



Fast block distributed CUDA implementation of the Hungarian algorithm

Paulo A.C. Lopes^{a,*}, Satyendra Singh Yadav^b, Aleksandar Ilic^a, Sarat Kumar Patra^c

^a Instituto de Engenharia de Sistemas e Computadores, Investigação e Desenvolvimento, Instituto Superior Técnico, Universidade de Lisboa, INESC-ID/IST/UL, Rua Alves Redol n.9, 1000-029 Lisbon, Portugal

^b Department of Electronics and Communication Engineering, National Institute of Technology Rourkela, Odisha, India

^c Department of Computer Science Engineering, Indian Institute of Information Technology, Vadodara, India

HIGHLIGHTS

- Block Distributed Alternating Path Search with nontrivial proof.
- Multiple Simultaneous Alternating Paths.
- Graph Search is directly based on the original Hungarian algorithm resulting in efficient implementation.
- Optimized reductions for initial dual variables and dual variables update.
- Chosen $O(n^4)$ Implementation shows better performance than $O(n^3)$ in low range matrices.

ARTICLE INFO

Article history:

Received 16 November 2017

Received in revised form 30 January 2019

Accepted 20 March 2019

Available online 8 April 2019

Keywords:

Hungarian algorithm

Linear assignment problem

GPU

ABSTRACT

The Hungarian algorithm solves the linear assignment problem in polynomial time. A GPU/CUDA implementation of this algorithm is proposed. GPUs are massive parallel machines. In this implementation, the alternating path search phase of the algorithm is distributed by several blocks in a way to minimize global device synchronization. This phase is very important and has a big contribution to the execution time. Other advanced features also implemented are: parallel graph traversal; the parallel detection of multiple alternating paths in a single iteration; a simplified and fast matrix compression that stores the zeros of the slack matrix, resulting in very fast graph traversal; highly optimized reductions for the initial slack matrix calculation and update. This results in a fast implementation for moderate size problems.

© 2019 Elsevier Inc. All rights reserved.

1. Introduction

The Hungarian algorithm (HA) [1] is a solution to the linear assignment problem (LAP). The goal is, given a cost matrix A , which has elements a_{ij} that represents the cost of assigning a worker i to a task j , to determine the optimal assignment of the workers that minimizes the total cost.

Auction algorithm (AA) [2] and the Shortest Augmenting Path algorithm (SAPA) [3] are solutions to the same problem. The Hungarian algorithm presented in this paper has a worst-case computational complexity of $O(n^4)$, but there are other versions with $O(n^3)$ worst case complexity [4,5]. The SAPA has $O(n^3)$ worst case complexity.

This paper deals with the Hungarian algorithm. The Hungarian algorithm gives the best run time for low range matrices in our CPU implementations as presented in Section 5 although its run

time is much higher in the worst case. This can be explained by the fact that, in the worst case the dual update steps of the algorithm produces only one zero in the slack matrix. However, in low range matrices, many more zeros can be produced (more than n) resulting in a substantial decrease in the number iterations and the computational complexity. This moves the complexity from the dual update phase to the maximal matching phase of the algorithm. Low range matrices are the best suitable in any practical problem where data has low precision.

Parallel implementations of the SAPA are presented in [6,7] where they use a modification to the algorithm so that pairwise disjoint augmentation paths can be used in parallel. In [7] they make the algorithm asynchronous. There are also parallel implementation of the AA [8,9], distributed implementations [10] and GPU implementations [11].

A GPU implementation of the HA is presented in [5]. There are also parallel and GPU implementations of maximum matching algorithms that are used in the HA [12,13].

Applications of the LAP are for instance: solve the traveling salesman problem [14], tracking in image processing [11],

* Corresponding author.

E-mail address: paulo.lopes@tecnico.ulisboa.pt (P.A.C. Lopes).

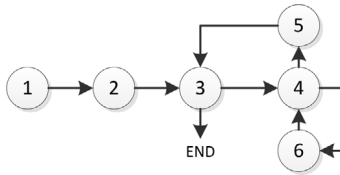


Fig. 1. The flow through the several steps of the Hungarian algorithm.

proteins–protein interaction networks, image feature extraction, linear algebra [9], telecommunication [15] and others. There are also other algorithms for other assignment problems in the literature [16].

2. The Hungarian algorithm

This work adopts a variation of the formulation of the Hungarian algorithm as presented in [1] described in the following.

2.1. Formulation of the Hungarian algorithm

Initialize the slack matrix with the cost matrix, and then work with the slack matrix.

STEP 1: Subtract the row minimum from each row. Subtract the column minimum from each column.

STEP 2: Find a zero of the slack matrix. If there are no starred zeros in its column or row star the zero. Repeat for each zero.

STEP 3: Cover each column with a starred zero. If all the columns are covered then the matching is maximum.

STEP 4: Find a non-covered zero and prime it. If there is no starred zero in the row containing this primed zero, go to step 5. Otherwise, cover this row and uncover the column containing the starred zero. Continue in this manner until there are no uncovered zeros left. Save the smallest uncovered value and go to step 6.

STEP 5: Construct a series of alternating primed and starred zeros as follows: Let Z0 represent the uncovered primed zero found in step 4. Let Z1 denote the starred zero in the column of Z0 (if any). Let Z2 denote the primed zero in the row of Z1 (there will always be one). Continue until the series terminates at a primed zero that has no starred zero in its column. Un-star each starred zero of the series, star each primed zero of the series, erase all primes and uncover every row in the matrix. Return to step 3.

STEP 6: Add the minimum uncovered value to every element of each covered row, and subtract it from every element of each uncovered column. Return to step 4 without altering any stars, primes, or covered rows.

The flow through the steps is illustrated in Fig. 1. This can be implemented by using an outer while loop for steps 3 to 6, and an inner while for steps 4 and 6. After step 4, the processor checks if it should go to step 5. In this case, it breaks the inner loop, resulting in running the code for step 5 that follows it. After this, the outer loop makes step 3 follow step 5. If step 3 detects that the algorithm is finished, then it breaks the outer loop, in fact terminating the code.

2.2. Proof that proposed formulation of the HA solves the LAP problem

In the following a proof of the HA is presented [17]. This is helpful to understand the algorithm and will be required to better understand some aspects of its GPU implementation.

The algorithm starts with a complete bipartite graph $G = (S, T; E)$ with n workers S and n tasks T and the goal is to find a perfect matching that minimizes the cost. The cost is in a

cost matrix A , which has elements a_{ij} that represent the cost of assigning a worker i to a task j .

First, if the matrix A is replaced by the slack matrix L with elements $L_{ij} = a_{ij} - u_i - v_j$ the solution does not change. This corresponds to subtractions of full rows or columns by constants. Let δ be the value subtracted at one full row or column, then for any possible assignment, the cost is also decreased by δ , since the row or column will appear once and only once in the assignment. So, the solution to the assignment problem is the same. The variables u_i and v_j are called the dual variables. Second, if it is possible to select the dual variables with a positive slack and the matching so that the resulting cost is zero, then this has to be the solution to the problem since the cost cannot be lower than zero for a positive slack. This can be achieved if the matching is done using the zeros in the slack matrix. The Hungarian algorithm accomplishes this.

After an initial selection of the dual variables, a subgraph G_y that only uses the zero slack edges is selected. Using this graph, an initial matching is made, but this matching is not maximum since it does not make the maximum number of matches using the subgraph.

Maximum matching phase: In order to achieve a maximum matching, do as follows. Let the edges used in the matching be called starred edges. Orient the graph G_y forming the graph \bar{G}_y making the un-starred edges run from S to T and the starred edges run from T to S . Now find a path through the graph from an unmatched worker to an unmatched task. This path is called an alternating path because it will alternate between going from S to T and then from T to S . After finding this path, we can traverse it and star the un-starred edges and un-star the starred edges, resulting in one more star or match. Repeat this step until there are no more alternating paths. This will result in a maximum matching.

Dual update phase: When getting the maximum matching, the König's theorem [18] states that we can also get the minimum vertex cover. Let R_S be the set of unmatched worker vertices and Z be the set of the vertices reachable from R_S . Then according to König's theorem $C = (S \setminus Z) \cup (T \cap Z)$ is a minimum vertex cover of G_y . Let $\Delta = \min\{a_{ij} - u_i - v_j : i \notin C, j \in C\}$. Note that each vertex of S corresponds to a row of the slack matrix and that each vertex of T corresponds to one column of the slack matrix. Δ is the minimum of the uncovered rows and columns of the slack matrix.

Proceed by decreasing u_i by Δ on the covered vertices of S and increase v_j by Δ on uncovered vertices of T . The operation just described has the net effect of changing the slack matrix as follows. It adds Δ to the elements with rows and columns covered and subtracts Δ from the uncovered elements, while the other elements remain unchanged. Note that Δ is the minimum of the uncovered elements. This results in at least one zero on the uncovered elements. Also, it removes any zeros in the elements covered by rows and columns. This increases the size of Z as described in the following.

The graph G_y is changed by adding and removing edges, the zeros of the slack matrix. The lost zeros will not alter the elements of Z while the new zeros increase the size of Z . The lost zeros do not matter because their edges do not start on Z . They have their rows covered, and so their starting vertices belong to $S \setminus Z$. Note that these zeros are oriented from S to T , since they are not starred. They are not starred because they are covered by rows and columns. Any starred zero cannot be covered by a row if it is covered by a column, since if its column belongs to $(T \cap Z)$ then its row belongs to Z since they are connected, and so it does not belong to $(S \setminus Z)$. The new zeros in the slack matrix increase Z because they start inside Z and finish outside Z . These are new zeros, un-starred and uncovered zeros, so they start at $(S \cap Z)$

and finish at $(T \setminus Z)$. This means that Z will grow for each dual variables update.

Finally, repeat the maximum matching phase and dual update phase until a perfect matching is found. Each time a dual update is performed Z grows until eventually a new alternating path is found. For each new alternating path found the number of matches grows until a perfect matching is found, proving the convergence of the algorithm.

Proposed formulation: In the following, we examine the several steps of the proposed algorithm formulation, and make the connection to the previous discussion. Note that the proposed formulation merges the maximum matching graph search with the determination of the minimum cover resulting in an efficient algorithm.

Step 1 makes initial guesses on the dual variables, u_i and v_j , in order to have the maximum number of zeros in the slack matrix. If there are many zeros, then it may be possible to get the optimum assignment in just a few iterations.

Step 2 chooses an initial assignment. The assignment is chosen using the zeros in the slack matrix. Each starred zero corresponds to one assignment. The zeros are selected randomly. Usually, it is not possible to make all the assignments using this random choice of zeros.

Step 3 initializes the search for an alternating path and checks for a perfect matching.

Step 4 searches for an alternating path through the graph. This step deserves detailed comments. The path found is marked using primes. The graph is actually traveled from the end to the start, so from unmatched columns to unmatched rows. At step 3 all matched columns are covered, then, starting at the unmatched columns and going through the graph in the reverse direction using an un-starred zero a row is reached. If there is an edge leaving this row, that is a starred zero in the row, then the step goes through it. Otherwise, an alternating path was found, and the algorithm continues to step 5. When going through the graph, the visited vertices need to be marked. This is accomplished by covering the row. Since the step is now at a new column, the column of the starred zero, then this column needs to be uncovered so that the new edges that leave from it are visible. When there are no more uncovered zeros, this means that there are no more edges to follow, and the end of the graph has been reached without finding any alternating paths. This means the algorithm found a maximum matching and it progresses to step 6 to update the dual variables.

There will be zeros marked as prime that will not be part of an alternating path, but if at step 5 one starts at an unmatched row and goes through the marked path in the graph using the primes, this time in the forward direction, it will always reach an unmatched column which is the actual goal.

Notice that, the cover and uncover operation only changes the way the star is covered by moving it from a column to a row, so stars are always covered by a row or by a column. This implies that the total of columns and rows covered is equal to the number of matches. The cover that resulted from step 4, after the maximum matching is found, is the minimum cover of the König's theorem. This can be seen by applying König's theorem to the mirror graph with the rows inverted with the columns. The minimum cover will be $C = (S \cap Z') \cup (T \setminus Z')$ where Z' is the set of vertices that can be reached by starting at unmatched columns (instead of rows), as is done in this step. As the step progresses through the graph, the reached rows are covered since as they belong to $S \cap Z'$. The reached columns are uncovered since they do not belong to $T \setminus Z'$.

Step 5 goes through the alternating path and exchanges starred zeros and not starred zeros. The path from rows to columns will be marked by primes and from columns to rows by stars. After a

star, there will always be a prime because in order for step 4 to reach the starred zero it had to go through a zero and prime it. This makes sure that the path always ends at a column and so it is, in fact, an alternating path.

Step 6 changes the dual variables as previously discussed.

3. Introduction to CUDA

This paper is about a Compute Unified Device Architecture (CUDA) implementation of the Hungarian algorithm. CUDA is NVIDIA's general-purpose parallel computing framework and programming model for GPUs [19]. It is available in several languages, but the C++ version is used in this work. GPUs are massive parallel machines that can achieve much higher performance than CPUs in some applications.

The GPUs are divided into several Streaming Multiprocessors (SM), each with a large number of cores that run threads. The threads are grouped into groups of 32 called warps that are run in a single instruction multiple (SIMD) data fashion. This means that all threads in the warp should be executing the same piece of code. Failure to achieve this causes warp divergence that can have a significant impact on performance. It is common in GPU to have many threads executing the same line of code (but processing different data).

In CUDA, an application is divided into kernels that can be started from the CPU or from another kernel (parent kernel) using dynamic parallelism. Each kernel is formed by a grid of threads that are grouped into blocks of threads. So, a grid is a set of blocks that are sets of threads. The grids can be one, two or three dimensional but in this work, only one-dimensional grids are used.

There are mainly three types of memory in the GPU. Local memory, exclusive to each thread, shared memory that is shared by all threads in a block, and global memory that is shared by all the threads in the GPU. Global memory is persistent across kernel calls. Shared and local memories are much faster than global memory. Shared memory is available at the SM. One SM can run one or more block, but one block cannot be split across multiple SMs (or they would not be able to share memory).

Synchronization on the GPU can be performed in a block by using a `__syncthreads()` directive, but not on the whole device. The independence between two blocks means that they can be run in any order, in series or parallel, in the same SM or different SMs. Global synchronization is achievable by stopping and starting the kernels. In this fashion, it is guaranteed, for instance, that all the blocks have finished before the new kernel starts. However, this can be time-consuming.

In this work, a careful analysis of how to split the code and data through blocks and kernels was required.

4. GPU implementation

This section describes our GPU implementation of the Hungarian algorithm. It goes from step 2 to 6 and finishes with 1 because the last is implemented with the techniques used in step 6. This code is available online at [20]. A CUDA implementation is assumed. The `thread_ID`, `block_ID` and `block_Dim` are the ID of the thread, the ID of the block and the number of threads per block, respectively. All the elements of the CUDA grid are taken to be unidimensional.

The HA assumes a serial implementation, so it has to be modified to allow parallelization. For instance, a step 4 direct implementation would require processing one zero at a time. A

simple improvement is to process all lines in parallel. Other steps have similar problems.

4.1. Data structures

Through the code, the following data structures are used. They are all instanced in global memory. This is required since they are used in different blocks of the kernels and different kernels. However, most of the kernels make intensive use of the GPU memory hierarchy that caches global memory using shared memory, and some use shared memory explicitly.

slack: $n \times n$ column-wise matrix of slack values. Using the slack instead of the cost and the dual variables leads to a more computationally efficient algorithm.

zeros: A vector containing the zero slack entries of the slack matrix in random order. This is called the compressed version of the slack matrix. The entries are stored as the integer that gives the position in the column-wise slack matrix, and there is a separate variable for the vector size. The maximum vector size is the $n \times n$.

covered_row and covered_column: These are two Boolean vectors that indicate if the row or columns are covered.

row_of_star_at_column and column_of_star_at_row: These vectors mark the slack zeros as stars. Since there is only one star per row or column, this can be done using these vectors. The vectors give the row/column of the start at a given column/row. One vector would be enough, but we use two vectors to make it easy travel through the graphs both ways, by going from columns to rows or from rows to columns. This means that starring a zero consists of writing in both vectors. The vectors are initialized at sentinel value of -1 that represents no star at the row or column.

column_of_prime_at_row: This vector marks the slack zeros as primes. Since there is only one prime per row, this can be done using a single vector that takes the row and gives the column of the prime. The vector also uses -1 as a sentinel value.

row_of_green_at_column: This vector marks the slack zeros as greens. Since there is only one green per column, this can be done using a single vector that takes the column and gives the row of the green. The vector also uses -1 as a sentinel value.

Note that this choice of data structures allows fast graph traversal since the actual slack matrix is only used in steps 1 and 6; the covering of visited nodes and covered rows and columns is unified, and the marking of the alternating path simplified.

4.2. Step 2 - Initial matching

The step 2 kernel runs in a single GPU block but uses the maximal number of threads available in the block. We chose to run it in a single block to reduce the synchronization requirements and because this is enough to guarantee that the step has very low contribution to the execution time.

If the matrix size is greater than the maximum number of threads, then each thread will process more than one zero. Otherwise, each thread will process one zero and try and make it starred, covering its row and column. Once a zero is covered it cannot be selected for starring, so the process of selecting the zero and starring its row and its columns should be atomic.

This is implemented using the pseudo-code in Algorithm 1. The `atomicExch` function stores the second argument in the first argument and returns the old value of the first argument atomically. Whenever a thread finds that it has to uncover a row that it had covered (because it cannot cover the column) the process has to be repeated. This makes sure that all possible matches are done.

At each iteration of the loop, at least one row or column gets covered. This is because if a zero passes the covered test, before the next iteration at least a row or a column gets covered. There are three options: (1) The zero is starred, and the row and column are covered. (2) The row was covered by another zero. (3) The column was covered by another zero. This means that in the worst case the loop terminates in $2n$ iterations. However, note that in the typical case the number of iteration will be much lower. If the zeros that pass the cover test are in different lines and columns then the number of covered lines and columns in one iteration will be much higher, and it is unlikely, although possible, that they are all in the same line or column.

4.3. Step 3 - Graph search initialization and termination testing

Step 3 has two kernels one for initialization and another for the actual step. This is required to achieve global synchronization after the initialization. The initialization makes all the rows and columns uncovered and resets the count of starred zeros to none.

The second kernel is run using several blocks, with the default block size, and with a total number of threads equal to the matrix size. It covers the column associated with the thread and block ID if there is a starred zero in that column and counts the number of starred zeros using an atomic-add. Although the atomic operation will serialize the count, this will be fast and will not be significant for the overall performance.

4.4. Step 4 - Alternating path graph search

Step 4 has a large share of the computations in many applications. Together with step 5, it forms the maximum matching part of the algorithm. Applications with low-cost ranges rely more on the maximal matching part and less on the dual update part (step 6) of the algorithm. In these applications, the original Hungarian algorithm has advantages relative to its $O(n^3)$ version, the shortest path, and Auction algorithm. The original implementation of step 4 is serial, but the fact that the zeros can be processed in any order allows a significant degree of parallelism.

Algorithm 1 Step 2

```

i = thread_ID
select zero i
repeat
  if column and row of zero are uncovered then
    if not atomicExch(covered_row[row_of_zero],1) then
      if not atomicExch(covered_column[column_of_zero],1)
      then
        star the zero
      else
        uncover row of zero
        signal to repeat
      end if
    end if
  end if
  sync threads of block
until not signaled to repeat

```

4.4.1. Step 4 - Implementation

This step was implemented using a kernel with the graph split in blocks processed by GPU blocks (of threads). A nontrivial proof that this can be done is presented in [Appendix A](#).

Splitting the graph and running each piece in a different block is a significant contribution of this paper, along with the proof that this can be done. This allows a significant reduction in global synchronization operations that are implemented by kernels stop and restart. Previous approaches require global synchronization after processing the zeros to guarantee that the changes in the covered columns and rows are seen by the threads in other blocks. In this approach, each block can continue to process the zeros uncovered by its set of lines before a global synchronization is required. Even inside each kernel, all the zeros are processed in parallel.

The graph is split by distributing the zeros by the GPU blocks. This can be done in any fashion, in the proposed algorithm they are just split by columns: zeros in column 0 to $a-1$ are in block 0; zeros in column a to $2a-1$ are in block 1, etc. Since the matrices are stored column-wise this results in faster memory access. The covered_row and covered_column global variables are treated as volatile in this step. This allows blocks to communicate with each other and guarantees S4 in our proof in [Appendix A](#). Despite this communication, there is no global synchronization guarantees inside the kernel, so in order to achieve this, the kernel needs to be stopped and restarted. This is done after each block finishes processing (it has no more uncovered zeros to process) if there were newly uncovered columns by some of the blocks and the variable signaling to go to 5 were not set. Only in this way, it will be certain that all the blocks have seen all the changes made by other blocks.

The pseudo code for this step is presented in Algorithm 2. It consists in repeatedly processing each zero as described by the HA in parallel until all zeros are covered or an alternating path is found. However, in order for this to work some premisses described in [Appendix A](#) need to be met.

Algorithm 2 Step 4

```

volatile covered_row and covered_column
repeat
  repeat
    sync threads of block
    resets variable that indicates if zeros were found
    sync threads of block
    gets a zero z of the block, its row r, and column c
    if the zero is uncovered (check if the column is covered first) then
      prime the zero
      signals to repeat the kernel
    if there is a star at row r then
      covers the row of the star
      device fence
      uncovers the column of the star
    else
      terminates and signals to go to step 5
    end if
  end if
  sync threads of block
until this block did not find any zeros
sync all threads (by stopping and restarting the kernel)
until no zeros were found or goes to 5

```

Algorithm 3 Step 5a

```

i = thread_ID + block_ID * block_Dim
column = column_of_prime_at_row[i]
if there is a prime but not a star at row i then
  row_of_green_at_column[column]=i
  while there is a star in column, column do
    row = row_of_the_star_at_column[column]
    column = column_of_prime_at_row[row]
    row_of_green_at_column[column]=row
  end while
end if

```

4.5. Step 5 - Applying the alternating paths

Step 5 has two kernels step 5a and step 5b. This is required because in step 4 multiple alternating paths can be found at the same time. Step 4 is equivalent to a search for alternating paths through the bipartite graph from the end to the beginning and marking the path by priming the zeros. Since there is only one prime per row for every row there will always be a single path to choose to get the desired end, so the paths found are easy to travel. However, two paths may join. Applying one of these paths invalidates the other, so one needs to be chosen.

The solution is to traverse the paths from beginning to end and mark the paths as green. Marking the path as green corresponds to storing the row of the green zero in a vector that takes the column as index. This means that there would only be one green per column (and per row because greens are also primes). When there are multiple primes per column, the last prime to be processed is selected to become a green, in fact implementing the path selection process previously described. Then the paths can be traversed from end to the beginning using the greens vector, and since there is only one green per row and per column, the path never splits or joins. This means that all paths can be applied in parallel. Note that it is not possible to select just one prime per column without going through the paths because the path marked in step 4 is meant to be traversed in the forward direction and in the reverse direction there are dead ends that should not be traversed. There are no dead ends in our green marked path when traversed from the end to the beginning because it was marked starting at the start.

Therefore, step 5a does the forward pass that builds the green vector and step 5b does the reverse pass that applies the alternating path. Multiple paths are traveled at the same time, since step 5a has a thread for each row, and step 5b has a thread for each

Algorithm 4 Step 5b

```

j = thread_ID + block_ID * block_Dim
row_of_Z0 = row_of_green_at_column[j]
if there is a green but not a star in column j then
  column_Z2 = column_of_star_at_row[row_Z0];
  column_of_star_at_row[row_Z0]=j
  row_of_star_at_column[j]=row_Z0
  while column_Z2 is positive or zero do
    row_Z0 = row_of_green_at_column[column_Z2]
    column_Z0 = column_Z2
    column_Z2 = column_of_star_at_row[row_Z0]
    column_of_star_at_row[row_Z0]=column_Z0
    row_of_star_at_column[column_Z0]=row_Z0
  end while
end if

```

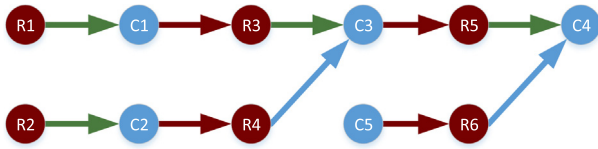


Fig. 2. Simple example showing a path marked with greens in a graph formed by the paths marked with primes. The starred zeros (edges) are red, the primed zeros are blue, and the green zeros are green. Rows are red circles and columns are blue circles. Unmarked zeros are not shown. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

column. However, since the number of alternating paths found at the same time is usually not very high, these kernels are mostly serial, although fast. The pseudo-code for these steps is presented in Algorithms 3 and 4.

Fig. 2 shows a simple example of a graph with the path marked with primes resulting from step 4. As can be seen there are point where two paths join, but after processing in step 5a, there is only one green at the union. The example also shows a possible dead end at C5 if the path was traveled from the end to the start (right to left) and the greens were chosen randomly one per column. As shown there is only a simple path through the greens from the end to the start.

4.6. Step 6 - Dual update

Step 6 is implemented with a reduction that calculates the minimum using two kernels, a kernel for the add-subtract operation and a final kernel to do the matrix compression. The two kernels for the reduction are implemented as follows.

This process is also described in [21] but since it was modified for step 1, it is also described here. The matrix is taken as a large vector, for which one wants to calculate the minimum of the uncovered values. First, some parallel-serial minimum calculations are performed using a large number of threads and blocks that results in full occupancy of the GPU. Each thread calculates the minimum of a set of data in a serial fashion and stores the result in shared memory. If the values are covered, then they are assumed to be the maximum integer so that they do not alter the minimum. The process continues with a reduction in shared memory. Two values are read from separate halves of the block of data; the minimum is calculated, and the result is stored back in shared memory. This is repeated with a data block of half size until the resulting size is one. This is done with a fully unrolled loop, assuming that the maximum size of the blocks of data is $2N$ if the maximum number of threads per block is N (N is usually 1024). When the number of working threads becomes smaller than 32, the size of a warp, then the code is further optimized to remove the thread synchronization command (`__syncthreads()`). Finally, the result of the reduction of each block is stored back in global memory. Note that each block of threads (that share shared memory) is responsible for one data block and will produce one result, so the global reduction is not finished. The second kernel is similar to the first kernel, but now it is called with only one block of threads and produces just one result, the required minimum. The pseudo-code for the first kernel of the reduction in step 6 is presented in Algorithm 5.

After the reduction step, the minimum is added to the elements that are covered by the rows and by the columns and subtracted from the elements that are uncovered, as in the pseudo-code in Algorithm 6.

The variable with the number of zeros is also initialized in this kernel. Finally, after the new matrix is calculated, the vector Zeros

Algorithm 5 Step 6 reduction 1

```

i = thread_ID + block_ID * block_dim * 2
tid = thread_ID
while i < n do
    i2 = i1 + block_dim
    Calculate rows and columns, from i1 and i2
    if i1 is not covered then
        g1 = global_data[i1]
    else
        g1 = Maximum
    end if
    do the same for i2 and g2
    shared_data[tid] = min(shared_data[tid], g1, g2)
    i = i + 2 * block_dim * number_of_blocks
end while
sync_threads
if block_dim >= 1024 then
    if thread_ID < 512 then
        shared_data[tid] = min(shared_data[tid],
                                shared_data[tid+512])
    end if
    sync_threads
end if
...
if block_dim >= 128 then
    if thread_ID < 64 then
        shared_data[tid] = min(shared_data[tid],
                                shared_data[tid+64])
    end if
    sync_threads
end if
if thread_ID < 32 then {inside the warp}
    if block_dim >= 64 then
        shared_data[tid] = min(shared_data[tid],
                                shared_data[tid+32])
    end if
    ...
    if block_dim >= 2 then
        shared_data[tid] = min(shared_data[tid],
                                shared_data[tid+1])
    end if
end if
if tid = 0 then
    global_data[block_ID] = shared_data[0]
end if

```

Algorithm 6 Step 6 add\sub

```

i = thread_ID + block_ID * block_dim
calculate row, l, and column, c, from i
if row is covered and column is covered then
    slack[i] = slack[i] + minimum
end if
if row is uncovered and column is uncovered then
    slack[i] = slack[i] - minimum
end if

```

is calculated in another kernel. This kernel is launched with one thread per matrix element. Each thread checks if the value is zero, and if so it stores its index in the zeros vector. Each time a zero is found the number of zeros is incremented using an atomic add. This serializes the writing of the vector but is very fast and not critical for the global algorithm performance. This is described in the pseudo-code in Algorithm 7.

Algorithm 7 Step 6 compress matrix

```

i = thread_ID + block_ID * block_dim
if slack[i] = 0 then
    increment zero_vector_size atomically
    zeros[zero_vector_size] = i
end if

```

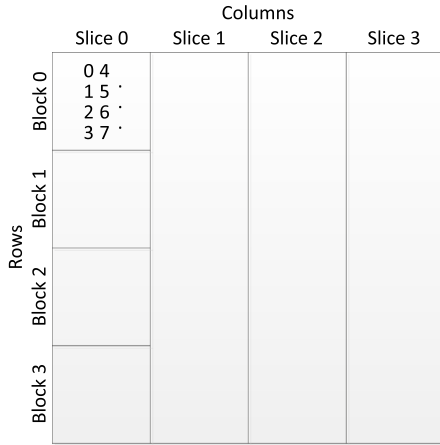


Fig. 3. How the matrix is split across threads and blocks for step 1, find the maximum in rows.

4.7. Step 1

Step 1 is also done with a reduction (like step 6), but it is more complex, so we left it to the end. This step is not critical in many cases because it runs only once. However, in low-resolution matrix applications, the solution can be achieved with very few iterations, and in this case, this step is important. In other cases, this step can be implemented in a much simpler way where the minimums or maximums in each row or column can be calculated serially, with one thread per row or column. In this subsection, however, we present an implementation where the minimums and maximums are calculated with a bunch of parallel reductions that results in significant savings in the time required for the step.

For the calculation of the maximum of the rows, the matrix is divided as presented in Fig. 3.

The matrix is first divided into slices. The first calculations will be to perform a large number of parallel–serial minimums (a large number of threads, each performing a serial minimum) between the values at the same position in different slices. This will result in a set of values in a single slice. Then the slice is divided into data blocks, that correspond to the thread blocks. Each block gets an independent set of rows so that each calculates a set of minimums independently. Inside the block, the threads are distributed column-wise across the data as represented in Fig. 3. This results in a code that is very similar to the reduction in step 6, the only difference being that the reduction stops when it would start to calculate minimums from two values on different rows, that is, when the distance between the two is less than or equal to the number of rows per block. This is illustrated by the pseudo-code fragment in Algorithm 8.

There are always more rows and columns than blocks, so there is no need for a second reduction step as in step 6. After calculating of the row maximums, the rows are subtracted by the maximums to get the new slack matrix as in Algorithm 9.

Next, the min in the columns of the new slack matrix are calculated. The algorithm is almost the same as the one used for calculating the max in rows, but now we exchange rows

Algorithm 8 Step 1 main differences to step 6 reduction 1

```

sync_threads
if block_dim >= 1024 and n_rows_per_block < 1024 then
    if thread_ID < 512 then
        shared_data[tid] = max(shared_data[tid],
                                shared_data[tid+512])
    end if
    sync_threads
end if
...
if block_dim >= 128 and n_rows_per_block < 128 then
    if thread_ID < 64 then
        shared_data[tid] = max(shared_data[tid],
                                shared_data[tid+64])
    end if
    sync_threads
end if

```

Algorithm 9 Step 1 max-slack

```

i = thread_ID + block_ID * block_dim
get the row l from i
slack[i] = max_in_row[l] - slack[i].

```

and columns, in fact transposing the matrix. Finally, the slack matrix is updated by subtracting the columns minimums from the columns.

5. Experimental results

This section starts by presenting a few results comparing CPU (writing in the C language) implementations of the $O(n^4)$ Hungarian algorithm (O4) and $O(n^3)$ Hungarian algorithm (O3), the shortest path (SP) and the auction algorithm (A). This is to show that in our results for small range of values for the cost and large sizes the O4 algorithm has the lowest run time. This is although it has higher worst-case complexity. These results show that the O4 algorithm is a good choice for GPU implementation in many cases.

In this section, the matrices used for comparison are random matrices of uniformly distributed integers within a given range of values. For matrices of sizes n the ranges are from 0 to $0.1n$; 0 to n or 0 to $10n$. Different ranges are used because this greatly affects the behavior of the algorithm. A small range of values makes zeros in the slack matrix very likely, while a large range makes zeros much less likely. In turn, this determines the number of iterations required for convergence of the algorithm. For small ranges, each run of step 6 tends to produce many zeros, resulting in much less iterations. So, step 6 will be very important for large ranges, and step 1 and 2 will be important in small ranges.

Fig. 4 plots the run time of the algorithms with varying problem size to show that they all have run times greater than the O4 algorithm for fairly large sizes when the range is equal to the matrix size. Fig. 5 shows that when the range increases the O4 starts getting worse becoming the worst algorithm for large ranges.

The section progresses to show experimental results showing that significant speedup is achieved by the CUDA implementation of the Hungarian algorithm when compared to its CPU version. The results were obtained in a GeForce GTX 970 GPU with 1664 CUDA cores.

First Fig. 6 shows a plot comparing the run time of the our CUDA implementation of the $O(n^3)$ Hungarian algorithm with

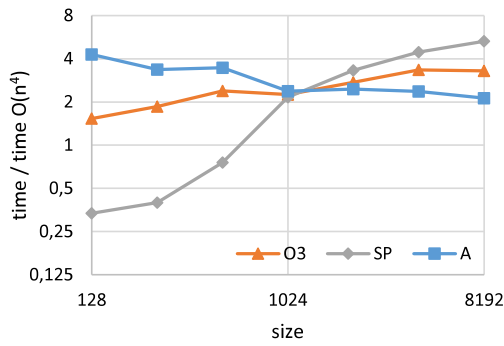


Fig. 4. Run time of several linear assignment algorithms when varying the problem size. The range is equal to the matrix size. The plot shows the run time of the algorithm divided by the run time of the $O(n^4)$ Hungarian algorithm.

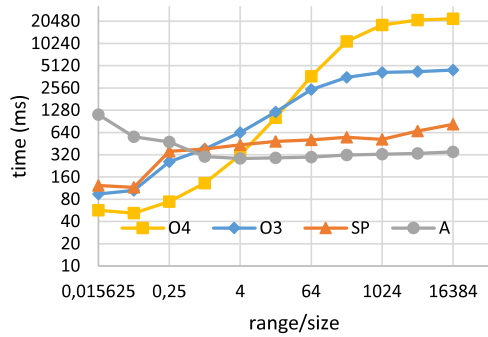


Fig. 5. Run time of several linear assignment algorithms when varying the range of the values in the problem for a fixed problem size of 2048.

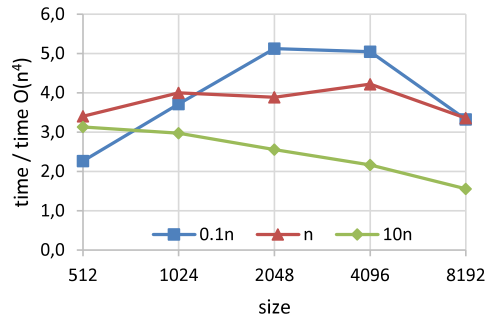


Fig. 6. Run time of the CUDA implementation of the $O(n^3)$ Hungarian algorithm divided by the run time of its CUDA $O(n^4)$ version. The lines show matrices with ranges 0 to $0.1n$, 0 to n and 0 to $10n$.

the run time of its CUDA $O(n^4)$ version, to confirm our previous results that O4 algorithm is faster for our test matrices by a factor of more than about two.

The remaining plots compare the CUDA O4 algorithm and a similar C/CPU implementation. The CPU version values were taken from a i7 6700 K CPU at 4.00 GHz with 4 cores and 32 GB of memory.

Figs. 7, 8 and Table 1 show the run time for the CUDA and CPU version of the Hungarian algorithm. It can be seen that the CUDA version time grows much slower with the matrix size, n , resulting in a significant performance gain for large n (≥ 512). This is as expected since the amount of parallelism that can be extracted from the algorithm grows with n . The speedup is shown

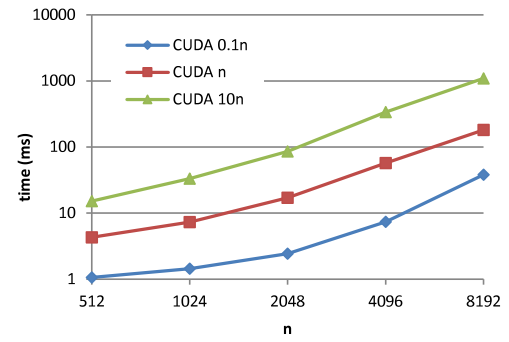


Fig. 7. Run time for the Hungarian algorithm implemented in CUDA with variable matrix size, n , and different range values: 0 to $0.1n$, 0 to n , and 0 to $10n$.

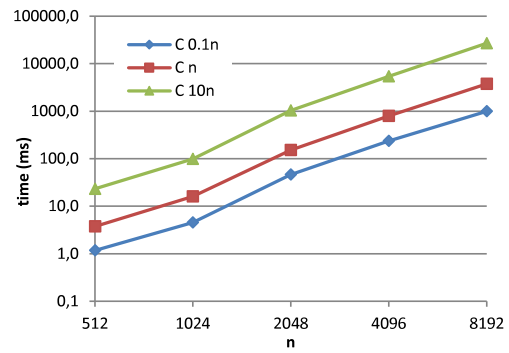


Fig. 8. Run time for the Hungarian algorithm implemented in CPU with variable matrix size, n , and different range values: 0 to $0.1n$, 0 to n , and 0 to $10n$.

Table 1

Run times for the Hungarian algorithm in CUDA and CPU (ms).

Size\range	CUDA			C		
	0.1n	n	10n	0.1n	n	10n
512	1.06	4.31	15.20	1.17	3.74	23.27
1024	1.45	7.30	33.18	4.52	16.03	98.75
2048	2.43	17.03	85.49	46.16	151.14	1041.48
4096	7.38	57.42	337.56	236.62	796.34	5419.01
8192	38.08	181.21	1094.78	994.81	3760.03	26954.23

Table 2

Speedup for the Hungarian algorithm in CUDA versus CPU.

n	0.1n	n	10n
512	1.10	0.87	1.53
1024	3.12	2.20	2.98
2048	18.99	8.88	12.18
4096	32.07	13.87	16.05
8192	26.12	20.75	24.62

in Table 2 and plotted in Fig. 9. Significant speedup is achieved for large sizes and all ranges.

Figs. 10–12 plot the relative contribution of the steps for the total time in the CUDA version of the algorithm. Almost all steps are important in one or another case. Note, however, that much effort was made to reduce the most time-consuming steps resulting in the current algorithm. The change of the importance of steps 1, 2 and 6 with the range was already discussed. The other steps importance is not so dependent on the range. The same charts are not plotted for the CPU version because these

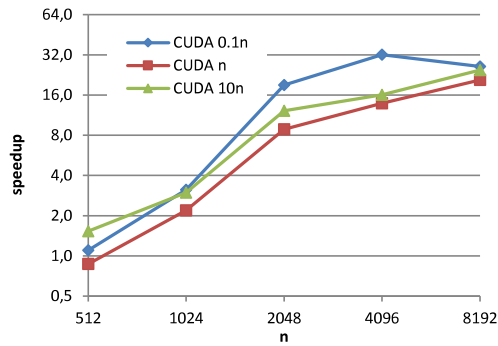


Fig. 9. Speedup for the Hungarian algorithm, CPU_time/CUDA_time, with variable matrix size, n , and different range values: 0 to $0.1n$, 0 to n , and 0 to $10n$.

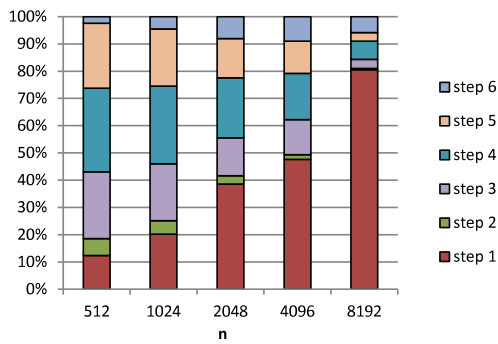


Fig. 10. Relative times taken in each step of the Hungarian algorithm implemented in CUDA with variable matrix size, n and a range of 0 to $10n$.

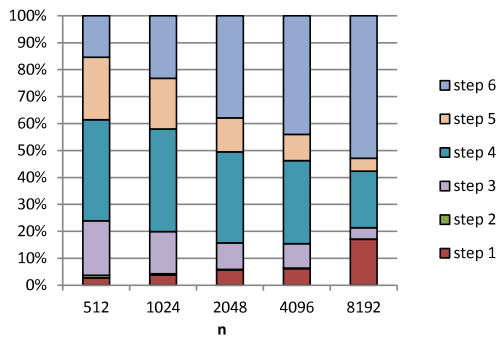


Fig. 11. Relative times taken in each step of the Hungarian algorithm implemented in CUDA with variable matrix size, n and a range of 0 to n .

are simpler and less interesting for our goal. In the CPU version for the range $0.1n$ the run time is dominated by step 1; for the range of n the time is dominated by steps 1 and 6; and for a range of $10n$ the time is dominated by step 6.

The speedup achieved in each step compared to the CPU version is presented in Table 3 for a range of 0 to n . This table is similar for other ranges. Step 6 achieves a high speed up because of the reduction used to obtain the minimum, the addition and subtraction operations and the matrix compression all are efficient use the GPU resources achieving a high occupancy. This allows the usage of the very high global memory bandwidth of

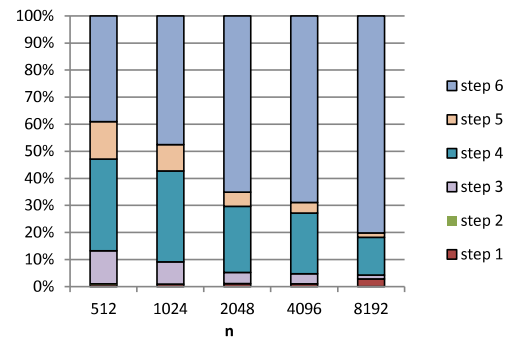


Fig. 12. Relative times taken in each step of the Hungarian algorithm implemented in CUDA with variable matrix size, n and a range of 0 to $10n$.

Table 3

Speedup for the several steps and problem sizes, for a range of 0 to n .

n	512	1024	2048	4096	8192
step 1	6.90	12.88	42.28	58.19	28.00
step 2	0.15	0.30	0.67	0.68	1.08
step 3	0.02	0.04	0.08	0.07	0.13
step 4	0.47	0.96	1.42	1.63	2.66
step 5	0.06	0.12	0.21	0.21	0.38
step 6	3.28	5.95	16.66	22.87	29.49
global	0.87	2.20	8.88	13.87	20.75

Table 4

Run times for the Hungarian algorithm in Tesla K20c GPU (ms).

n	$0.1n$	n	$10n$
512	2.08	8.85	30.76
1024	2.72	16.20	66.20
2048	4.28	32.39	156.86
4096	12.36	100.33	531.53
8192	44.10	282.98	1716.38

the GPU. Step 1 is similar, but in this case, a large set of parallel reductions is used.

In steps 1 and 6 use the GPU very efficiently. GPU processing also speeds up the processing of step 4 for the largest problem sizes. Other steps do not generally have a speedup compared to CPU, and in fact, they usually show worse performance than CPU code. This is because there is not enough parallelism in these steps to achieve a high occupancy of the GPU even with the techniques presented in this paper and because these kernels finish very fast and the time for global synchronization (kernel stop and restart) becomes important. However, the global results are still very encouraging. This can be seen in Table 3 that presents the speed up for a range of 0 to n . The steps speedups for other ranges are similar. Step 3, for instance, has the worst performance but it has low contribution to the total processing time even with its poor performance. The processing of these steps (2, 3 and 5) was kept in the GPU and not in the CPU to avoid expensive data copy operations between GPU and CPU even when for instance when transferring only the compressed matrix after steps 1 or 6. However, further study of this issue may be future work.

Last, times using a Tesla K20c GPU are presented in Table 4 to make it easy to compare with the results in [5] that use a similar GPU. Our times are much better making it the fastest GPU implementation known by the authors for the problem sizes considered in this paper.

6. Conclusion

This paper presents a GPU/CUDA implementation of the Hungarian algorithm (HA). Large speedups are obtained compared with the single processor version, especially for relatively large problems and low ranges. These gains are mostly on the highly optimized reductions in steps 1 and 6 and for large matrices in step 4. In some steps, global synchronization time (implemented by kernel stop and restart) seems to be an important factor. Namely, the alternating path search, step 4, is very important and was optimized so that it is efficiently distributed by blocks reducing global synchronization operations. A proof that the implementation is correct is included in the paper. Losses in all steps are minimized through careful optimization of every step. Other advanced features of the implementation are: the determination and application of several alternating paths in parallel (steps 4 and 5); the simplified matrix compression and marking of primes and stars that enable very fast graph traversal. All these together result in significant speedup.

Acknowledgment

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2019.

Appendix A. Proof of the parallel/serial equivalence of the algorithm used in step 4

In this section, it is shown that Algorithm 2 is correct. This is done in the following theorem.

Definition 1. Let the value of a given variable seen by a thread at a given time be the value that would result from a read of that variable by that thread at that time.

Theorem 1. *Zeros of the slack matrix can be processed in parallel as long as: (S1) Only one prime per line is stored. (S2) To check if the zero is covered, the column is checked before the row. (S3) The cover row operation is seen before the uncover columns operation. (S4) A zero A on the same line as another zero B sees all the changes in covered rows and columns made by B before or at the same time as any other zero C. (S5) Repeated processing and synchronization guarantees that all the threads eventually read the values written by other threads that can affect their behavior unless the processing is finished or interrupted because one or more alternating paths were found. This corresponds to the pseudocode in Algorithm 2 using the data structures in Section 4.1.*

The ordering in S2 is guaranteed due to dependencies, and the ordering in S3 is guaranteed using a memory fence. S4 can be guaranteed by making sure the covered_row and covered_column are treated as volatile in step 4 as in the proposed algorithm. This implies that a device memory fence should be used to guarantee S3. The zeros can be distributed through GPU blocks in any way.

Another way to guaranteed S4 is to: (1) split slack matrix by the lines so that zeros on the same line are on the same GPU block; (2) use a block memory fence; (3) move covered_row and covered_column to shared memory in step 4; (4) make covered_row and covered_column volatile. In this case, S3 can be guaranteed by the block memory fence. This implementation is further discussed in [Appendix B](#).

Definition 2. Processing a zero consists in priming it, covering the row and uncovering the column of the start on the same line if any; if not it signaling to go to step 5.

Definition 3. Checking a zero consists on seeing if it is covered and, if not, processing it.

Definition 4. The parallel implementation runs the pseudo-code in Algorithm 2 in a CUDA machine using the data structures defined in Section 4.1.

Proof. In order to prove the theorem, it will be shown that there is one serial implementation that produces the same results as the run of the parallel implementation, namely the covered rows, columns, and the primes. Note that different runs of the parallel implementation may produce different results and correspond to different serial implementations.

The parallel implementation consists in multiple kernel runs as seen in Algorithm 2. It will be shown that each kernel run has a serial equivalent that produces the same results. Then since the kernels calls are serialized the global algorithm will also have a serial equivalent that is just the concatenation of each of the serial equivalents. Also, the termination conditions are the same, i.e. there are no more uncovered zeros, or there is a signal to go to step 5, so both implementations will produce the same results. The proof proceeds to show that each kernel run has a serial equivalent.

When a zero is processed, it (actually its thread) may uncover or cover other zeros. We model this operation as sending messages to the other zeros, the messages: “you were uncovered” and “you were covered”. The sending of the message corresponds to a write in a variable by the sending thread. The receiving corresponds to a read of the updated versions of the same variable by the receiving thread when processing the corresponding zero. With this model, the receiving of the messages may be delayed arbitrary (until a synchronization operation where the threads are guaranteed to see the changes made by other threads).

After a zero gets the message “you were covered” it will not do anything else, since once one zero is covered (it is always by a row) it cannot be uncovered. After a zero gets the message “you were uncovered” it is processed, and will not do anything else because it is also covered at the end. A directed graph can be built where the edges are the messages “you were uncovered” that were received by the zeros. Note that each zero can only receive one message. There may be some ambiguity in selecting which zero sent a message if there were several writes of the same value to the same variable before it was read by the destiny. In this case, any of the senders can be selected. Call this graph the processing order graph (POG). Notice that the POG is a tree and that all the zeros in the POG are processed in the parallel implementation.

In the following, let A be a zero, then A_i represents the same zero, and i is the line of the zero. Also, let A and B be vertices in some directed graph G , then $A \rightarrow B$ (A connects to B) means that there is a path from A to B in G .

Let A and B be vertices of a graph G then let joining B to A in G be: (1) moving to A all the edges that connect to B ; (2) removing B and any duplicates. Now, let the SPOG be the graph obtained by the following procedure. For each line join all the zeros of the line in the POG to the zero that was primed on the line. Finally call, POG_k the intermediate graphs that were generated in the procedure, where POG_0 is equal to the POG and POG_{k+1} is the graph obtained from POG_k by joining two zeros from the same line A_i and B_i . Let $SPOG = POG_L$.

We now discuss some properties of POG_k for any k .

Lemma 1. *For any k , there are no four zeros A_i, B_i, C_j, D_j in the POG_k from two different lines i and j such that C_j connects to A_i and B_i connects to D_j . In short:*

$$\forall k, \nexists \{A_i, B_i, C_j, D_j\} \in POG_k, C_j \rightarrow A_i \wedge B_i \rightarrow D_j \quad (A.1)$$

Table A.5
Threads Parallel Execution Diagram.

C_j	A_i	B_i	D_j
CR			
UC			
	CC?	CR	
	CR?	UC	
			CC?
			CR?

Proof. First, this will be proved for $k = 0$, i.e. for the POG (A). Then it will be proved by induction for any k (B).

A - For the POG) In the code for the checking of each zero there are the following atomic operations: cover column read (CC?), cover row read (CR?), cover row write (CR), uncover column write (UC). S2 and S3 impose some ordering on this operations. As seen by all the threads, we can say that (see Table A.5):

1. UC of B_i was before the CC? of D_j , since it got its uncover message from B_i .
2. The CR of B_i was before the UC of B_i because of S3.
3. The CR? of A_i is before the CC? of D_j . If it was after at time t then:
 - (a) At time t , D_j would have seen the UC of B_i as in 1.
 - (b) A_i would have seen the UC of B_i because of S4. If D_j saw it so has A_i .
 - (c) A_i would have seen the CR of B_i because of S3.
 - (d) A_i would not be processed since B_i covers A_i , but both A_i and B_i are processed.

Note, that is not always true that CR? of A_i is before CR of B_i because for instance there may be a delay in the memory write, but once D_j sees the UC of B_i then A_i must also see CR of B_i because of S4 and S3 as discussed.

4. The CC? of A_i was before the CR? of A_i because of S2.
5. The UC of C_j was before the CC? of A_i because C_j got its uncover message from A_i .
6. D_j sees the CR of C_j because: A_i has seen UC of D_j ; S3; S4; D_j CR? is after A_i CC? and C_j and D_j are on the same line.

Now this implies that D_j was covered and could not be processed. This is summarized in Table A.5.

B - Induction) It is required to prove that,

$$\begin{aligned} \nexists \{A_i, B_i, C_j, D_j\} \in \text{POG}_k, C_j \rightarrow A_i \wedge B_i \rightarrow D_j \Rightarrow \\ \nexists \{A_i, B_i, C_j, D_j\} \in \text{POG}_{k+1}, C_j \rightarrow A_i \wedge B_i \rightarrow D_j \end{aligned} \quad (\text{A.2})$$

By logic transposition this is equivalent to,

$$\exists \{A_i, B_i, C_j, D_j\} \in \text{POG}_{k+1}, \dots \Rightarrow \exists \{A_i, B_i, C_j, D_j\} \in \text{POG}_k, \dots \quad (\text{A.3})$$

Going from POG_k to POG_{k+1} corresponds to joining to zeros into one (by joining edges that connect to the zeros). The reverse operation consists in splitting one zero into two on the same line. It can be easily seen that by splitting each of the zeros in $\{A_i, B_i, C_j, D_j\}$ always results that one of the zeros that results can be used in place of the original zero to satisfy the condition. \square

Lemma 2. For any k , in the POG_k there are no two zeros A_i, B_i , from the same line i such that A_i connects to B_i . In short:

$$\forall k, \nexists \{A_i, B_i\} \in \text{POG}_k, A_i \rightarrow B_i \quad (\text{A.4})$$

Proof. First, this will be proved for $k = 0$, i.e. for the POG. Then it will be proved by induction for any k . For the POG any zero B_i that receives an uncover message that comes directly or indirectly from another zero A_i must first receive the cover message as

shown in the following. Let $\{l = 0 \dots m, C^l\}$ be the set of zeros that connect A_i to B_i in the POG, with $C^0 = A_i$ and $C^m = B_i$. S4 implies that any B_i must see the cover message sent by A_i (CMA) at the same time or before C^1 . C^1 sees CMA before it sees its uncover message because of S3. The time that C^l sees the uncover message increases with l because they depend on each other. So, for $l = m$, B_i must see the CMA before its uncover message. B_i cannot receive the uncover message because it is already covered so it cannot be in the POG.

Now the statement will be proved by induction for any k . It is known that there is no indirect path from A_i to B_i because of Lemma 1 by making $A_i = B_i$ in the lemma. It remains to be shown that there is no direct connection. When generating POG_{k+1} from POG_k , C_j is joined to D_j on the line j . In this operation, new edges are added to D_j . Call the set of these edges E . None of the edges in E will connect a zero A_i on the same line to another zero B_i on the same line because: (1) For every edge x in E one of its vertices was moved while the other remained the same, but the moved vertex moves to another zero on the same line, so the lines of the zeros of the edge remain the same. (2) This did not happen in POG_k . \square

Lemma 3. For any k , all the zeros in POG_k are reachable through POG_k starting from the set of initially uncovered zeros.

Proof. First, this will be proved for $k = 0$, i.e. for the POG. Then it will be proved by induction for any k .

For the POG, all the zeros are reachable through the POG starting from the set of initially uncovered zeros, because: (1) every zero in the POG is an initially uncovered zero or has an edge that connects it to another zero that uncovers it, otherwise it will not be in the POG; (2) The POG starts at the uncovered zeros; (3) For every zero it is possible to form a backward path through the POG until the start, that is not circular because the POG is a tree and is finite because the POG is finite.

Now the statement will be proved by induction for any k . When generating POG_{k+1} from POG_k , B_i is joint to A_i on the line i . For any C_j it is known that: there is a path on POG_k from start to C_j , $0 \rightarrow C_j$, that does not pass through A_i or B_i (C_j is above A_i and B_i); or A_i connects to C_j (C_j is below A_i); or B_i connects to C_j (C_j is below B_i). Note the join operation removes B_i . First note that after this A_i is still reachable because the path from the start to A_i does not pass through B_i , otherwise B_i would connect to A_i and this is not true because of Lemma 2. Now, if C_j is not below B_i then nothing is changed; the same path as in POG_k and be used for POG_{k+1} . If C_j is below B_i then, the new path to C_j is simply by the union of the path to A_i with the previous path from B_i to C_j . \square

Definition 5. A kernel serial equivalent consists in going through the SPOG starting at the set of initially uncovered zeros, in some order, and checking each zero using a serial computer.

Now it will be proved that a kernel run produces the same results as the kernel serial equivalent.

First, all the zeros in the SPOG are checked in the serial equivalent, because all are reachable using the SPOG by Lemma 3. Second, all the zeros in the SPOG are processed in the serial equivalent because none is covered. This is since their column is uncovered by the zero at the start of the edge they were reached by. Their row is not covered because there is only one zero per line in the SPOG, and rows are covered by zeros on the same row.

Third, the covered columns and covered rows are the same in the kernel run and serial equivalent because: (a) Processing two or more zeros on the same line produces the same covered columns and covered rows as only one. (b) For every zero that

is processed in the kernel (in the POG), there is one zero that is processed in the serial equivalent (in the SPOG) on the same line, that does the same. (c) The order of processing the zeros is not important when regarding covered rows and columns, because rows are only covered and columns are only uncovered. (d) Finally processing the same set of zeros in parallel or in series gives the same results regarding covered rows and column because there are no dependencies between the operations. This is because of (c) and because there are no reads.

Fourth, the primed zeros are the same. This is because the set of primed zeros in the kernel is equal to the set of zeros in the SPOG (all the other were removed when building it), and all these zeros (and only these) are processed and primed in the serial equivalent.

Finally, it remains to discuss one of the termination conditions, namely the signal to go to step 5. If this signal is activated during the serial equivalent run, then this implementation should stop, and some of the zeros of in the SPOG would not be processed. However, this signal indicates that an alternating path was already found so that the search can stop, but it is not required that it stops immediately, so in this implementation, it simply continues until all the threads see the signal. □

Appendix B. Other implementations of step 4

The proposed algorithm results in a very fast implementation of step 4, for several reasons: (1) It allows to split the graph through several blocks. (2) It reduces global synchronization. (3) It allows communication between the blocks through the volatile global variables, that further reduces global synchronization requirements. (4) It has lower block synchronization requirements. In the GPUs tested global synchronization is very costly, so this made it our choice for this paper, but other options could also be considered: A, B, and C.

(A) *Split by the lines, all zeros in parallel, in shared memory.* In this option, the zeros are split by the lines, so that zeros from the same line are in the same block. At the beginning of the kernel the covered rows and covered columns are read from global memory to the shared memory of each block, and at the end of the kernel, they are written back to global memory. Note, that in order to correctly mix the results from different blocks only rows that are covered and columns that are uncovered are written, since each block can only cover rows and uncover columns. The resulting covered rows are the union of the covered rows in each block, and the resulting covered columns are the intersection of the covered columns in each block. Covered rows and covered columns variables should then be treated as volatile variables in shared memory. In order to guarantee S3 in Appendix A a block memory fence is used. This option as the advantage that it makes good use of shared memory, avoiding expensive global memory reads and writes during the duration of the kernel. However, it has a serious disadvantage that a global synchronization operation corresponding to a kernel stop and restart is required for the other blocks to become aware of any of the changes made by the other blocks since these are only in shared memory.

(B and C) *Split by the lines, one zero per line, in global memory or shared memory.* In the proposed implementation and option A all the zeros are processed in parallel. This is not case in options B and C. In both the zeros are split by the lines as in option A. However, in a B and C the kernel start by writing in parallel all the uncovered zeros to a single vector with one zero per line, followed by a block thread synchronization. The result of this operation is to select randomly (due to the race) one uncovered zero per line. Then the zeros from this vector are processed in parallel as in the other options. For this implementation the proof of correctness can be simplified but, but it requires one more

synchronization between the threads of the block and that the graph is split by the lines. In option B this is done using global memory as in the proposed implementation, while in option C with shared memory as in option A.

Note, that although the proposed implementation has 3 sync threads of block, the frequency of them can be greatly reduced for instance by looping the outer if in Algorithm 2. The sync threads in Algorithm 2 have two functions: (1) to make sure the kernel terminates once there are no more uncovered zeros, or there is a signal to go to step 5; (2) to guarantee that the GPU resources are distributed evenly by the threads. Both functions can be delayed without imparting the correctness of the algorithm. Note, that the signal to go to 5 means that an alternating path was found, but there is no need to stop the step 4 immediately.

Appendix C. Supplementary data

Supplementary material related to this article can be found online at <https://doi.org/10.1016/j.jpdc.2019.03.014>.

References

- [1] J. Munkres, Algorithms for the assignment and transportation problems, J. Soc. Ind. Appl. Math. 5 (1) (1957) 32–38.
- [2] D.P. Bertsekas, The auction algorithm for assignment and other network flow problems: A tutorial, Interfaces 20 (4) (1990) 133–149.
- [3] R. Jonker, A. Volgenant, A shortest augmenting path algorithm for dense and sparse linear assignment problems, Computing 38 (4) (1987) 325–340.
- [4] E.L. Lawler, Combinatorial Optimization: Networks and Matroids, Dover Books, 1976.
- [5] K. Date, R. Nagi, GPU-accelerated Hungarian Algorithms for the linear assignment problem, Parallel Comput. 57 (2016) 52–72.
- [6] E. Balas, D. Miller, J. Pekny, P. Toth, A parallel shortest augmenting path algorithm for the assignment problem, J. ACM 38 (4) (1991) 985–1004.
- [7] D.P. Bertsekas, D.A. Castanon, Parallel asynchronous Hungarian methods for the assignment problem, ORSA J. Comput. 5 (3) (1993) 261–274.
- [8] D.P. Bertsekas, D.A. Castañón, Parallel synchronous and asynchronous implementations of the auction algorithm, Parallel Comput. 17 (6) (1991) 707–732.
- [9] M. Sathe, O. Schenk, H. Burkhart, An auction-based weighted matching implementation on massively parallel architectures, Parallel Comput. 38 (12) (2012) 595–614.
- [10] M.M. Zavlanos, L. Spesivtsev, G.J. Pappas, A distributed auction algorithm for the assignment problem, in: Decision and Control, 2008. CDC 2008. 47th IEEE Conference on, IEEE, 2008, pp. 1212–1217.
- [11] C.N. Vasconcelos, B. Rosenhahn, Bipartite graph matching computation on GPU, in: International Workshop on Energy Minimization Methods in Computer Vision and Pattern Recognition, Springer, 2009, pp. 42–55.
- [12] A. Azad, M. Halappanavar, F. Dobrian, A. Pothén, Computing maximum matching in parallel on bipartite graphs: worth the effort? in: Proceedings of the 1st Workshop on Irregular Applications: Architectures and Algorithms, ACM, 2011, pp. 11–14.
- [13] M. Deveci, K. Kaya, B. Uçar, Ü.V. Çatalyürek, GPU Accelerated maximum cardinality matching algorithms for bipartite graphs, in: European Conference on Parallel Processing, Springer, 2013, pp. 850–861.
- [14] E. Lawler, J.K. Lenstra, A.R. Kan, D. Shmoys, The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization, Wiley and Sons New York, 1985.
- [15] H. Yin, H. Liu, An efficient multiuser loading algorithm for OFDM-based broadband wireless systems, in: Global Telecommunications Conference, 2000. GLOBECOM'00. IEEE, vol. 1, IEEE, 2000, pp. 103–107.
- [16] D.W. Pentico, Assignment problems: A golden anniversary survey, European J. Oper. Res. 176 (2) (2007) 774–793.
- [17] R.E. Burkard, M. Dell'Amico, S. Martello, Assignment Problems, Revised Reprint, SIAM, 2009.
- [18] N. Biggs, E.K. Lloyd, R.J. Wilson, Graph Theory, 1736–1936, Oxford University Press, 1976.
- [19] Nvidia, Compute Unified Device Architecture (CUDA) Programming Guide, 2007.
- [20] P.A.C. Lopes, S.S. Yadav, A. Ilic, S.K. Patra, HungarianGPU on GitHub, <http://github.com/paclopes/HungarianGPU>.
- [21] M. Harris, et al., Optimizing parallel reduction in CUDA, NVIDIA Dev. Technol. 2 (4) (2007).



Paulo A. C. Lopes was born in Lisbon, Portugal, in 1974. He received the Ph.D. degree in electrical engineering from Instituto Superior Técnico (IST), Lisbon, in 2003. Currently, he is an Assistant Professor at IST and a Researcher at Instituto de Engenharia de Sistemas e Computadores, Investigação e Desenvolvimento (INESC-ID), Lisbon. His current research interests are MIMO communications for 5G systems, Power Line Communications, Active Noise Control and Computer Architectures.



Aleksandar Ilic received his Ph.D. degree in electrical and computer engineering from Instituto Superior Técnico (IST), Universidade de Lisboa, Portugal, in 2014. He is currently an Assistant Professor in the Department of Electrical and Computer Engineering of IST and a Senior Researcher of the Signal Processing Systems Group (SiPS), Instituto de Engenharia de Sistemas e Computadores R\&D (INESC-ID). His research interests include high-performance and energy-efficient computing and modeling on parallel heterogeneous systems.



Satyendra Singh Yadav was born in Tikamgarh (India) in 1991. He received his Bachelor of Engineering in Electronics and Communication from RITS—Bhopal, affiliated to Rajiv Gandhi Proudyogiki Vishwavidyalaya (RGPV) a State University of Madhya Pradesh (India), in 2012. In 2012 he joined National Institute of Technology, Rourkela (India), where he is working toward the Ph.D. degree. From Oct-2015 to June 2016, he was with Instituto de Engenharia de Sistemas e Computadores Investigação e Desenvolvimento (INESC-ID), Instituto Superior Técnico, Lisbon, Portugal, under India–EU NA-



Sarat Kumar Patra received B.sc. (Engg.) in Electronics and Telecommunication Eng and M.Sc. (Eng) in Electronics System and Communication specialization from VSSUT, Burla and NIT Rourkela respectively. He received Ph.D. from University of Edinburgh, UK in 1998. His research interests include adaptive signal processing, fuzzy systems, wireless communication and related areas. Prof. Patra is a life member of the Indian Society of Technical Education, Institution of Electronic and Telecommunication Engineers, India; Institute of Engineers, India; Computer Society of India. He is also a

senior member of IEEE since 2007. Currently he is professor at Indian Institute of Information Technology, Vadodara, India. Dr. Patra had supervised number of Ph.D.s and published more than 170 papers in journals and conference proceedings.

MASTE Mobility Project. His research interests include Parallel algorithms for wireless communications systems as well as Graphics Processing Unit (GPU) computing. Since 2014 he is a student member of IEEE.