subject to} \sum_{i=0}^{m-1} b_{ij} \leq 1, \forall j = 0, 1, \ldots, n-1 \tag{2}$$

$$\sum_{j=0}^{n-1} b_{ij} = 1, \forall i = 0, 1, \ldots, m-1. \tag{3}$$

Consider the simple assignment problem shown in Table 1. Here, the set of agents is $A = \{Alice, Bob, Charlie\}$ and the set of objects are $O = \{o_1, o_2, o_3\}$. The value r_{ij} represents the reward associated with assigning agent a_i to object o_j.

2.2 Methods for the Assignment Problem

2.2.1 Simplex Methods. The assignment problem can be solved with well-studied algorithms from linear programming like the simplex method [33]. The simplex method starts by observing that the solution space of a linear programming problem is a polytope defined by the constraints of the linear program. Since the cost function is linear it can be shown that the optimum must lie on a vertex of this polytope. For the assignment problem, each possible mapping of agents to

Table 1. A Simple Assignment Problem

	Object		
Agent	o_1	o_2	o_3
Alice	5	10	1
Bob	4	11	3
Charlie	5	14	8

objects make up a vertex in a $(n+m)$-dimensional polytope, where n is the number of agents and m is the number of objects. The Simplex method starts by picking an initial vertex. It then traverses the edges, from vertex to vertex, until the optimum is found. In the worst case, the algorithm must visit all vertices, giving it a computational complexity equivalent of a brute force method with $O(2^N)$.

The assignment problem belongs to a special class of linear programming problems known as transportation problems. Transportation problems can be represented as graphs where the decision variables are the weights of the edges. Each decision variable only occurs twice in the constraints, once for the source node and once for the destination node. This additional structure is taken advantage in, e.g., the Streamlined Simplex [22].

In recent years, enterprise optimization software such as Gurobi [15] and CPLEX [6] have gained popularity. Here, the assignment problem is solved as a linear integer programming problem using Simplex-like algorithms.

2.2.2 The Hungarian Method. The Hungarian method was introduced by Khun as a more efficient solution to the assignment problem than the traditional linear programming methods. In the Hungarian method, the assignment problem is modeled as a graph $G = (V, E)$ and a reward matrix R, where $V = A \cup O$ is the set of vertices, $E = \{(a_i, o_j)\}_{a_i \in A, o_j \in O}$ is the set of edges, and r_{ij} is the weight of edge (a_i, o_j). A matching M is a subset of E such that no edges in M share a common vertex. We call M a perfect matching if it matches all the vertices of V. Given a graph G and a matching M, an augmenting path is a path in G that starts and ends at an unmatched vertex and contains alternating edges from M and $E - M$. Augmenting paths can be used to iteratively find perfect matchings.

The assignment problem can thus be reformulated as finding the perfect matching with the maximum weight. Compared to the general class of transportation problems, the assignment problem has two additional features. First, the weights are binary, i.e., either 1 or 0. Second, if a particular edge between agent a_i and object o_j has the weight 1, then all other edges connected to both a_i and o_j has the weight 0. This information is not utilized by the linear programming methods and makes them comparatively slow [5]. The Hungarian method solves the assignment problem in polynomial with a computational complexity of $O(N^3)$. The Hungarian method is a primal-dual algorithm where the starting point is a feasible solution. It works by maintaining and iteratively updating two data structures. The first is a labeling l of G. A labeling is a function mapping the set of vertices to the real numbers. A valid labeling is defined as one where the the sum of the labels for two vertices is less than or equal to the weight of the edge connecting them. The second is a matching M. It can be shown that given a labeling l and the equality subgraph G_l, then a perfect matching M of G_l is also the maximum weight matching of G. The Hungarian method starts with a matching M and a valid labeling l. It then augments the matching until either it finds a perfect matching or no augmenting path exists. In the latter case, it improves the labeling function l before going back to looking for an augmenting path.

The Hungarian method is not suitable for parallelization as both the labeling improvement and finding an augmenting path are inherently sequential operations.

2.2.3 LAPJV. The LAPJV algorithm was introduced by Jonker and Volgenant [21] as an improvement of the Hungarian method. While the Hungarian method works by finding any feasible augmenting path, LAPJV looks for the shortest augmenting path. This is found using Dijkstra's shortest path algorithm.

2.2.4 The Auction Algorithm. The Auction algorithm was proposed by Bertsekas [4] as a general solution to the assignment problem. The assignment problem is modeled as an auction and the reward r_{ij} is interpreted as how much agent a_i is willing to pay for the object o_j.

The Auction method is not a primal-dual algorithm like the Hungarian method, and each iteration is not guaranteed to reduce the total cost. Optimality can be guaranteed if ϵ, the minimum raise, is less than a threshold. The algorithm is guaranteed to terminate if there exists a feasible solution. The relaxed constraint on monotonically improving solutions enables the Auction method to reach the optimum in fewer steps. Its computational complexity is $O(N^2 log(N))$ [29].

The Auction algorithm works by updating a price vector P, which contains the current price for each object, and an assignment vector that stores information about which agents are assigned to which objects. The price vector is initialized to 0 and is increased as agents bid for objects. Each iteration consists of two phases: the bidding phase and the assignment phase. In the bidding phase, a subset of the agents $A_{bid} \subseteq A$ calculate bids for a subset of the objects $O_{bid} \subseteq O$. In the assignment phase, the highest bid is picked and the bidding agent is assigned to the object. There are two commonly used approaches to the bidding phase. The first alternative is referred to as the Jacobi variant. Here, all unassigned agents bid for a single unassigned object; i.e., $A_{bid} = A_{unassigned}$ and $O_{bid} = o_k$, where $A_{unassigned}$ is the set of unassigned agents and o_k is a single unassigned object. Referring to the example in Table 1, the first iteration of the Jacobi method would consist of Alice, Bob, and Charlie bidding of object o_1. The second alternative is referred to as the Gauss-Seidel variant. Here, a single unassigned agent bids for all the unassigned objects. This means that $A_{bid} = a_k$ and $O_{bid} = O$. Here, a_k is a single unassigned agent. The first iteration of the Gauss-Seidel variant would consist of Alice bidding for objects o_1, o_2, o_3. The Jacobi variant can be implemented by assigning a processing element to each *agent* to compute the bids in parallel. The disadvantage of this approach is that as more agents are assigned, fewer of the processing elements are used. The Gauss-Seidel variant can be implemented by assigning a processing element to each *object*. It does not suffer from the same disadvantage as the Jacobi method, as the number of objects being bid on remains constant as the algorithm proceeds. For the rest of this work, the Gauss-Seidel variant is used.

The winning bid each round is the bid for the object that has the highest benefit to the agent. The benefit of object j for agent i is the difference between the agents' reward $R(i, j)$ for the object o_j and the current price $P(j)$, computed as:

$$b(i, j) = R(i, j) - P(j). \tag{4}$$

When the object with the highest benefit is found, the agent decides its bid for that object. The bid is made such that the object still has the highest benefit to the agent. This is achieved by raising the bid for the object with an amount equal to the difference between the highest and the second-highest benefit, expressed as:

$$\text{raise} = b_{\max} - b_{\max-1}. \tag{5}$$

The pseudo-code for the Auction algorithm is presented in Figure 1. Lines 1–3 initialize the arrays containing object prices and assignments and the queue containing the unassigned agents.

```
1:  prices[0 → N] ← 0
2:  objectAssignments[0 → N] ← invalid
3:  unassigned.enqueue(0 → N)
4:  while unassigned not empty do
5:     currAgent ← unassigned.dequeue()
6:     benefit ← reward[currAgent] − prices
7:     object ← benefit.indexOfMax()
8:     highest ← benefit.sorted().take(2)
9:     raise ← highest[0] − highest[1]
10:    if raise < 0 then
11:       continue
12:    end if
13:    bid ← price[object] + raise
14:    if objectAssignments[object] then
15:       oldAgent ← objectAssignments[object]
16:       unassigned.enq(oldAgent)
17:    end if
18:    objectAssignments[object] ← currAgent
19:    prices[object] ← bid
20: end while
```

Fig. 1. Auction algorithm, Gauss-Seidel variant.

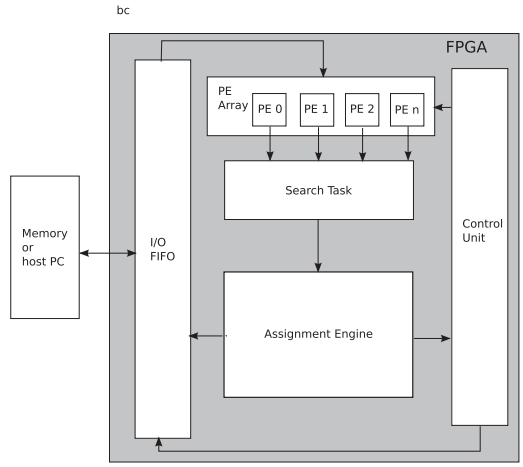


Fig. 2. Architecture of the accelerator of Zhu et al. adapted from Reference [36].

Inside the loop, an unassigned agent is dequeued and agents' benefits for each object are computed. Line 7 extracts the index of the object with the highest benefit, while Line 8 gets the highest and second-highest benefit. Line 9 calculates the raise according to Equation (5). Lines 14–17 unassign any previously assigned agent. This proceeds until agents are either assigned or disregarded due to not being able to create a positive raise (lines 10–13). An optimized C++ implementation is published in Reference [19].

2.2.5 Execution of the Auction Algorithm. Figure 3 shows the execution of the four iterations needed to solve the assignment problem in Table 1 using the Auction algorithm.

(1) In the first iteration, *Alice* is bidding. Since the prices are all zero, the object benefits are equal to the object rewards. The most beneficial is o_2, marked with blue, and the second most beneficial object is o_1, marked with green. The *raise* is the difference between the most and the second-most beneficial object, which is 5. After iteration 1 *Alice* is assigned to o_2 and the price is set to 5.
(2) In the second iteration, Bob is bidding. His benefits are calculated using his rewards and the updated prices. His most beneficial object is o_2 and his raise is 2. Alice is thus unassigned from o_2 and replaced by Bob. The price is raised to 7.
(3) In the third iteration, Charlie is bidding. His most beneficial object is o_3, which has 1 unit more benefit than o_2. He thus bids 1 for o_3 and is assigned to it.
(4) In the fourth and last iteration, Alice is bidding again. With the updated prices, o_1 is now the most beneficial object, 2 units more beneficial than o_2. She bids 2 for o_1 and is assigned to it. After iteration 4 there are no more unassigned agents and the algorithm terminates.

2.3 The Assignment Problem in MOT

In MOT, the goal is to estimate the position and velocity of multiple targets, or objects, based on sensor measurements from a single or multiple sensors. Probabilistic models are used to account for clutter (false positives) and misdetections (false negatives) in the sensor measurements. A type

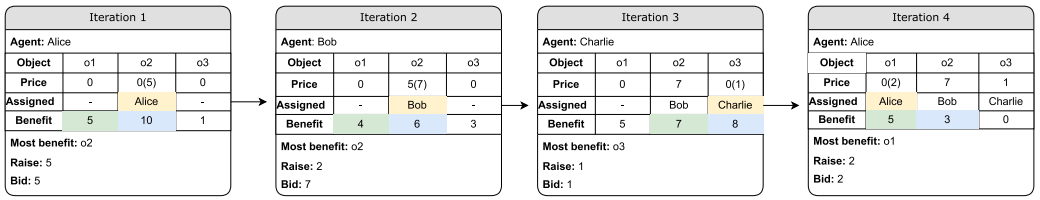


Fig. 3. Execution of the Gauss-Seidel auction algorithm.

of Bayesian filter is usually used to predict *a priori* state estimates and combine them with sensor data into *a posteriori* state estimates [13].

Targets that are sufficiently close to each other are clustered together. In this manner, the complexity is lowered, as the algorithm can be run independently on each cluster. A possible association between a measurement and a target is called a single-target hypothesis. The measurements within the range of a cluster will result in data association hypotheses for all the targets in the cluster. Finding the optimal data association hypothesis is modeled as an assignment problem. The number of hypotheses in a cluster can reach hundreds or even thousands. However, it is common that relatively few hypotheses contains the majority of the probability density mass. Validation gating is a common technique where hypotheses with probability below a certain threshold are set to zero. This can yield large and highly sparse assignment problems. In the **Multiple Hypothesis Tracker (MHT)** [8], a set of hypotheses are maintained over consecutive iterations. For each new set of measurements, the hypotheses give birth to child hypotheses that inherit the probability properties from their parents. To avoid an explosion of hypotheses, pruning is applied to remove the most unlikely candidates.

In recent years, a generalization of MHT known as the **Poisson Multi-Bernoulli Mixture (PMBM)** filter [12] has been proposed as a state-of-the-art approach to MOT. We evaluate SaS Auction on a set of assignment problems generated by a PMBM tracker. It is referred to as the MOT dataset.

3 RELATED WORK

In Reference [32], Vascencelos et al. propose accelerating the Auction algorithm on a GPU claiming a 7× speedup relative to a baseline CPU implementation. The algorithm is divided into a bidding phase and an assignment phase. In the bidding phase, all agents bid on an object in parallel, whereas all objects find a valid bid and update their price in the assignment phase. While achieving good performance for the big dense problems, the GPU is characterized by having high energy consumption, making it unfit for many embedded applications.

In References [17, 18] Hung et al. propose an FPGA accelerator for solving the assignment problem. The architecture is based on a recurrent neural network, but the performance scales poorly to big problems, which can take multiple seconds to solve.

In Reference [36], Zhu et al. propose a hardware architecture for Auction algorithm accelerators on FPGAs. They claim a 10× speedup relative to a baseline CPU implementation. A simplified overview of the architecture is shown in Figure 2. It consists of a pipeline with a configurable number of **processing elements (PEs)**. In each iteration, object-rewards for a single object are read from memory or the host PC through the I/O FIFO. The rewards are distributed among the PEs in the PE Array. Each PE contains a small LUTRAM where the object prices related to that PE are stored. For an architecture with N PEs capable of solving problems with up to M objects PE-k will store prices $P_{k+i\cdot N}$ for $k = 1 \ldots M/N$. The resulting benefits are passed to the SearchTask module, which finds the highest and second-highest benefit. The SearchTask is implemented as a tree of

comparators and its depth scales with $\log_2(N)$. The two highest benefits are further passed to the AssignmentEngine, which computes the raise. The AssignmentEngine contains the assignment array and unassigns the previous agent holding the highest bid for the object in question. The raise is then forwarded from AssignmentEngine to the ControlUnit, which updates the price within the right PE. The I/O FIFO receives information of unassigned agents such that it can request its object rewards from memory. In the rest of this article, we refer to the FPGA implementation by Zhu et al. as **state-of-the-art (SoA)** Auction.

4 SPARSE AND SPECULATIVE (SAS) AUCTION

SaS Auction is a flexible FPGA accelerator that efficiently solves the assignment problem. There are multiple reasons for using an FPGA rather than a CPU or GPU for solving the assignment problem. Due to its ability to implement arbitrary datapaths, it has greater potential for taking advantage of sparsity. This can yield designs with lower latency and lower energy consumption. In robotic applications embedding target tracking the Auction algorithm will typically be on the critical path. Reducing the latency of the algorithm can thus allow the robot to navigate faster, Also, for mobile battery-driven autonomous robots, energy efficiency is paramount. FPGAs can achieve much higher energy efficiency than CPUs and GPUs [26]. Last, for some autonomous robots, like drones, the weight of the computational platform is an important parameter. In some cases, the weight can be the limiting factor for performance [23]. Due to reduced power consumption, using an FPGA could remove the need for a heavy heatsink.

In the following, the term *agent* is used to refer to the row of object-rewards associated with the agent. We use *agent-index* to refer to the index, or ID, of the agent.

4.1 Sparse Problems

An assignment problem is sparse if the majority of the elements in the reward matrix R are zero. If $r_{ij} = 0$, then object-j is never assigned to agent-i. We also refer to these as non-valid object rewards. Figure 4 shows the level of sparsity found in the assignment problems in the MOT dataset. The x-axis shows the density, which is the inverse of the sparsity. Clearly, these are highly sparse assignment problems. The vast majority of problems have less than 5% density.

Figure 5 shows how the depth of the SearchTask scales with the number of PEs. Each level of the search tree is implemented in a separate pipeline stage. Over-dimensioning the accelerator by using too many PEs has a negative performance impact. The throughput is affected by the depth of the SearchTask, as pipeline stalling takes place at the PE stage until the preceding agent has finished executing in the SearchTask. The number of PEs also impacts the area and the obtainable clock rate of the accelerator.

However, under-dimensioning the accelerator with too few PEs is also costly and reduces the throughput, since it makes each agent span multiple pipeline stages.

Generally, the optimal number of PEs for a particular problem is the number of object rewards per agent. By taking advantage of sparsity, the optimal number of PEs can be greatly reduced and thus improving throughput and latency.

4.1.1 Representing Sparsity. There are multiple ways of representing sparse matrices [30]. A simple tuple representation [28] was chosen, where the element and the column index are stored as a tuple. The original reward matrix R resides in the off-chip DRAM and each element of the reward matrix is represented by a 64-bit data word. The transformed vector C is the sparse representation of R and resides in the on-chip BRAM. For architecture with n PEs, each BRAM data word in C contains n tuples of object reward and column index as given:

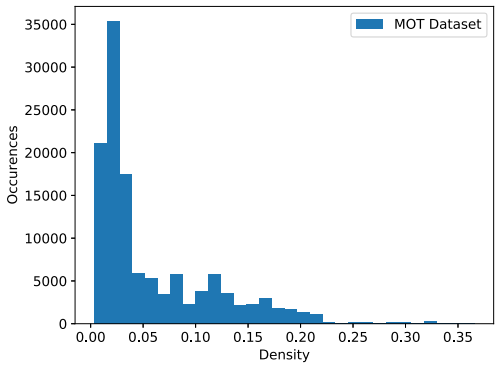


Fig. 4. Sparsity in MOT simulations.

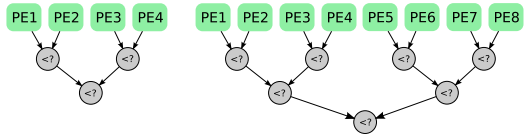


Fig. 5. SearchTask depth for 4 and 8 PEs.

$$c_i = (e_1, e_2, \ldots, e_n),$$
$$e_j = (\text{column}, \text{value}).$$

The number of bits used to represent a single object reward is configurable. The words of the transformed BRAM vector C are padded such that the first non-zero element of each row of the original DRAM matrix R always occupies the first element e_1 of a BRAM word c. This means that each BRAM word only contains rewards from the same row of the original matrix. A row of the original matrix may span several words of the transformed vector C. A separate array is used to map the row indices of the original matrix to indices of the transformed matrix.

In Figure 6, an example of a transformation between a problem matrix R and its sparse representation C is given.

$$R = \begin{bmatrix} 0 & 0 & 1 & 2 & 0 & 0 & 0 & 7 \\ 0 & 0 & 0 & 0 & 0 & 4 & 0 & 8 \\ 6 & 0 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 4 & 0 & 2 & 9 \\ 0 & 8 & 4 & 0 & 2 & 6 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 7 & 0 \end{bmatrix}$$

(a)

$$C^T = \begin{bmatrix} (0,0),(5,1),(4,2),(0,7) \\ (0,0),(0,0),(2,4),(0,8) \\ (0,0),(0,0),(7,6),(1,2) \\ (0,0),(3,4),(1,2),(0,9) \\ (0,0),(0,0),(0,0),(6,8) \\ (5,4),(3,2),(2,6),(1,5) \\ (0,0),(0,0),(0,0),(1,7) \end{bmatrix}$$

(b)

Fig. 6. (a) Original reward matrix. (b) Transformed reward vector.

An architecture with 4 PEs is assumed for simplicity. This means that each BRAM word stores up to four valid, non-zero object rewards and their column indices. The reward matrix R is sparse with the majority of the rewards being zero. The bottom right corner has indices (0,0). Each element c_i in the transformed vector C consists of four tuples $e_1 \ldots e_4$. The individual tuples consist of the element value and its column index in the original matrix R. The transformed vector C is not fully dense, because the first element of each *row* in R is aligned to the beginning of a data *word* in C. This is done to efficiently read out an entire row of the original input matrix by reading a single BRAM data word. In the example, elements 1 and 2 of C (zero-indexed starting from the bottom) both belong to row 1 of R. When the number of valid elements in a row in R exceeds the number of PEs in our architecture the elements are spread out over multiple words. The row index of each element in C is stored in an array that maps each row of R to an element in C.

$$RowIndex = \begin{bmatrix} 0 & 1 & 3 & 4 & 5 & 6 \end{bmatrix}.$$

4.1.2 Cost of Sparsity-aware Processing. There are three major costs associated with using a sparse representation. First, the translation from dense to sparse representation must be performed. In the SaS Auction architecture, the translation is done in hardware, and this incurs an area cost as opposed to the time cost associated with a software implementation. Second, a sparse representation removes the static relationship between PE and object price. In SoA Auction, PE0 always calculates the benefit of object 0 for the various agents. This means that each PE can store the prices locally and thus can all the prices be accessed in parallel at low cost. To achieve parallel lookup of all PEs, SaS Auction duplicates the BRAM with object prices for each PE. Last, sparse representation adds overhead, because two values are needed for each element: the column index and the element value. The memory needed to store a sparse matrix is given by:

$$Mem_{sparse} = w_{reward} \cdot log_2(n_{max}) \cdot r_{sparse}, \tag{6}$$

where w_{reward} is the bits needed to represent a reward, n_{max} is the maximum problem size, and r_{sparse} is the rate of non-zero elements. The memory requirements for a dense representation is as follows:

$$Mem_{dense} = w_{reward} \cdot n_{max}^2. \tag{7}$$

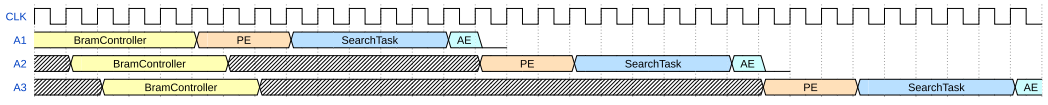
For an FPGA implementation, both Mem_{sparse}, n_{max} and w_{reward} are fixed at compile-time. For a given combination of these parameters, there is an upper bound on the size and density of a problem matrix that can be stored.

4.2 Dependency Speculation

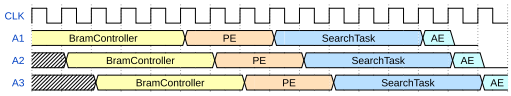
Each iteration of the Auction algorithm consists of reading the object rewards for a specific agent and comparing them to the current object prices. An iteration may result in a bid and thus change the state of the object prices. This means that unless the processing pipeline is stalled until the current agent is done, the next agent might use stale prices when calculating its bid. We define an **unresolved agent** as an agent that is currently computing a bid but has yet to commit it by updating the prices. There is a data dependency between agents a_1 and a_2 if the output of processing a_1 is dependent on whether it executes before or after a_2. Dependencies are dynamic, i.e., two agents might be dependent at some point during the processing and independent at another. Dependencies can be divided into two categories. A **strong dependency** occurs if both a_1 and a_2 have the same most-beneficial object. In this case, a_1 calculates an illegal bid if it does not wait for a_2 to commit its bid first. A **weak dependency** occurs when a_2 bids for the second-most beneficial object to a_1. In this case, the order does not change which object a_1 bids for, but it affects the size of the bid. It makes a_1 bid more conservative, which is equivalent to making a smaller step toward the global optimum. In Reference [36], the dependencies are handled by stalling all agents in the PE stage until one agent computes its bid introducing expensive pipeline bubbles.

The processing stages for agent a_i through the pipeline of SoA Auction with 8 PEs are shown in Figure 7(a). The components will be described in more detail in Section 4.3. The pipeline is stalled such that only one agent traverses the loop consisting of PEs, SearchTask, and AssignmentEngine stages at the time. In the proposed SaS Auction, a basic static dependency speculation scheme is employed. It is always speculated that there are no strong dependencies between the agent currently at the PE stage and the unresolved agents that have not committed their bids yet. In the cases where there actually is a strong dependency, the agents bid is disregarded and it is added back in the unassigned queue. This is detected by comparing the bid to the current price. If it is lower, then a misspeculation has occurred.

Figure 7(b) shows a timing diagram for SaS Auction, an implementation supporting dependency speculation, characterized by no pipeline bubbles, which provides 9× increase in throughput compared to SoA auction implementation in Reference [36]. The actual speedup is limited by the rate



(a) SoA Auction



(b) SaS Auction

Fig. 7. Pipeline timing diagrams. AE is short for AssignmentEngine.

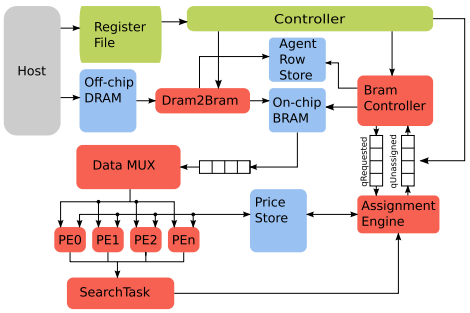


Fig. 8. High-level architecture of SaS auction.

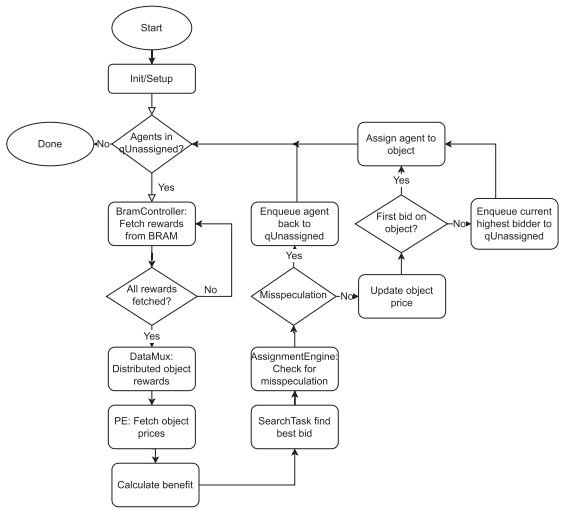


Fig. 9. Flowchart of SaS auction.

of misspeculations and pipeline bubbles naturally occurring, because there are few unassigned agents left.

4.3 Hardware Architecture

The proposed SaS Auction is an optimized accelerator generator for efficiently solving the assignment problem. It is specifically optimized for the sparse problems found in, e.g., MOT.

Figure 8 shows an overview over the SaS Auction hardware architecture, and Figure 9 shows the speculative Auction algorithm represented as a flow-chart. A streaming dataflow architecture optimized for throughput is proposed. It is inspired by SoA Auction by Zhu et al. The accelerator is characterized by n_{PE}, its number of PEs. This decides the width of the pipeline and BRAM interfaces. The accelerator reads the problem matrix from the off-chip DRAM into the on-chip BRAM. It also initializes the queue of unassigned agents. Then, it proceeds to perform the steps in the algorithm in Figure 1. The accelerator reads one row of the problem matrix, finds the most beneficial object, and calculates the bid. Then, it updates the price and assignment vector. This continues until there are no unassigned agents left. In the following, each component is described briefly.

(1) Register File provides the user interface of the accelerator. It is conventional AXI4-Lite-based memory mapped control and status register file. The user software writes the memory

address of the input problem matrix along with its dimensions into registers residing at predefined addresses. Starting and stopping the accelerator is also achieved through the register file.

(2) Dram2Bram functions as a DMA that reads problem data from the off-chip DRAM, compresses it to the proposed sparse representation, and writes it to the on-chip BRAM. The parsing and compressing is pipelined in two stages.

(1) Non-valid, i.e., rewards that are zero, are filtered out. The valid rewards are entered into an register-based array, $ValidR$, corresponding to a single BRAM word.
(2) When either the array is full or a complete row of the input problem has been read, the array is written to the on-chip BRAM, and the address at which it is stored is written to the AgentRowStore

The length of $ValidR$, and consequently the width of the on-chip BRAM interface, is equal to n_{PE}. This allows the rewards for all the PEs to be read out in a single cycle and.

(3) qUnassignedAgents and **qRequestedAgents** are **first-in-first-out (FIFO)** queues between the AssignmentEngine and the BramController. They hold the unassigned agents and the currently executing agents, respectively. BramContoller dequeues agent-indices from the qUnassignedAgents and enqueues them back on the qRequestedAgents after their rewards have been requested from the BRAM. When a bid arrives from the SearchTask, the AssignmentEngine dequeues from the qRequestedAgents and will potentially enqueue a new agent-index on the qUnassignedQueue.

(4) AgentRowStore is a BRAM storing the mapping of the row index of the original input matrix to index of the on-chip BRAM data word containing the transformed version of that row. The Dram2Bram module writes to the AgentRowStore as it is parsing and compressing the input matrix. The AgentRowStore is read by the BramController before fetching rewards in the on-chip BRAM.

(5) BramController controls the requesting and the parsing of the rewards from the BRAM. It is pipelined over multiple stages.

(1) Dequeue index of unassigned agent from qUnassigned
(2–3) Read the BRAM address of the agent from AgentRowStore
(4–5) REad rewards from BRAM and forward to DataMux

(6) PEs perform the actual computation as expressed in Equation (4). The PE stage is pipelined with the following stages:

(1) Read current price of object from PriceStore
(2) Calculate benefit of object
(3) Forward benefit to SearchTask.

The PE performs static dependency speculation and always assumes that there are no dependencies between current agent and any unresolved agent. Any violation, which is caused by a strong dependency, will be caught in the AssignmentEngine.

(7) PriceStore is the BRAM-based centralized storage for the object prices. Each PE has a dedicated BRAM where all the prices are stored, i.e., the prices are duplicated for each PE. The write ports of all the duplicate BRAMs are connected to the AssignmentEngine, which updates all the price copies after a bid. Each duplicate BRAM has a single read port that is connected to its PE. The AssignmentEngine also has a duplicate BRAM, which is used resolve the dependency speculation.

(8) SearchTask is illustrated in Figure 5 and is a tree of comparators that finds the highest and second-highest benefits, among the benefits outputted by the PE stage. Each level of the tree is implemented as a pipeline stage. The SearchTask can perform multiple iterations to support solving problems where the agents span multiple pipeline stages. When the highest and second-highest benefits are found, the SearchTask computes the bid as expressed in Equation (5).

(9) AssignmentEngine is in charge of maintaining the prices, assignments, and detecting and handling misspeculations. The AssignmentEngine is pipelined with the following stages:

(1) Receive bid b for object o_j from the SearchTask and the corresponding agent-idx a_i from qRequestedAgents. Also read current price of the object $P(o_j)$ from the PriceStore. And the currently assigned agent $objectAssignments(o_j)$.
(2) Resolve speculation by comparing $P(o_j)$ with b
(3a) If $b <= P(o_j)$, then there has been a misspeculation caused by a strong dependency between a_j and a previous agent. In that case, a_i is enqueued back to qUnassignedAgents.
(3b) If $b > P(o_j)$, then both the PriceStore and ObjectAssignments are updated. The previously assigned agent, which was requested in the first stage, is enqueued to qUnassignedAgents.

Weak dependencies are not detected or handled by the AssignmentEngine.

(10) Controller performs control tasks. It fills up the qUnassignedAgents FIFO when the accelerator is started, detects the end of accelerator's operation, and initiates writing the result back to DRAM.

Figure 9 shows a flowchart model of the pipeline. It is focused on the core processing pipeline, which starts and ends with the qUnassigned. The interaction with off-chip DRAM, host software, and transformation of input problems to sparse representations are abstracted away in the Init/Setup process.

4.4 Comparison to SoA Auction

The key differences between the SaS and the SoA Auction are as follows:

4.4.1 Sparsity. SoA Auction contains a simple memory interface that either can read the rewards from an off-chip main memory or an on chip BRAM. The latter necessitates that the SW has transferred the data to BRAM first. SaS Auction implements a DMA in Dram2Bram, which moves the data from off-chip DRAM to on-chip BRAM and compresses the data on the fly.

4.4.2 Price Storage. With sparsity-aware processing, the static mapping of object price to PE no longer exists. A object price might be needed by any PE. To enable single-cycle price-lookup for all PEs, SaS Auction stores a copy of the prices for each PE.

4.4.3 Speculation. SaS Auction implements speculative processing. While SoA Auction stalls execution in the PE stage, SaS Auction continues with possibly stale prices. The AssignmentEngine will detect a strong data dependency and reschedule the agent that used stale prices.

4.5 Performance Model for Speculation

In the following, a performance model for speculative execution in SaS Auction is derived.

4.5.1 Stalling Cost. First, a cost model for the state-of-the-art implementation is derived. The cost of stalling execution at the PE stage depends on the number of clock cycles needed by the preceding agent to calculate and commit its bid. It can be expressed as:

$$\begin{aligned} C_{\text{stall}} &= d_{\text{PE}} + d_{\text{SearchTask}} + d_{\text{AssignmentEngine}} \\ C_{\text{stall}} &= 4 + log_2(n_{\text{PE}}), \end{aligned} \tag{8}$$

where d_{PE}, $d_{\text{SearchTask}}$ and $d_{\text{AssignmentEngine}}$ refer to the latency (or depth) of the respective modules, given in Table 2. Note that these values are based on our implementation inspired by Zhu et al. [36], where no data regarding the latencies of the original implementation is available.

Clearly, the stalling cost is independent of the input problem dimension and only depends on the number of PEs. However, the total processing latency of one agent also depends on the size of

Table 2. Pipeline Latencies

Module			
PE	Search Task	Assignment Engine	Bram Controller
3	$2+\log_2(n_{\text{PE}})$	1	5

input problem dimension relative to the number of PEs. Thus, the *relative* cost of stalling decreases the more underdimensioned the accelerator is.

4.5.2 Speculation Cost. The speculation cost is defined as the additional number of clock cycles needed to process an agent if it is not stalled at the PE stage, but rather being speculatively processed. This cost is a stochastic variable, as it depends on whether there is a data dependency between the agent and any unresolved agents. The goal is thus to find the *expected* value of the speculative cost. The misspeculation cost is first derived. It is the penalty of dispatching an agent from the PE stage if there is an unresolved strong dependency and is defined as:

$$C_{\text{miss}} = 1 + n_{\text{bubbles}} + C_{\text{stall}}. \tag{9}$$

The constant cost of C_{stall} is the worst-case expected cost of redoing the loop consisting of PE, SearchTask, and AssignmentEngine stages once the agent is ready in the PE-stage again. The variable n_{bubbles} is the number of pipeline bubbles occurring between the misspeculation and the next instant the agent is ready in the PE-stage. It is estimated as:

$$n_{\text{bubbles}} = \max(0, d_{\text{PE}} - n_{\text{unassigned}}). \tag{10}$$

If $n_{\text{unassigned}}$ is larger than the number of clock cycles from the AssignmentEngine to the PE stage, then there are no bubbles. The expected costs of stalling and speculating are expressed as:

$$\begin{aligned} E(\text{stall}) &= C_{\text{stall}} \\ E(\text{speculate}) &= p_{\text{miss}} \cdot C_{\text{miss}}, \end{aligned} \tag{11}$$

where p_{miss} is the probability of a misspeculation, i.e., the probability that the agent at the PE stage has a strong dependency on an unresolved agent.

4.5.3 Probability of Misspeculation. To estimate the probability of a strong data dependencies, p_{miss}, some simplifications must be made. It is assumed that the most beneficial object o_{pick} for an agent a_k is an independent random variable drawn from a uniform distribution. This can be expressed as follows:

$$p(o_{pick} = o_i) = \frac{1}{m}, \tag{12}$$

where m is the number of objects.

The probability of a strong dependency can thus be expressed as the probability of an agent picking the same object as any of the unresolved agents.

$$p_{\text{miss}}(n) = 1 - \left(\frac{m-1}{m}\right)^n, \tag{13}$$

where n is the number of unresolved agents and m is the number of objects. The actual number of unresolved agents n depends several factors.

First, it has an upper bound defined by the number of PEs in the accelerator. The number of PEs decides the number of pipeline stages between reading the prices in the PE stage and committing

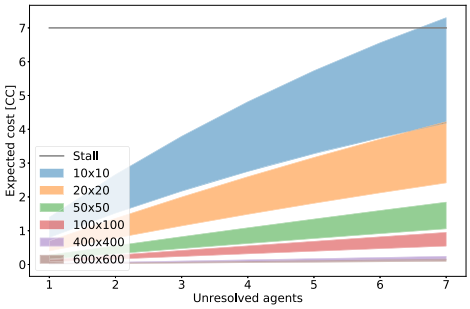


Fig. 10. Expected cost of speculating and stalling for different problem sizes.

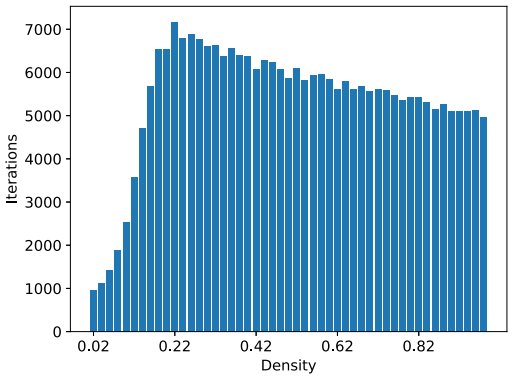


Fig. 11. Auction algorithm iterations for random sparse 50 × 50 problems.

a bid in the AssignmentEngine. In an architecture with 8 PEs, there are 7 pipeline stages between PE and the commit operation.

Second, it also depends on how many pipeline stages each agent spans. In a scenario with 8 PEs and agents with 64 valid object rewards, the agent will span 8 pipeline stages. This means that when a new agent arrives at the PE stage, there will be at most one unresolved agent that would span all the 7 pipeline stages between the PE stage and the commit stage. Consider another scenario, also with 8 PEs, but this time with agents with at most 8 valid object rewards. This implies that each agent only spans a single pipeline stage. In this scenario, when a new agent arrives at the PE stage, there might be as many as 7 unresolved agents.

Last, it also depends on how many unassigned agents are left. When the agent of the previous scenario arrives at the PE stage. The number of unresolved agents could be anywhere from 0–7. The actual number depends on how full the pipeline is, which depends on how many unassigned agents are left.

Finally, using the derived probability of misspeculation p_{miss}, Equation (11) is plotted in Figure 10 for an accelerator with 8 PEs with different input problem sizes. The grey horizontal line represents the cost of stalling, which is constant and independent of the number of unresolved agents and the number of potential pipeline bubbles. The expected cost of speculating depends on both $n_{\text{unresolved}}$ and the potential number of bubbles n_{bubbles}. Thus, Figure 10 shows the *range* of expected cost of speculating. The maximum expected cost occurs in the case of an empty pipeline *behind* the agent in the PE stage. The minimum cost arises when the pipeline is full and there are enough unassigned agents such that $n_{\text{bubbles}} = 0$.

Clearly, for bigger problems the probability of misspeculation is very small and thus also the expected cost of speculation. Intuitively, increasing number of unresolved agents leads to increased chance of misspeculation and thus increased expected cost. Given the assumption of uniform probability distribution, a static speculation scheme is sufficient. In static speculation, as opposed to dynamic speculation, the pipeline always proceeds in the PE stage, regardless of the number of unresolved agents or the number of expected pipeline bubbles. Such a static speculation scheme is implemented by SaS Auction.

5 EVALUATION

The accelerator is implemented as a parameterizable hardware generator in the hardware description language Chisel [2]. The Zedboard [1] with a Xilinx Zynq XC7020 and off-chip DRAM is used

as a target platform for evaluation. The Zynq is a SoC FPGA, which comprises a dual core ARM Cortex A9 and a Xilinx Virtex FPGA. The open source FPGA-tidbits project [31] is used to generate host and DRAM interfaces for the accelerator. The SW baselines execute on a dedicated i9-10900X CPU running at 1.2 GHz. To filter out the non-determinism introduced by the operating system, each problem is run five times and the median execution time is used.

We evaluate three different SaS Auction designs and one SoA Auction against a C++ implementation and a version using the popular Gurobi framework.

We use two datasets for the evaluation. The first dataset consists of randomly generated sparse assignment problems. The practice of using random number generators to create assignment problems is discussed in Reference [7]. This dataset consists of assignment problems ranging from small 10×10 problems up to 600×600 problems. For each size the sparsity is varied from only a few valid object rewards per agent up to fully dense.

The second dataset is from a simulation of the state-of-the-art MOT algorithm introduced by Garcia et al. in Reference [12]. We have used different levels of validation gating to create problems with a greater range of sparsity. The algorithm maintains 100 tracks over 50 iterations, generating over 100,000 assignment problems.

The following implementations are evaluated:

- **C++ (SW)** is a single threaded C++ implementation of the Auction algorithm available at Reference [19]. It was allocated a dedicated i9-10900X core.
- **Gurobi (SW)** is using the industrial Gurobi Optimizer [15]. It was also allocated a single dedicated i9-10900X core. Gurobi solves the assignment problem as an integer linear programming problem using the simplex method.
- **SoA *x*PE** is our implementation of the state-of-the-art where x is the number of PEs. It is based on the work of Zhu et al. [36].
- **SaS *x*PE** is the Sparse and Speculative Auction implementation where x is the number of PEs.

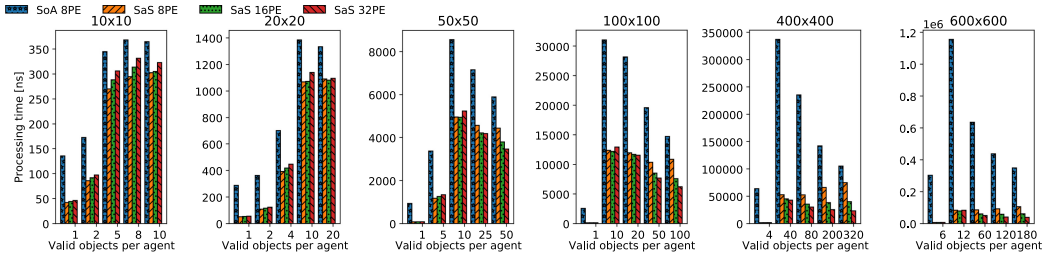
5.1 Performance

5.1.1 Random Sparse Dataset. Figure 12 shows the results of evaluating SaS and SoA Auction on the random sparse dataset. Each plot is divided into six subplots. Each subplot shows problems with the same input dimensions, which are given in the subplot title. The y-axes denote the processing time in nanoseconds, while the x-axes show the number of valid object-rewards per agent, i.e., the sparsity level. There are five sparsity levels in each subplot; e.g., in the 600×600 subplot there are results for problems with 6, 12, 60, 120, and 180 valid object rewards per agent. For each sparsity level, the reported value is the average processing time for that particular sparsity level.

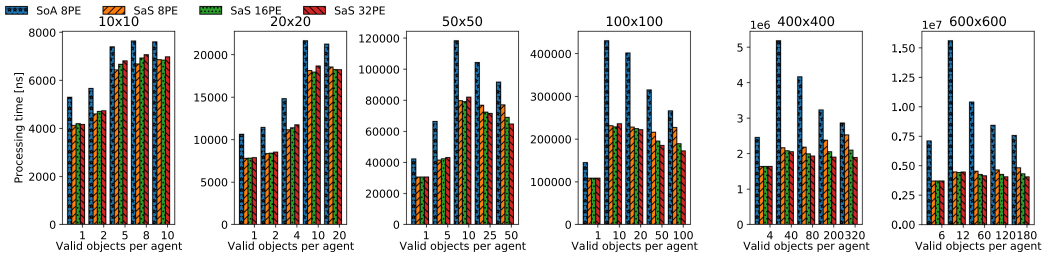
In Figure 12(a) the *core* processing times of SaS and SoA Auction with different number of PEs is shown. The core processing time is the time spent doing actual computation, neglecting the time spent transferring the problem matrices from DRAM to the on-chip memory. The core processing time is measured by the accelerator itself, which counts the cycles spent computing the solution. There are several things to note.

As expected, the potential speedup when taking advantage of speculation and sparsity-aware processing is big, as high as 50× for big sparse problems. It is also clear that the optimal number of PEs is related to the sparsity of the problem. The 50×50 subplot shows that SaS 8PE performs best while there are less than 8 valid objects per agent. Then SaS 16PE performs better until we have more than 16 valid objects per agent when SaS 32PE is best.

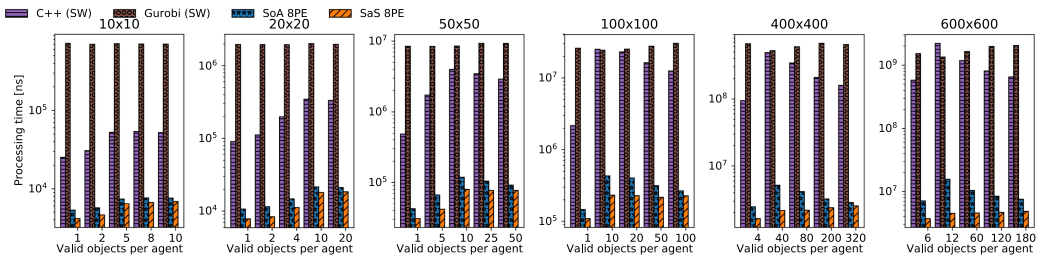
It is also clear that the speedup of SaS relative SoA is generally higher for sparse than for dense problems. For large and sparse input problems the potentially speedup is enormous. This is mostly



(a) *Core* processing time



(b) *Total* processing time



(c) *Total* processing time compared to SW

Fig. 12. Evaluation of SaS Auction on the random sparse dataset.

due to the sparsity-aware processing; e.g., for a 600 × 600 problem each agent in SoA 8PE will span 75 pipeline stages, regardless of the sparsity. When the problem is fully dense, the only remaining speedup is due to the dependency speculation.

Figure 12(b) depicts the *total* processing times of SaS and SoA Auction. The total processing time includes the time spent transferring the rewards from DRAM to the on-chip-memory and the time spent writing the final results back. Clearly, the relative speedup of the SaS accelerators is limited by the fact that a major portion of the runtime is spent moving data from the DRAM to the on-chip BRAM. This effect is most visible for highly sparse problems. Consider the 600 × 600 problems with only 6 valid object rewards per agent. The *core* processing time of SoA 8PE is 50× higher than that of SaS 8PE. However, the *total* processing time is only 1.8× higher. The reason is that these highly sparse problems require fewer iterations to solve than dense problems. This results in a situation where the core processing time only accounts for 2% of the total processing time.

Another interesting observation is that neither the core, nor the total, processing time is monotonically increasing as the problems get denser. Rather, the processing time peaks at a relatively low density. Figure 11 shows the number of iterations needed by the C++ (SW) implementation to solve a dataset of random sparse 50 × 50 problems. The x-axis shows the level of density and the y-axis shows the number of iterations. Surprisingly, the processing time peaks at 22% density with

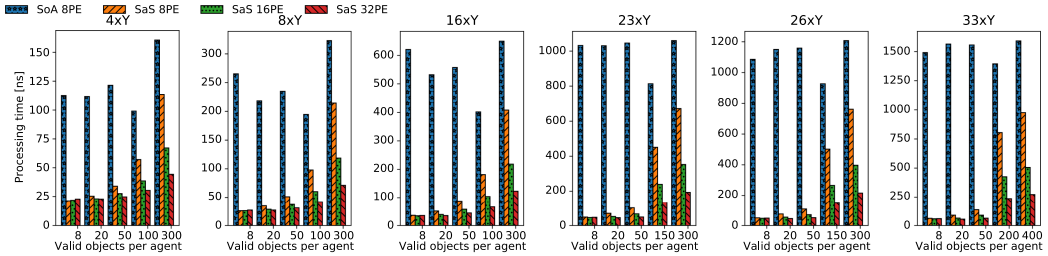


Fig. 13. *Core* processing time for the MOT dataset.

an average core processing time of 7,173 ns before monotonically decreasing as density increases. At 100% density the core processing time has decreased to an average of 4,964 ns. The mechanism behind this is not clear to the authors.

Figure 12(c) compares the *total* processing time of SaS and SoA Auction with SW implementations. The y-axis now uses a logarithmic scale. As described in prior work, there is enormous potential for speedup: from 100× speedup for the small 10 × 10 problems to 1,000× speedup for the sparse 600 × 600 problems. It is also clear that a vanilla C++ auction algorithm implementation outperforms industrial optimization software based on linear programming. However, for larger problems the speedup is less significant.

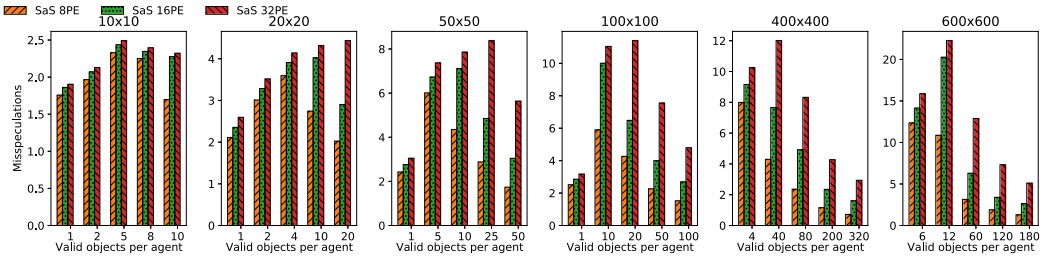
5.1.2 MOT Dataset. Figure 13 shows the results of evaluating SaS and SoA Auction on the MOT dataset. There are a few subtleties that make this plot different than the previous ones. Each subplot now shows problems with the same number of rows (i.e., agents). The subplots have titles like 16xY, which means that the problems have 16 agents, however, the number of objects vary from 1 to over 300. The x-axes show the number of valid object-rewards per agent; this is no longer directly tied to the sparsity level, since the number of non-valid object-rewards varies. The values on the x-axes now indicate limits; e.g., for the 16xY subplot, the first bar-group is the average processing time for problems with 1–8 valid object-rewards, the second is for 8–20 valid object-rewards, and so on.

As expected, the processing time of SaS Auction increases as the number of valid object-rewards increases. This is in line with the results from the randomly sparse dataset. Interestingly, this is not the case for SoA Auction, where the processing time does not seem to strongly correlate with the number of valid object-rewards. The reason is that the processing time of SoA Auction mostly only depend on the input dimensions of the problem. The input distribution of the MOT problems are such that the agents seldom bid for the same objects. Thus, the sparsity level is not an indicator of how many iterations are needed and thus to overall processing time. The SaS Auction, however, can take advantage of the sparsity and drastically reduce the core processing time.

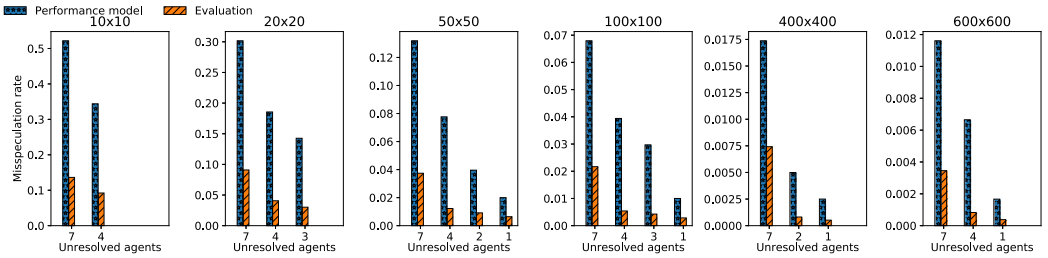
5.2 Performance Model Evaluation

Figure 14(a) shows the number of misspeculations for the SaS Auction accelerator when evaluating the random sparse dataset. As expected, SaS 8PE consistently has fewer misspeculations due to the fact that it has a lower bound on the maximum number of unresolved agents. In general, the number of misspeculations increase with the number of valid object-rewards until the number exceeds the number of PEs.

Figure 14(b) compares the expected rate of misspeculations, derived in the performance model section, with the measured rate of misspeculation rate for the SaS 8PE accelerator. The y-axis shows the misspeculation rate and the x-axis show the number of unresolved agents. The number



(a) Misspeculations



(b) Expected vs actual misspeculation rate as a function of unresolved agents

Fig. 14. Performance model evaluation on the randomly generated sparse dataset.

of unresolved agents are based on the sparsity level of the input problem according to

$$\mathrm{a_{span}} = \frac{n_{nonzero}}{n_{PE}} \tag{14}$$

$$\mathrm{unresolved} = \frac{\mathrm{max_{unresolved}}}{\mathrm{a_{span}}}, \tag{15}$$

where $\mathrm{a_{span}}$ is the agent span, i.e., how many pipeline stages each agent spans. The n_{nonzero} is the number of non-zero object rewards per agent, i.e., the sparsity level. Finally, $\mathrm{max_{unresolved}}$ is the maximum number of unresolved agents possible; i.e., the latency from PE stage to commit. For SaS 8PE, $\mathrm{max_{unresolved}} = 7$.

Clearly, the performance model is inaccurate and overestimates the misspeculation rate. The major reason for this is that the predicted misspeculation rate is based on the assumption that the pipeline is always full; i.e., for a 20×20 problem with 8 non-zero rewards per agent, we assume that there always will be 7 unresolved agents in the pipeline. In reality, the pipeline is only full in the first part of the execution. As there are fewer and fewer unassigned agents left, the pipeline will be less and less full until there is only one unassigned agent left. The execution can proceed for many iterations with only a single unassigned agent. During those last iterations, a misspeculation is impossible. We thus conclude that the performance model gives overly conservative upper bound of the misspeculation rate.

5.3 Area

Table 3 shows the resource utilization for the different implementations. SoA 8PE is our implementation of the accelerator based on the work of Zhu et al., while Zhu 16PE are the numbers reported in Reference [36]. The difference in BRAM usage stems from the fact that SoA 8PE stores the rewards on-chip while Zhu 16PE accesses the rewards from an off-chip memory. SoA 8PE needs 130 BRAMs to be able to solve 600×600 problems efficiently. A solution reading rewards from off-chip DRAM would be a lot slower and not be fair comparison to SaS, which stores the rewards, with

Table 3. Resource Utilization

	Accelerator				
	SoA 8PE	**Zhu 16PE**	**SaS 8PE**	**SaS 16PE**	**SaS 32PE**
Flip-flops	2,966	1,647	3,177	4,586	7,414
LUT	2,346	2,265	2,235	2,927	4,315
LUTRAM	57	540	121	229	432
BRAMs	134.5	6	79	83	91

sparse representation, on-chip. The difference in LUT and flip-flop usage likely stems from the fact that the SoA 8PE is based on SaS 8PE. The sparsity-aware and speculative optimizations were removed, but some overhead still remains.

We note that SaS 16PE uses 2.8× more flip-flops and 1.3× more LUTs than Zhu 16PE. This overhead is high, but expected. The Dram2Bram, PriceStore, and on-chip storage of the rewards make up over a third of the flip-flop overhead. Moreover, the pipeline of SaS 16PE is considerably longer and more information must be passed through the pipeline to support sparsity-aware processing and speculation.

6 CONCLUSIONS

In this article, Sparse and Speculative Auction, a novel implementation of the popular Auction algorithm for FPGAs, has been presented. It takes advantage of the high level of sparsity found in the assignment problems formulated by many algorithms. It employs both sparsity-aware processing and speculation and achieves up to 50× speedup over the state-of-the-art for highly sparse problems. It is evaluated on randomly generated sparse problems and realistic problems from a MOT simulator.

REFERENCES

[1] Avnet. 2021. ZedBoard HW Users Guide. Retrieved from https://digilent.com/reference/_media/reference/programmable-logic/zedboard/zedboard_ug.pdf.

[2] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. 2012. Chisel: Constructing hardware in a Scala embedded language. In *Proceedings of the Design Automation Conference*. 1212–1221. DOI: https://doi.org/10.1145/2228360.2228584

[3] Michael Beard, Ba Tuong Vo, and Ba-Ngu Vo. 2020. A solution for large-scale multi-object tracking. *IEEE Trans. Sig. Process.* 68 (2020), 2754–2769. DOI: https://doi.org/10.1109/TSP.2020.2986136

[4] D. P. Bertsekas. 1988. The auction algorithm: A distributed relaxation method for the assignment problem. *Ann. Oper. Res.* 14, 1–4 (June 1988), 105–123. DOI: https://doi.org/10.1007/BF02186476

[5] Dimitri P. Bertsekas. 1990. The auction algorithm for assignment and other network flow problems: A tutorial. *INFORMS J. Appl. Analyt.* 20, 4 (1990), 133–149. DOI: https://doi.org/10.1287/inte.20.4.133 arXiv:https://doi.org/10.1287/inte.20.4.133

[6] Christian Bliek1ú, Pierre Bonami, and Andrea Lodi. 2014. Solving mixed-integer quadratic programming problems with IBM-CPLEX: A progress report. In *Proceedings of the 26th RAMP Symposium*. 16–17.

[7] David F. Crouse. 2016. On implementing 2D rectangular assignment algorithms. *IEEE Trans. Aerosp. Electron. Syst.* 52, 4 (2016), 1679–1696. DOI: https://doi.org/10.1109/TAES.2016.140952

[8] R. Danchick and G. E. Newnam. 2006. Reformulating Reid's MHT method with generalised Murty K-best ranked linear assignment algorithm. *IEE Proc. - Radar, Sonar Navig.* 153 (Feb. 2006), 13–22. DOI: https://doi.org/10.1049/ip-rsn:20050041

[9] Xiaojun Duan, Huiying Liu, Hong Tang, Qing Cai, Fan Zhang, and Xiaotian Han. 2020. A novel hybrid auction algorithm for multi-UAVs dynamic task assignment. *IEEE Access* 8 (2020), 86207–86222. DOI: https://doi.org/10.1109/ACCESS.2019.2959327

[10] Abbas Ehsanfar and Paul Grogan. 2020. Auction-based algorithms for routing and task scheduling in federated networks. *J. Netw. Syst. Manag.* 28 (04 2020), 1–27. DOI: https://doi.org/10.1007/s10922-019-09506-y

[11] Ahed Elmsallati, Connor Clark, and Jugal Kalita. 2016. Global alignment of protein-protein interaction networks: A survey. *IEEE/ACM Trans. Comput. Biol. Bioinf.* 13, 4 (July 2016), 689–705. DOI: https://doi.org/10.1109/TCBB.2015.2474391

[12] A. F. Garcia-Fernandez, J. L. Williams, K. Granström, and L. Svensson. 2018. Poisson multi-Bernoulli mixture filter: Direct derivation and implementation. *IEEE Trans. Aerosp. Electron. Syst.* (2018). DOI: https://doi.org/10.1109/TAES.2018.2805153

[13] Karl Granstrom, Marcus Baum, and Stephan Reuter. 2017. Extended Object Tracking: Introduction, Overview and Applications. arXiv:cs.CV/1604.00970.

[14] Peng Gu, Zhongliang Jing, and Liangbin Wu. 2022. Robust adaptive multi-target tracking with unknown measurement and process noise covariance matrices. *IET Radar, Sonar & Navigation* 16, 4 (2022), 735–747. https://doi.org/10.104/rsn2.12216 arXiv:https://ietresearch.onlinelibrary.wiley.com/doi/pdf/10.1049/rsn2.12216.

[15] Gurobi. 2022. Gurobi optimizer reference manual, 2022. Retrieved from http://www.gurobi.com.

[16] Nina S. T. Hirata and Frank D. Julca-Aguilar. 2015. Matching based ground-truth annotation for online handwritten mathematical expressions. *Pattern Recog.* 48, 3 (Mar. 2015), 837–848. DOI: https://doi.org/10.1016/j.patcog.2014.09.015

[17] Donald Hung and Jun Wang. 2003. Digital hardware realization of a recurrent neural network for solving the assignment problem. *Neurocomputing* 51 (04 2003), 447–461. DOI: https://doi.org/10.1016/S0925-2312(02)00627-6

[18] D. L. Hung and Jun Wang. 1998. A FPGA-based custom computing system for solving the assignment problem. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines.* 298–299. DOI: https://doi.org/10.1109/FPGA.1998.707924

[19] Erling Rennemo Jellum. 2012. Auction algorithm C++. Retrieved from https://github.com/erlingrj/auction-cpp.

[20] Erling Rennemo Jellum. 2021. Auction Accelerator. Retrieved from https://github.com/erlingrj/auction-accelerator.

[21] Roy Jonker and A. Volgenant. 2005. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing* 38 (2005), 325–340.

[22] D. Klingman and R. Russell. 1978. A streamlined simplex approach to the singly constrained transportation problem. *Nav. Res. Logist. Quart.* 25, 4 (1978), 681–695.

[23] Srivatsan Krishnan, Zishen Wan, Kshitij Bharadwaj, Paul Whatmough, Aleksandra Faust, Gu-Yeon Wei, David Brooks, and Vijay Janapa Reddi. 2020. The Sky Is Not the Limit: A Visual Performance Model for Cyber-Physical Co-Design in Autonomous Machines. *IEEE Computer Architecture Letters* 19, 1 (2020), 38–42. https://doi.org/10.1109/LCA.2020.2981022.

[24] Kang Li, Xiaoguang Zhao, Zengpeng Sun, and Min Tan. 2017. Robust target detection, tracking and following for an indoor mobile robot. In *Proceedings of the IEEE International Conference on Robotics and Biomimetics (ROBIO).* 593–598. DOI: https://doi.org/10.1109/ROBIO.2017.8324481

[25] Wei Lian and Lei Zhang. 2020. A concave optimization algorithm for matching partially overlapping point sets. *Pattern Recog.* 103 (2020), 107322.

[26] Murad Qasaimeh, Kristof Denolf, Jack Lo, Kees Vissers, Joseph Zambreno, and Phillip H. Jones. 2019. Comparing energy efficiency of CPU, GPU and FPGA implementations for vision kernels. In *Proceedings of the IEEE International Conference on Embedded Software and Systems (ICESS'19).* 1–8. DOI: https://doi.org/10.1109/ICESS.2019.8782524

[27] Cyril Robin and Simon Lacroix. 2015. Multi-robot Target Detection and Tracking: Taxonomy and Survey. *Autonomous Robots* 40 (2015). https://doi.org/10.1007/s10514-015-9491-7

[28] Yousef Saad. 2003. *Iterative Methods for Sparse Linear Systems* (2nd ed.). Society for Industrial and Applied Mathematics. DOI: https://doi.org/10.1137/1.9780898718003

[29] B. L. Schwartz. 1994. A computational analysis of the auction algorithm. *Eur. J. Oper. Res.* 74, 1 (1994), 161–169. DOI: https://doi.org/10.1016/0377-2217(94)90214-3

[30] Fethulah Smailbegovic, Georgi Gaydadjiev, and Stamatis Vassiliadis. 2005. Sparse matrix storage format.

[31] Yaman Umuroglu. 2020. FPGA Tidbits. Retrieved from https://github.com/maltanar/fpga-tidbits.

[32] Cristina Vasconcelos and Bodo Rosenhahn. 2009. Bipartite graph matching computation on GPU. 42–55. DOI: https://doi.org/10.1007/978-3-642-03641-5_4

[33] Philip Wolfe. 1959. The simplex method for quadratic programming. *Econometrica* 27 (1959), 170.

[34] Kai Xue, Zhiqin Huang, Ping Wang, and Zeyu Xu. 2021. An exact algorithm for task allocation of multiple unmanned surface vehicles with minimum task time. *J. Marine Sci. Eng.* 9, 8 (2021). DOI: https://doi.org/10.3390/jmse9080907

[35] Ziying Zhang, Jie Wang, Dong Xu, and Yulong Meng. 2017. Task allocation of multi-AUVs based on innovative auction algorithm. In *Proceedings of the 10th International Symposium on Computational Intelligence and Design (ISCID'17).* 83–88. DOI: https://doi.org/10.1109/ISCID.2017.231

[36] Pengfei Zhu, Chun Zhang, Hua Li, Ray C. C. Cheung, and Bryan Hu. 2012. An FPGA-based acceleration platform for auction algorithm. In *Proceedings of the IEEE International Symposium on Circuits and Systems.* DOI: https://doi.org/10.1109/ISCAS.2012.6271395

Received 1 December 2021; revised 26 May 2022; accepted 13 June 2022