Parallel Computing 57 (2016) 52–72

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

GPU-accelerated Hungarian algorithms for the Linear Assignment Problem

Ketan Date, Rakesh Nagi*

Department of Industrial and Enterprise Systems Engineering, 117 Transportation Building, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA

ARTICLE INFO

Article history:
Received 1 December 2014
Revised 8 April 2016
Accepted 19 May 2016
Available online 20 May 2016

Keywords:
Linear assignment problem
Parallel algorithm
Graphics processing unit
CUDA

ABSTRACT

In this paper, we describe parallel versions of two different variants (classical and alternating tree) of the Hungarian algorithm for solving the Linear Assignment Problem (LAP). We have chosen Compute Unified Device Architecture (CUDA) enabled NVIDIA Graphics Processing Units (GPU) as the parallel programming architecture because of its ability to perform intense computations on arrays and matrices. The main contribution of this paper is an efficient parallelization of the augmenting path search phase of the Hungarian algorithm. Computational experiments on problems with up to 25 million variables reveal that the GPU-accelerated versions are extremely efficient in solving large problems, as compared to their CPU counterparts. Tremendous parallel speedups are achieved for problems with up to 400 million variables, which are solved within 13 seconds on average. We also tested multi-GPU versions of the two variants on up to 16 GPUs, which show decent scaling behavior for problems with up to 1.6 billion variables and dense cost matrix structure.

1. Introduction

The objective of the linear assignment problem (LAP) is to assign n resources to n tasks such that the total cost of the assignment is minimized. The mathematical formulation for the LAP can be written as follows:

$$\min \quad \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij}; \tag{1}$$

$$\text{s.t.} \sum_{j=1}^{n} x_{ij} = 1 \quad \forall i = 1,\ldots,n; \tag{2}$$

$$\sum_{i=1}^{n} x_{ij} = 1 \quad \forall j = 1,\ldots,n; \tag{3}$$

$$x_{ij} \in \{0,1\} \quad \forall i,j = 1,\ldots,n. \tag{4}$$

* Corresponding author. Tel.: +1-217-244-3848; fax: +1-217-244-5705.
E-mail addresses: date2@illinois.edu (K. Date), nagi@illinois.edu (R. Nagi).

http://dx.doi.org/10.1016/j.parco.2016.05.012
0167-8191/

The decision variable $x_{ij} = 1$, if resource i is assigned to task j and 0 otherwise. Constraints (2) and (3) enforce that each resource should be assigned to exactly one task and each task should be assigned to exactly one resource. c_{ij} is the cost of assigning resource i to task j, and $\mathbf{C}_{n\times n} = \left[c_{ij}\right]$ is the cost matrix of the LAP.

LAP is one of the most well-studied optimization problems that can be solved in polynomial time. Until now, many efficient sequential algorithms have been proposed in the literature. These algorithms can be classified into three main classes (Burkard and Çela [8], Jonker and Volgenant [13]): (1) *Linear programming based algorithms*, which involve variants of the primal and dual simplex algorithms; (2) *Primal-dual algorithms* such as the famous Hungarian algorithm (Kuhn [16]) and the Auction algorithm (Bertsekas [4]); and (3) *Dual algorithms* such as the successive shortest path algorithm (Jonker and Volgenant [13]). Due to their polynomial worst-case complexity, the primal-dual and shortest path algorithms generally outperform the simplex-based algorithms. Several variations of the Hungarian and the shortest path algorithms have been proposed in the literature, for improving their execution time (Jonker and Volgenant [12], [14], Volgenant [30]). The theoretical complexity of the most efficient implementation of the primal-dual or shortest path algorithms is $O(n^3)$, where n is the number of resources or tasks.

Owing to their cubic worst-case complexity, sequential algorithms can prove to be a significant bottleneck for large instances of the LAP. This calls for the development of a parallel algorithm, which can take advantage of a specific architecture and divide the work among multiple processors, to alleviate the computational burden. Until now many parallel versions of the aforementioned sequential algorithms have been proposed which include parallel asynchronous version of the Hungarian algorithm (Bertsekas and Castañon [6]); parallel version of the shortest path algorithm (Balas et al. [2], Storøy and Sørevik [28]); and parallel synchronous and asynchronous versions of the Auction algorithm (Bertsekas and Castañon [5], Buš and Tvrdík [9], Naiem et al. [22], Sathe et al. [26], Wein and Zenios [31]). An empirical analysis of the sequential and parallel versions of the Auction and shortest path algorithms was performed by Kennington and Wang [15]. All the above parallel algorithms were designed for prevalent parallel computing architectures and they were shown to achieve significant speedups.

In recent years, there have been significant advancements in the graphics processing hardware. Since graphics processing tasks generally require high data parallelism, the GPUs are built as compute-intensive, massively parallel machines, which provide a cost-effective solution for high performance computing applications. Vasconcelos and Rosenhahn [29] developed a parallel version of the synchronous Auction algorithm for a single GPU. The authors tested the algorithm on problem instances with up to 16 million variables, which gets automatic scalability through CUDA with increasing number of GPU cores. Roverso et al. [25] developed a GPU implementation of the *deep greedy switching* (DGS) heuristic of Naiem and El-Beltagy [21], for solving the LAP under real-time constraints. It was shown that the heuristic sacrifices optimality in favor of significant speedup, on problem instances with up to 100 million variables.

In this paper, we are proposing parallel versions of two variants of the Hungarian algorithm, specifically designed for the CUDA enabled NVIDIA GPUs. We have chosen to parallelize the Hungarian algorithm, mainly because it operates on the cost matrix of the LAP and the GPUs are well suited for performing intense computations on arrays and matrices. Our main contribution is an efficient algorithm for the augmenting path search phase, which happens to be the most time intensive phase in the Hungarian algorithm. The prominent feature of our algorithm is that it takes advantage of the race condition to generate multiple vertex-disjoint augmenting paths, which can be used simultaneously to improve the current solution. We show that this parallelization leads to a dramatic reduction in the execution time, for both small and large sized problem instances. LAPs serve as sub-problems to many NP-hard optimization problems such as the Traveling Salesman Problem (TSP), the Quadratic Assignment Problem (QAP), and the Generalized Assignment Problem (GAP). Finding good solutions to these problems generally requires solving multiple LAPs in an iterative fashion. Therefore, having a fast, scalable, and cost effective LAP solver is extremely important. We believe that our GPU-accelerated algorithms stand true on all the three requirements.

The rest of the paper is organized as follows. In Section 2, we briefly describe the two variants of the sequential Hungarian algorithm. In Section 3, we describe some preliminaries, which will be useful in the development of the parallel algorithms. In Sections 4 and 5, we describe the various stages of our parallel algorithms, and their implementation on single and multi-GPU architectures. In Section 6, we present the experimental results on randomly generated problem instances. Finally in Section 7, we conclude the paper with a summary.

2. Sequential Hungarian algorithm

The Hungarian method developed by Kuhn [16] was the first systematic approach for finding the optimal solution to an LAP. Although, the algorithm is primarily based upon the works of Hungarian mathematicians König and Egerváry, the main idea behind the algorithm can be better explained with the help of linear programming duality (Bazaraa et al. [3], Nering and Tucker [23]). The dual of the assignment problem (1)–(4) can be written as follows:

$$\max \quad \sum_{i=1}^{n} u_i + \sum_{j=1}^{n} v_j; \tag{5}$$

$$\text{s.t.} \quad u_i + v_j \le c_{ij} \quad \forall i, j = 1, \ldots, n; \tag{6}$$

$$u_i, v_j \sim \text{unrestricted} \quad \forall i, j = 1, \ldots, n; \tag{7}$$

where, u_i and v_j are the dual variables corresponding to each constraint of the primal problem. Using the Karush-Kuhn-Tucker complimentary slackness condition for the optimal solution, we can write:

$$(c_{ij} - u_i^* - v_j^*)x_{ij}^* = 0. \tag{8}$$

Thus, if we find values for the dual variables u_i and v_j such that slack variables $c_{ij} - u_i - v_j = 0$, then the corresponding x_{ij} can be set to 1 (i.e., resource i can be assigned to task j), as long as they are present in independent rows and columns (necessary condition for primal feasibility). If the zero-valued slack variables are not independent, then we need to update the corresponding dual variables and find a new solution, which satisfies this condition. Thus, we start from dual feasibility and iteratively achieve primal feasibility.

Based on the above result (and the theorems by König and Egerváry), the Hungarian algorithm operates in two stages. In the first stage ("augmenting path search"), the algorithm finds the maximum matching corresponding to the edges with $c_{ij} - u_i - v_j = 0$, by building a directed tree rooted at an unassigned row, potentially ending at an unassigned column, and alternating between assigned and unassigned edges. If the alternating tree manages to terminate at an unassigned column, then it can potentially be used to increase the total number of assignments by one. If the maximum matching found at the end of this stage equals the total number of rows (or columns), the algorithm stops with the optimal assignment. Otherwise the second stage ("dual update") is executed, in which the dual variables are modified to introduce at least one new edge with zero slack. The algorithm continues to iterate between these two stages until an optimal solution is found.

We will now describe the two variants of the Hungarian algorithm: (1) The "classical" Kuhn-Munkres variant developed by Munkres [20]; and (2) The "alternating tree" variant developed by Lawler [17].

2.1. Data structures

The following data structures are used in this implementation.

1. *Cost matrix* (**C**): This matrix is stored as an array of n^2 integers (or doubles), in row-major order.
2. *Row/column assignment arrays* (A_r/A_c): Each of these arrays is stored as an array of n integers. They are used for recording the row and column assignments, with -1 as the sentinel value. $A_r[i] = j$ indicates that row i is assigned to column j; and $A_c[j] = i$ indicates that column j is assigned to row i.
3. *Row/column cover arrays* (V_r/V_c): Each of these arrays is stored as an array of n booleans. They are used for recording the row and column covers. $V_r[i] = 1$ indicates that row i is covered, and 0 indicates otherwise. Column cover array V_c follows a similar convention.
4. *Row/column dual variable arrays* (D_r/D_c): Each of these arrays is stored as an array of n doubles. They are used for recording the dual variable values corresponding to the rows and columns.
5. *Column slack variable array* (*slack*): This array is stored as an array of n doubles. It is used to store the minimum slack for each column, and it is only used in the alternating tree variant.
6. *Row/column predecessor arrays* (P_r/P_c): Each of these arrays is stored as an array of n integers. They are used for recording the predecessor indices of the rows and columns, with -1 as the sentinel value. They are primarily used during the augmenting path search phase of the algorithm (see Section 4.3).
7. *Row/column successor arrays* (S_r/S_c): Each of these arrays is stored in the device memory as an array of n integers. They are used for recording the successor indices of the rows and columns, with -1 as the sentinel value. They are also used during the augmenting path search phase of the algorithm.

2.2. Classical Hungarian algorithm

The classical variant of the Hungarian algorithm was proposed by Munkres [20] which systematizes the Hungarian method of Kuhn [16]. The pseudo-code for this variant is presented below. In each iteration, the algorithm either increases the number of assignments by one or introduces new edges with slack $c_{ij} - u_i - v_j = 0$ (each of these steps has complexity of $O(n^2)$). An adjacency list is maintained during each iteration to store the edges with zero slack, which is modified/recreated after the dual update step. Since there are n^2 elements in the cost matrix, the "search" and "update" steps could be executed at most n^2 times, and therefore the classical variant has complexity of $O(n^4)$.

```
algorithm classical_hungarian
input: Matrix C
output: Optimal assignments A_r and A_c
begin
    /* Initial reduction */
    foreach i ∈ {1, ···, n} do D_r[i] ← min_j{C[i, j]};                    /* row reduction */
    foreach j ∈ {1, ···, n} do D_c[j] ← min_i{C[i, j] − D_r[i]};        /* column reduction */
    repeat
      /* Optimality check */
      match_count ← 0; [*** check ***]
      reset V_r, V_c, P_r, P_c to "sentinels";
```

```
      foreach i ∈ {1, ···, n} do
        if A_r[i] ≠ −1 then
            V_r[i] ← 1;
            match_count ← match_count + 1;
        end
      end
      if match_count = n then go to exit;
      /* Augmenting path search */
      ST ← ∅;                                                          /* stack */
      foreach i ∈ {1, ···, n} do [*** search ***]
        if V_r[i] = 0 then ST.push(i);
        Z[i] ← ∅;                                  /* initialize adjacency list for row i */
        foreach j ∈ {1, ···, n} do
          if C[i, j] − D_r[i] − D_c[j] = 0 then Z[i].push(j);
        end
      end
      while ST ≠ ∅ do
        i ← ST.top();
        ST.pop();
        while Z[i] ≠ ∅ do
          j ← Z[i].front();
          Z[i].pop();
          i_new ← A_c[j];
          if i_new = i then continue;                          /* continue on to next j */
          if V_c[j] = 0 then                                   /* if column is uncovered */
            P_c[j] ← i;                                        /* update predecessor index */
            if i_new = −1 then                                 /* unassigned column */
              augment(j);
              go to check;
            else
              ST.push(i_new);
              P_r[i_new] ← j;                                  /* update predecessor index */
              V_r[i_new] ← 0;                                  /* uncover the row */
              V_c[j] ← 1;                                      /* cover the column */
            end
          end
        end
      end
      update();
      go to search;
    end [*** exit ***]
end
/* Procedure for augmenting the current assignments by 1 */
procedure augment
input: Unassigned column j, Row predecessors P_r, Column predecessors P_c
output: Updated assignment arrays A_r and A_c
begin
    c_cur ← j;
    r_cur ← −1;
    while c_cur ≠ −1                    /* repeat until current row has no predecessor */
      r_cur ← P_c[c_cur];
      A_r[r_cur] ← c_cur;
      A_c[c_cur] ← r_cur;
      c_cur ← P_r[r_cur];                                  /* update current column index */
    end
end
/* Procedure for updating the dual variables */
procedure update
input: Cover arrays V_r and V_c
output: Updated dual variable arrays D_r and D_c
begin
```

```
    θ ← ∞;
    foreach i ∈ {1, ···, n} do
      if V_r[i] = 0 then θ ← min{θ, min_{j|V_c[j]=0}{C[i, j] − D_r[i] − D_c[j]}};
    end
    foreach k ∈ {1, ···, n} do
      if V_r[k] = 0 then D_r[k] ← D_r[k] + θ/2; else D_r[k] ← D_r[k] − θ/2;
      if V_c[k] = 0 then D_c[k] ← D_c[k] + θ/2; else D_c[k] ← D_c[k] − θ/2;
    end
  end
```

2.3. Alternating tree Hungarian algorithm

This alternating tree variant of the Hungarian algorithm was proposed by Lawler [17] which improves the performance of the classical variant with a smarter choice of data structures. The pseudo-code for this variant is presented below. During the execution, the predecessor information of the columns is updated dynamically, and therefore, it is not required to construct the adjacency list at the beginning of the "search" step. Additionally the algorithm maintains the minimum "slack" ($c_{ij} - u_i - v_j$) for each column, which reduces the complexity of the "dual update" step from $O(n^2)$ to $O(n)$. In this variant, the "search" step is executed exactly n times before an optimal solution is found, and therefore, the complexity of this variant is $O(n^3)$.

```
algorithm alternating_tree_hungarian
input: Matrix C
output: Optimal assignments A_r and A_c
begin
    execute initial_reduction;
    repeat
      execute optimality_check; [*** check ***]
      if match_count = n then go to exit;
      foreach j ∈ 1, ···, n do slack[j] ← ∞;
      /* Augmenting path search */
      ST ← ∅;                                                    /* stack */
      foreach i ∈ {1, ···, n} do
        if V_r[i] = 0 then ST.push(i);
      end
      while ST ≠ ∅ do [*** search ***]
        i ← ST.top();
        ST.pop();
        foreach j ∈ {1, ···, n} do
          if slack[j] > C[i, j] − D_r[i] − D_c[j] then
            slack[j] ← C[i, j] − D_r[i] − D_c[j];
            P_c[j] ← i;
          end
          if C[i, j] − D_r[i] − D_c[j] = 0 then
            i_new ← A_c[j];
            if V_c[j] = 0 then                          /* if column is uncovered */
              if i_new = −1 then                             /* unassigned column */
                augment(j);
                go to check;
              else
                ST.push(i_new);
                P_r[i_new] ← j;                    /* update predecessor index */
                V_r[i_new] ← 0;                              /* uncover the row */
                V_c[j] ← 1;                                  /* cover the column */
              end
            end
          end
        end
      end
      update_2();
      go to search;
    end [*** exit ***]
end
```

```
/* Procedure for updating the dual solution */
procedure update_2
input: Cover arrays V_r and V_c
output: Updated dual solution arrays D_r and D_c
begin
    θ ← min_j{slack[j] > 0};
    foreach k ∈ {1, ···, n} do
      if V_r[k] = 0 then D_r[k] ← D_r[k] + θ/2; else D_r[k] ← D_r[k] − θ/2;
      if V_c[k] = 0 then D_c[k] ← D_c[k] + θ/2; else D_c[k] ← D_c[k] − θ/2;
    end
    foreach j ∈ {1, ···, n} do
      if slack[j] > 0 then
        slack[j] ← slack[j] − θ;
        if slack[j] = 0 then ST.push(P_c[j]);
      end
    end
end
```

3. Preliminaries

In this section, we will first introduce the readers to GPU and CUDA architecture (as described in NVIDIA, CUDA C programming guide 5.0, NVIDIA Corporation [24]), and then explain some concepts which will be helpful in devising the parallel algorithm.

3.1. Introduction to GPU and CUDA

GPUs are predominantly used for processing and rendering high quality graphics on a computer display. A GPU is built around an array of multi-threaded *streaming multiprocessors* (SMs), each of which contains an array of processor cores. Each processor core is equipped with data processing transistors and on-chip shared memory, which has very low latency. The GPU itself has a global memory, which can be accessed by all SMs but it is slightly slower than the former one. Since a GPU has more number of transistors devoted for data processing than a CPU, it is suitable for parallel computations with high arithmetic intensity.

CUDA is a general purpose parallel programming platform developed by NVIDIA to take advantage of the compute engine in their GPUs. A CUDA program is divided into two parts: (1) *host code* which is executed on the CPU; and (2) *kernels*, which are executed on the GPU. Kernels are blocks of instruction which are executed by a number of threads in parallel. The threads are logically arranged into *blocks* and the blocks are logically arranged into a *grid*. Each block is randomly scheduled on any available multiprocessor. When the multiprocessor finishes processing that block, next block gets assigned to it, and thus the application gets automatic scalability with increasing number of processor cores.

3.2. Parallelization strategy

In the parallel algorithm(s) that we have implemented, each step of the sequential algorithm(s), described in Section 2, is executed on the GPU by one or more CUDA kernels. After the execution of each kernel, the control is given to the CPU, for coordinating the program flow. This also provides natural synchronization points in the parallel algorithm. We make the following observations in the sequential algorithm(s), which will provide insights into the parallelization strategy for each step.

1. The initial reduction, optimality check, and dual update steps can be easily parallelized and they possess a higher degree of granularity. It means that we can easily define one thread for each element of the cost matrix (or at least one thread per row/column), all of which can be processed simultaneously. Therefore these steps will benefit the most from parallelization on GPU.
2. During each iteration of the augmenting path search, we are interested in only a small fraction of elements. For example, the augmenting path search step in the classical variant operates only on $m \ll n^2$ zero-slack edges. Therefore, we need to create an array of these relevant elements so that each element can be processed by a single thread and proper utilization of the threads can be achieved.
3. Finally, the augmenting path search step itself is difficult to parallelize since we cannot avoid its iterative nature. However, it is an application of the parallel breadth-first-search algorithm, which can be implemented efficiently on a GPU (see Section 3.4).

To this end, we will describe the concepts of stream compaction and parallel breadth-first search algorithm, in the next two sections.

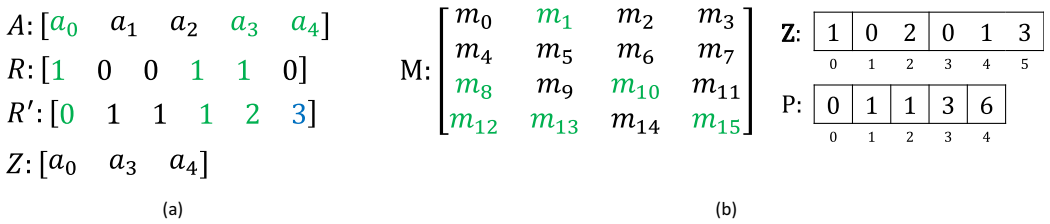

Fig. 1. (a) Array compression; (b) CSR matrix compression.

3.3. Sparse matrix representation

To construct an array of relevant elements in CUDA, we have used the concept of stream compaction as described by Harris and Sengupta [10]. The main idea behind this operation can be described as follows. Consider an input array A of size n, from which only $m < n$ elements are relevant. To compress these elements, we first define a "predicate" array R of size $n+1$. In this array we record "1" corresponding to the relevant elements and "0" corresponding to irrelevant elements. Then we perform a prefix-sum operation on this predicate array, which generates the scatter addresses of the relevant elements in the new array. The entry $R[n]$ represents the size m of the new array. Finally, we create an output array Z of size m and scatter the elements to the respective locations as indicated in the predicate array. Fig. 1(a) shows an example of array compression with 3 relevant elements.

In the classical variant, we also need to store the adjacency list of the edges with zero slack. For this purpose, we have used the *compressed sparse row* (CSR) storage format for matrix compression. Matrix compression can be achieved using the same operations mentioned above, with the exception that we store the column indices of the relevant elements, rather than elements themselves. For this purpose, we need two arrays: adjacency list **Z** and row pointer array P. Array **Z** is of size m, equal to the number of relevant elements, and it is used to store their column indices, traversed in row-major order. Array P is of size $n+1$, and the element $P[i]$ points in the array **Z**, the first relevant element of row i. The sub-array $\mathbf{Z}[P[i]]$ represents the adjacency list of row i, containing all the relevant elements from that row. Its size can be obtained by simply evaluating the expression $P[i+1]-P[i]$. The element $P[n]$ indicates the size of the array **Z**. Fig. 1(b) shows an example of the CSR arrays for a matrix **M**, containing six relevant elements. The adjacency list of row 2 begins from index 1 in **Z**, and it contains: $3-1=2$ elements, in columns 0 and 2 respectively.

Parallel prefix-sum is an important operation in array and matrix compression. Given an input array I, the prefix-sum operation produces an output array O in which each element is the sum of all the previous elements of the input array, i.e., $O[i]=\sum_{j=0}^{i-1} I[j]$. Blelloch [7] first developed an efficient parallel algorithm for the prefix-sum on vector processors, which has a work complexity of $O(n)$ and a step complexity of $O(\log n)$. Sengupta et al. [27] implemented this work-efficient algorithm on the NVIDIA GPU, which was shown to be significantly faster. Prefix-sum has important applications in sorting, stream compaction, lexical analysis, etc. In our implementation, we have used the prefix-sum function from the Thrust library for CUDA, developed by Hoberock and Bell [11]. The operation of compressing the zero-slack edges has a work complexity of $O(n^2)$ and a step complexity of $O(\log n)$.

3.4. Parallel breadth-first search algorithm

Breadth-first search (BFS) is a fundamental algorithm in graph traversal, for finding all vertices satisfying a particular property (Ahuja et al. [1]). The BFS algorithm traverses the graph from a *source* vertex, by successively marking the vertices along the outgoing edges and expanding the frontier. At the termination of this algorithm, we get a *tree* graph, rooted at the source vertex, with the property that a path between the source vertex and any other vertex in the tree, is a shortest path. The complexity of the sequential BFS algorithm is $O(n+m)$, where n is the number of vertices and m is the number of edges in the graph.

Parallelizing the BFS algorithm on a GPU is a non-trivial task. In the simplest implementation of the parallel BFS algorithm, the graph is represented as an adjacency list with n^2 elements. During the execution, the threads scan every edge or at least every vertex and expand the frontier by one hop during each iteration. Since there could be n iterations in the worst case, this parallelization has a quadratic complexity of $O(n^2)$. In most graphs, the number of edges is much smaller than n^2, due to which these quadratic parallelization strategies can prove to be extremely inefficient. For more details on the quadratic parallelization strategies, we direct the readers to Luo et al. [18].

Recently, Merrill et al. [19] has proposed a work-efficient parallel algorithm in which each vertex and each edge is scanned exactly once, and hence it has a linear complexity of $O(n+m)$. This parallel algorithm is probably the most efficient implementation of the BFS on a GPU. In this implementation, the graph is stored as a compressed adjacency list **Z**, using CSR format. During each iteration, the algorithm maintains two *frontier* arrays F_{in} and F_{out}. The array F_{in} contains the vertices which are currently "active," and it is initialized using the *source* vertex(s). The remaining vertices in **Z** are marked as "inactive." Each BFS iteration is carried out in the following two phases which are repeated until all the vertices are visited:

1. *Expansion*: In this phase, F_{out} is initialized with a size equal to the total number of neighbors of all the vertices in F_{in}. This operation is also known as *allocation*, which is another application of the parallel prefix-sum. The kernel is executed by defining one thread for each vertex in F_{in}. Each thread traverses its corresponding adjacency list in **Z** and *gathers* its neighbors into F_{out}, which serves as a staging ground for the new frontier.
2. *Contraction*: In this phase, F_{out} is compressed by removing the "visited" vertices. After compression, the array F_{out} represents the new frontier, which is 1-hop distance away from the vertices in F_{in}. Finally, all the vertices in F_{in} are marked as "visited" and they are removed from the array. The vertices from F_{out} are copied into F_{in}, their labels are changed to "active," and the algorithm returns to the expansion phase.

The motivation behind introducing the parallel BFS algorithm in this section is that the augmenting path search of the Hungarian algorithm is similar to constructing multiple trees rooted at some unassigned rows. For this step to have linear time complexity, we need to make sure that each vertex and each edge is scanned at most once. In the sequential algorithm(s), this is achieved with the help of queues and stacks, which are not easy to construct in CUDA. However, using the concept of stream compaction we can construct arrays that mimic the above data structures, for relatively lower computational cost. Thus, to efficiently parallelize the augmenting path search step (both in classical as well as alternating tree variant), we have used the concepts from the parallel BFS algorithm mentioned above. To the best of our knowledge, our algorithm is the first known application of a GPU-based parallel BFS in an LAP solver.

4. Accelerating the Hungarian algorithm

In this section, we will describe the specifics of parallelization for each step of the Hungarian algorithm. All the data structures mentioned in Section 2.1 remain the same and they are initialized in the device memory instead of the host memory, so as to minimize host-device memory transactions.

4.1. Initial reduction

In this step, an initial dual feasible solution is found by executing row and column reduction kernel (depicted in Algorithm 1) on the GPU. This kernel is executed with n threads, each corresponding to one row (or column) of the matrix

Algorithm 1: Initial reduction kernel.

```
Data: Matrix C
Result: Arrays D_r and D_c
parallel foreach i ∈ {1, ..., n} do
    D_r[i] ← min_j {C[i, j]} ;                    /* row reduction kernel */
end
synchronization ;
parallel foreach i ∈ {1, ..., n} do
    D_c[j] ← min_i {C[i, j] − D_r[i]} ;           /* column reduction kernel */
end
```

C. At the end of this step, we obtain a dual feasible solution corresponding to the arrays D_r and D_c. These kernels have a work complexity of $O(n^2)$. There is no transfer of data between host and device before and after the execution of this kernel.

4.2. Optimality check

This step is executed using the kernel shown in Algorithm 2, and it serves as an optimality check for the current assignment solution. Initially, the elements from the cover arrays V_r and V_c are reset to 0. Then the kernel is executed with n threads, each corresponding to one element of the assignment array A_r. Each thread checks if the corresponding row is assigned to a column, and if so, it covers that row in the row cover array V_r, and increments an integer variable **match_count**. The work complexity of this kernel is $O(n)$. After the execution of this kernel, we need to transfer the **match_count** (a single integer) from the device to host.

If all the rows are covered at the termination of this kernel (i.e., **match_count** $= n$), the algorithm is terminated with the optimal assignment corresponding to the arrays A_r and A_c. Otherwise, all the values in the predecessor/successor arrays P_r, P_c, S_r, and S_c are reset to -1, and we go to the augmenting path search step described in the next section. In the alternating tree variant, the *slack* array is reset to ∞.

4.3. Augmenting path search

This is the most important step of the Hungarian algorithm, in which an alternating tree is built starting from an unassigned row vertex and potentially ending at an unassigned column vertex, and alternating between assigned and unassigned edges. According to Balas et al. [2], the augmenting path search can be parallelized in two ways: (1) each processor

Algorithm 2: Optimality check kernel.

```
Data: Row assignment array A_r
Result: Row cover array V_r, match_count
parallel for i ∈ {1, ..., n} do
    if A_r[i] ≠ −1 then                                   /* row is assigned */
        V_r[i] ← 1 ;                                      /* update row cover */
        match_count ← match_count +1 ;                    /* atomic add */
    end
end
```

independently searches for an augmenting path from different unassigned vertices; and (2) several processors jointly attempt to find an augmenting path from the same unassigned vertex. The method that we are proposing can be considered as a hybrid approach, in which multiple CUDA threads jointly search for augmenting paths from all the unassigned rows, and they identify multiple vertex disjoint paths, taking advantage of the "race" condition, all of which can be used to augment the current solution. Although our method is specifically designed for the GPUs, it can be readily extended to multi-core CPUs using OpenMP directives, which adds another facet to our contribution.

The augmenting path search is executed in three phases: forward pass, reverse pass, and augmentation pass, which are described below.

4.3.1. Forward pass

The forward pass is a parallel, iterative BFS, rooted at all unassigned rows containing at least one zero-element. The forward pass algorithms for the classical and alternating tree variants are described below.

Forward pass in the classical variant.

1. Initially, the column indices of zero-slack edges are compressed into the adjacency list **Z**, using the CSR format. Since the matrix compression is an expensive operation, it is performed only if "dual update" was executed in the previous iteration and new zero-slack edges were introduced. Otherwise, the adjacency list from the previous iteration can be reused. This small modification leads to significant improvement in the execution time. After constructing the adjacency list, the indices of the unassigned rows having at least one neighbor column are marked as "active" and they are added to the frontier array F_{in}. The indices of rows with no neighboring columns are marked as "visited." All the remaining row indices are marked as "inactive." All the column indices are also marked as "inactive."
2. Next, the *expansion* phase of the BFS is executed with one thread for each element in F_{in}. Main steps of this phase are outlined in Algorithm 3. During the execution, each thread traverses the "inactive" column indices, from its adjacency list in **Z**; looks up the subsequent row indices from the assignment array A_c; and *gathers* these row indices into F_{out}. During its traversal, the thread updates the predecessor arrays (P_r, P_c), and the cover arrays (V_r, V_c); and marks the column indices as "visited," to prevent cycling. Unassigned column indices are marked as "reverse," which are possible candidates for the reverse pass. All the row indices in F_{in} are marked as "visited" and they are removed from the array.
3. Next, the *contraction* phase of the BFS is executed, in which F_{out} is compressed by removing any "visited" row indices. The remaining row indices from F_{out} are marked as "active;" they are copied into F_{in}; and the algorithm returns to the *expansion* phase with this new frontier. The two phases are repeated until no more "active" row indices can be found.
4. If there exists at least one column marked as "reverse," then the current solution can be improved by executing the reverse and augmentation passes, as explained in Sections 4.3.2 and 4.3.3. Otherwise, the dual solution needs to be updated to introduce new zero-slack edges, as explained in Section 4.4.

Forward pass in the alternating tree variant.

1. Initially, the unassigned row indices are marked as "active" and added to the frontier array F_{in}. All the remaining row indices are marked as "inactive."
2. Next, the *expansion* phase of the BFS is executed with one thread per column vertex (main difference between this variant and the classical one). Main steps of this phase are outlined in Algorithm 4. During the execution, each thread traverses the current frontier and updates the minimum "slack" value and corresponding predecessor row index for the column vertex. Then, the same thread looks up the subsequent row index from the assignment array A_c and marks it as "active" for the next frontier (if it is "inactive" in the current iteration). During this traversal, the thread updates the predecessor arrays (P_r, P_c), and the cover arrays (V_r, V_c); and marks the column indices as "visited," to prevent cycling. Unassigned column indices with zero slack are marked as "reverse," which are possible candidates for the reverse pass. All the row indices in F_{in} are marked as "visited" and they are removed from the array.
3. Next, the *contraction* phase of the BFS is executed, in which the "active" row indices are compressed into F_{in}; and the algorithm returns to the *expansion* phase with this new frontier. The two phases are repeated until no more "active" row indices can be found.

Algorithm 3: Forward pass expansion kernel in classical variant.

```
Data: Frontier array F_in, Adjacency list Z, Assignment array A_c, Cover arrays V_r and V_c
Result: Frontier array F_out, Modified predecessor arrays P_r and P_c, Modified cover arrays V_r and V_c
parallel foreach i ∈ F_in do
    foreach j ∈ Z_i do
        if V_c[j] = 0 then                                   /* column j is uncovered */
            P_c[j] ← i ;                                     /* update predecessor of column j */
            i_new ← A_c[j] ;                                 /* lookup assignment of column j */
            if i_new = i then continue ;                     /* continue on to next j */
            if i_new ≠ −1 then                               /* column j is assigned */
                P_r[i_new] ← j ;                             /* update predecessor of row i_new */
                V_r[i_new] ← 0 ;                             /* uncover row i_new */
                V_c[j] ← 1 ;                                 /* cover column j */
                if i_new not "visited" then
                    Mark i_new as "active" ;
                end
                Gather i_new into F_out ;
            else                                             /* column j is unassigned */
                Mark j as "reverse" ;                        /* reverse pass candidate */
            end
        end
    end
    Mark i as "visited" ;
end
```

4. Once again, if there exists at least one column marked as "reverse," then the current solution can be improved by executing the reverse and augmentation passes, as explained in Sections 4.3.2 and 4.3.3. Otherwise, the dual solution needs to be updated to introduce new zero-slack edges, as explained in Section 4.4.

Correctness of forward pass. At the termination of of the forward pass, we obtain one or more directed out trees, represented by the predecessor arrays P_r and P_c, each of which is: (a) rooted at unassigned rows, (b) ending at either assigned rows or unassigned columns, and (c) alternating between assigned and unassigned edges. These alternating trees exhibit a very important property, as proved by the following proposition.

Proposition 1. *The alternating trees produced by the forward pass algorithm are vertex-disjoint.*

Proof. This proposition can be proved using the structure of the graph containing assigned and unassigned zero-slack edges. We make the following important observations: (1) each column vertex with an incoming unassigned edge can have at most one outgoing assigned edge; (2) each row vertex with an incoming assigned edge can have multiple outgoing unassigned edges; (3) during the forward pass, only one of the predecessor indices of a column vertex will survive, due to the race condition. Therefore, the structure of any tree obtained during forward pass is such that each row can have at most one column as its predecessor and multiple columns as successors; while each column can have at most one row as its predecessor and one row as its successor.

Now, let us assume that two alternating trees T_1 and T_2 rooted at rows R_{i_1} and R_{i_2} ($R_{i_1} \neq R_{i_2}$) are not vertex-disjoint. It means that the two trees either merge at some common row vertex or column vertex. Let us also assume that the two trees merge at a common row vertex R_{i_k}, such that $R_{i_k} \in T_1$ and $R_{i_k} \in T_2$. It means that the row R_{i_k} must have two predecessor columns $C_{j_p} \in T_1$ and $C_{j_q} \in T_2$. However, the row R_{i_k} can have at most one predecessor. Therefore, either $R_{i_k} \in T_1$, or $R_{i_k} \in T_2$, and not both, which is a contradiction. If we assume that the two trees merge at a common column vertex, we arrive at a similar contradiction. Therefore, the trees T_1 and T_2 must be vertex-disjoint. □

The importance of having multiple vertex-disjoint trees can be explained as follows. If more than one of those trees contain at least one unassigned column as a leaf vertex, then we can get more than one alternating paths. All these paths can be used to increase the current number of assignments, as opposed to only one potential assignment per iteration in the sequential algorithm. Therefore, the parallel algorithm can converge to the optimal solution in fewer number of iterations, thereby reducing the overall execution time. After the execution of this kernel, we need to transfer a boolean flag from the device to host which indicates whether reverse pass or dual update should be executed next.

4.3.2. Reverse pass

The alternating trees obtained during the forward pass are vertex-disjoint, however, each tree can potentially have multiple unassigned columns as leaf vertices. Therefore, to identify alternating, vertex-disjoint paths, we execute the reverse pass

Algorithm 4: Forward pass expansion kernel in alternating tree variant.

Data: Frontier array F_{in}, Matrix **C**, Dual arrays D_r and D_c, Assignment array A_c, Cover arrays V_r and V_c, *slack* array
Result: Modified predecessor arrays P_r and P_c, Modified cover arrays V_r and V_c
parallel foreach $j \in \{1, \cdots, n\}$ **do**
  if $V_c[j] = 0$ **then** /* column j is uncovered */
    foreach $i \in F_{in}$ **do**
      if $slack[j] > \mathbf{C}[i, j] - D_r[i] - D_c[j]$ **then**
        $slack[j] = \mathbf{C}[i, j] - D_r[i] - D_c[j]$; /* update slack of column j */
        $P_c[j] \leftarrow i$; /* update predecessor of column j */
      end
      $i_{new} \leftarrow A_c[j]$; /* lookup assignment of column j */
      if *slack[j] = 0* **then**
        if $i_{new} \neq -1$ **then** /* column j is assigned */
          $P_r[i_{new}] \leftarrow j$; /* update predecessor of row i_{new} */
          $V_r[i_{new}] \leftarrow 0$; /* uncover row i_{new} */
          $V_c[j] \leftarrow 1$; /* cover column j */
          Mark i_{new} as "active" ;
        else /* column j is unassigned */
          Mark j as "reverse" ; /* reverse pass candidate */
        end
      end
    end
  end
  Mark i as "visited" ;
end

algorithm. To improve the thread utilization, we create a compressed array F_{rev} containing only those column indices which are labeled as "reverse" during forward pass. Then we execute the kernel by defining one thread per element of F_{rev}. The steps involved in the reverse pass are outlined in Algorithm 5. The work complexity of the reverse pass algorithm is $O(n)$

Algorithm 5: Reverse pass kernel.

Data: Array F_{rev}, Predecessor arrays P_r and P_c
Result: Modified successor arrays S_r and S_c
parallel foreach $j \in F_{rev}$ **do**
  $r_{cur} \leftarrow -1$;
  $c_{cur} \leftarrow j$;
  while $c_{cur} \neq -1$ **do** /* repeat until current row has no predecessor */
    $S_c[c_{cur}] \leftarrow r_{cur}$; /* update successor of current column index */
    $r_{cur} \leftarrow P_c[c_{cur}]$; /* update current row index */
    $S_r[r_{cur}] \leftarrow c_{cur}$; /* update successor of current row index */
    $c_{cur} \leftarrow P_r[r_{cur}]$; /* update current column index */
  end
  Mark r_{cur} as "augment" ; /* augmentation pass candidate */
end

per thread. There is no transfer of data between host and device before and after the execution of this kernel.

During the execution, each thread traverses the tree by looking up the predecessor indices of the rows and columns from the arrays P_r and P_c, and records the successor indices in the arrays S_r and S_c. At the termination, we obtain one or more directed paths, represented by the successor arrays S_r and S_c, each of which is: (a) rooted at an unassigned row, (b) ending at an unassigned column, and (c) alternating between assigned and unassigned edges. These paths are also vertex-disjoint, as proved by the following proposition.

Proposition 2. *The alternating paths produced by the reverse pass algorithm are vertex-disjoint.*

Proof. Consider two different trees T_1 and T_2 obtained during forward pass. From Proposition 1, we know that T_1 and T_2 are vertex-disjoint. Additionally, if each of those trees has only one leaf vertex, then the two paths $P_1 = T_1$ and $P_2 = T_2$ must be vertex-disjoint.

Now, consider a single tree with multiple leaf vertices, each of which is assigned to one thread. Each thread traverses the tree and marks the successor indices of the rows and columns. Due to the structure of the tree, any two paths can potentially converge at a row and the thread responsible for each path will try to update the successor index of that row. However, due to the race condition, only one thread will succeed, and therefore only one of the alternating paths will survive. From Proposition 1, this path must be vertex-disjoint from any paths arising from other alternating trees. □

4.3.3. Augmentation pass

In this step, the alternating paths obtained during the reverse pass are used to augment the current assignment solution. Again, we create a compressed array F_{aug} containing only those row indices which are labeled as "augment" during reverse pass. Then, we execute the kernel by defining one thread per element of F_{aug}. The steps involved in the augmentation pass are outlined in Algorithm 6. The work complexity of the augmentation pass algorithm is $O(n)$ per thread. There is no transfer

Algorithm 6: Augmentation pass kernel.

Data: Array F_{aug}, Successor arrays S_r and S_c
Result: Modified assignment arrays A_r and A_c
parallel foreach $i \in F_{aug}$ **do**
  $r_{cur} \leftarrow i$;
  $c_{cur} \leftarrow -1$;
  while $r_{cur} \neq -1$ **do** /* repeat until current column has no successor */
    $c_{cur} \leftarrow S_r[r_{cur}]$; /* update current column index */
    $A_r[r_{cur}] \leftarrow c_{cur}$; /* update row assignment */
    $A_c[c_{cur}] \leftarrow r_{cur}$; /* update column assignment */
    $r_{cur} \leftarrow S_c[c_{cur}]$; /* update current row index */
  end
end

of data between host and device before and after the execution of this kernel.

During the execution, each thread traverses a single path by looking up the successor indices from arrays S_r and S_c; and interchanges the assigned and unassigned edges along that path. At the termination, the number of assignments in the current solution will be increased by the total number of traversed paths, and we return to the optimality check in Section 4.2.

4.4. Dual solution update

In this step the dual solution is updated and new edges with zero slack are introduced. The algorithms for the two variants are described below.

Dual update for the classical variant. The dual update for classical variant is executed in two stages. Initially, Stage 1 kernel (depicted in Algorithm 7) is executed with n threads, each corresponding to one row of the matrix **C**. This kernel finds the minimum uncovered slack θ. This kernel has work complexity of $O(n^2)$. There is no transfer of data between host and device before and after the execution of this kernel.

Next, Stage 2 kernel (depicted in Algorithm 8) is executed with n threads, each corresponding to one row or column. The kernel updates the dual variables, which creates at least one new zero-slack edge. This kernel has a work complexity of $O(n)$. After termination of this kernel, the algorithm returns to the augmenting path search step in Section 4.3. There is no transfer of data between host and device before and after the execution of this kernel.

Dual update for the alternating tree variant. The dual update for the alternating tree variant is similar to that of the classical variant, with a few exceptions. Initially, the minimum non-zero slack θ can be found using a simple $O(n)$ reduction operation of the *slack* array, i.e., by computing $\theta = \min_j\{slack[j] > 0\}$. Once we have the θ value, then we can execute the kernel depicted in Algorithm 9 with n threads, which updates the dual variables and the column slacks. If some column has zero slack, then its predecessor row is marked as "active," and the algorithm returns to the augmenting path search step in Section 4.3. There is no transfer of data between host and device before and after the execution of this kernel.

4.5. Overall algorithm complexity

The overall complexity of the accelerated algorithms remain the same as their sequential counterparts, however, the work is split between multiple threads, which translates into significant parallel speedup. For non-pathological cases, the accelerated algorithms potentially find more than one augmenting paths per iteration, and therefore they rapidly converge to the optimal solution. In Section 6, we will empirically demonstrate the benefits of our accelerated algorithms over the sequential algorithm.

Algorithm 7: Dual update kernel in classical variant: Stage 1.

Data: Cost matrix **C**, Dual arrays D_r and D_c, Cover arrays V_r and V_c, Temp array U
Result: Minimum uncovered slack θ
parallel foreach $i \in \{1, \ldots, n\}$ **do**
  $U[i] \leftarrow \infty$;
  if $V_r[i] = 0$ **then** /* row i is uncovered */
    foreach $j \in \{1, \ldots, n\}$ **do**
      if $V_c[j] = 0$ **then** /* column j is uncovered */
        $U[i] \leftarrow \min\{\mathbf{C}[i, j] - D_r[i] - D_c[j], U[i]\}$
      end
    end
  end
end
if $tid = 0$ **then**
  $\theta \leftarrow \infty$; /* executed by a single thread with id $= 0$ */
  foreach $i \in \{1, \ldots, n\}$ **do**
    $\theta \leftarrow \min\{\theta, U[i]\}$; /* update the minimum */
  end
end

Algorithm 8: Dual update kernel in classical variant: Stage 2.

Data: Minimum uncovered slack θ, Cover arrays V_r and V_c
Result: Modified dual arrays D_r and D_c
parallel foreach $k \in \{1, \ldots, n\}$ **do**
  if $V_r[k] = 0$ **then**
    $D_r[k] \leftarrow D_r[k] + \frac{\theta}{2}$;
  else
    $D_r[k] \leftarrow D_r[k] - \frac{\theta}{2}$;
  end
  if $V_c[k] = 0$ **then**
    $D_c[k] \leftarrow D_c[k] + \frac{\theta}{2}$
  else
    $D_c[k] \leftarrow D_c[k] - \frac{\theta}{2}$
  end
end

Algorithm 9: Dual update kernel in alternating tree variant.

Data: Minimum uncovered slack θ, Cover arrays V_r and V_c, Predecessor array P_c
Result: Modified dual arrays D_r and D_c, Modified *slack* array
parallel foreach $k \in \{1, \ldots, n\}$ **do**
  if $V_r[k] = 0$ **then**
    $D_r[k] \leftarrow D_r[k] + \frac{\theta}{2}$;
  else
    $D_r[k] \leftarrow D_r[k] - \frac{\theta}{2}$;
  end
  if $V_c[k] = 0$ **then**
    $D_c[k] \leftarrow D_c[k] + \frac{\theta}{2}$
  else
    $D_c[k] \leftarrow D_c[k] - \frac{\theta}{2}$
  end
  if $slack[j] > 0$ **then**
    $slack[j] \leftarrow slack[j] - \theta$;
    if $slack[j] = 0$ **then** Mark $P_c[j]$ as "active";
  end
end

5. Multi-GPU implementation

We implemented both the classical and alternating tree variants of the Hungarian algorithm in a multi-GPU setting. In the single GPU implementation, it may become challenging to store the cost matrix in the GPU memory, for larger problems. One of the alternatives to overcome this problem is to split the cost matrix across multiple GPUs. Consequently, some of the steps of the algorithm can be performed in parallel on multiple GPUs and additional parallel speedup can be achieved. In this paper, we have used grid architecture with multiple compute nodes, each containing one CPU–GPU pair. Communication between the different CPUs is accomplished using *message passing interface* (MPI). The overall algorithmic scheme for this multi-GPU implementation is described below:

1. **Initialization:** The program is initialized with p MPI processes, equal to the number of nodes in the grid. It is assumed that one MPI process gets allocated to exactly one node. The node with rank 0 is chosen as the root. The rows of the cost matrix $\mathbf{C}$ are split evenly between all the devices in the grid, i.e., each device owns a sub-matrix $\mathbf{C}^i$ of size $\lceil \frac{n}{p} \rceil \times n$. For the alternating tree variant, we divide the columns of the cost matrix among the devices, instead of the rows.
2. **Initial reduction:** The row reduction step from Algorithm 1 can be trivially parallelized, and it is simultaneously executed by all the devices on their individual sub-matrices to obtain partial row dual arrays. These partial arrays are transferred to the host ($O(n/p)$ memory transfer) and gathered at the root to construct an array of global row duals. During the column reduction phase, each device independently finds the local column duals from the sub-matrix $\mathbf{C}^i$ and stores them in an array. These arrays are first transferred from device to host ($O(n)$ memory transfer); then they are gathered at the root using "MPI_Gather" operation; and finally global column duals are identified on the root by finding the minimum for each column. The arrays of global row and column duals are broadcast to all the nodes, which are then transferred to the corresponding devices ($O(n)$ memory transfer).
3. **Optimality check:** In this step, each device independently executes Algorithm 2 on the rows within its scope and counts the number of assigned rows. The total count is calculated at the root using "MPI_Reduce" operation, and it is broadcast to all the nodes. If all the rows are assigned, then the program is terminated, otherwise augmenting path search is executed.
4. **Augmenting path search:**
   (a) In the **classical variant**, the matrix compression operation is the most expensive step of the algorithm (as seen in Section 6), but it can be trivially parallelized. Each device independently compresses the zero-slack edges from each row, into a partial adjacency list $\mathbf{Z}^i$. These partial adjacency lists are first transferred to the host ($O(m/p)$ memory transfer) and then they are gathered at the root node to construct a complete adjacency list $\mathbf{Z}$. After matrix compression, the forward pass (including Algorithm 3) is executed only on the root node, as described in Section 4.3.
   (b) In the **alternating tree variant**, the matrix compression operation does not exist and the forward pass is the main bottleneck. Recall, that forward pass is an iterative BFS, in which the vertex frontier is expanded during each iteration. To parallelize this step, we divide the column vertices equally among the devices. The initial frontier consists of unassigned row vertices. Each device executes the *expansion* phase of the parallel BFS (Algorithm 4), producing a local frontier of row vertices. In the *contraction* phase, these local frontier lists are gathered at the root node using "MPI_Gather" operation and a global frontier is created. This frontier is broadcast to all the nodes for the next BFS iteration. Since this step requires $O(n)$ memory transfer between host and device plus $O(n)$ MPI communication during each iteration, the benefit of parallelization is outweighed by the communication and this particular implementation has poor scalability (as seen in Section 6).

   In both variants, if an augmenting path is found, then the current solution is improved by executing the reverse and augmentation passes (Algorithms 5 and 6) on the root node. For the alternating tree variant, the partial column predecessor arrays are gathered at the root node before executing the reverse pass. After the augmentation pass is executed, modified assignment arrays are broadcast to all the nodes and transferred to the corresponding devices ($O(n)$ memory transfer) and the algorithm returns to the optimality check. Otherwise, modified cover arrays are broadcast to all the nodes and transferred to the corresponding devices ($O(n)$ memory transfer), and then the dual update step is executed.
5. **Dual Update:** The dual update step can also be trivially parallelized. Initially, each device independently identifies the minimum non-zero slack (from the sub-matrix $\mathbf{C}^i$ for the classical variant using Algorithm 7 or from the column slack array for the alternating tree variant using the reduction operation). The root node reduces these local minima and identifies the global minimum slack θ. This value is broadcast to all the nodes, and it is used to update the dual variables on all devices (using Algorithm 8 or Algorithm 9). The algorithm then returns to the augmenting path search step. The memory transfer between host and device is $O(1)$.

We would like to point out that in the classical variant, the augmenting path search is performed on the root node, for which the adjacency list needs to be stored in the memory of the root GPU. Therefore, this approach is not completely immune to the memory restrictions for substantially large problems. One way to tackle this issue is to store the adjacency list in the CPU memory and copy it in the GPU memory as required. This approach, however, may add significant communication overhead, and alternative options need to be explored. The alternating tree variant does not have this memory restriction, which means that we can potentially solve problems of any size, provided we have sufficient number of GPUs. However, in this approach the forward pass requires synchronization after every BFS iteration, which introduces significant MPI communication overhead and limits the scalability of the algorithm. In a different architecture containing multiple GPUs

Table 1
Test results for small instances.

n	Obj Val	Time (s)			
		OMP-1	OMP-8	CU-CLASS	CU-TREE
500	568.0	0.07	0.13	0.51	0.45
1000	1161.0	0.31	0.34	0.60	0.50
1500	1730.3	0.79	0.66	0.68	0.70
2000	2352.3	1.55	1.03	0.84	0.82
2500	2861.0	2.35	1.36	0.86	0.87
3000	3467.3	3.92	1.93	0.98	1.32
3500	4066.0	5.41	2.50	1.11	1.48
4000	4715.7	7.48	3.21	1.19	1.56
4500	5291.7	9.27	3.79	1.24	1.59
5000	5812.0	11.64	4.35	1.39	1.70

Table 2
Number of assignments found during different stages for CU-TREE.

n	Initial	Assignments in Iterations						
	Assignments	[1–5]	[6–10]	[11–15]	[16–20]	[21–25]	[26–30]	[31–35]
1000	866	96	26	6	6			
2000	1745	177	47	18	10	3		
3000	2608	203	143	24	10	6	6	
4000	3473	254	184	52	21	11	4	1
5000	4316	354	231	61	10	22	4	2

on a single host, it might be possible to achieve good parallel scalability, with the help of *unified virtual addressing* (UVA) and *peer-to-peer* (P2P) memory access.

6. Computational testing

We implemented both variants in CUDA C programming language, and the tests were performed on the computational resources from the Blue Waters Super-computing Facility at the University of Illinois at Urbana-Champaign. The host code was executed on AMD Interlagos model 6276 processor, with 8 cores, 2.3GHz clock speed, and 32GB memory. The device code was executed on NVIDIA GK110 "Kepler" K20X GPU, with 2688 processor cores, and 6GB memory. The total execution times reported in the tables contain the time required for initialization of arrays on the host and device, the execution time of the algorithm, and the time required for cleanup. We also developed an OpenMP version of our parallel algorithm, in which the kernel-analogues were implemented on the host using `# pragma omp parallel for` directives.

6.1. Single GPU experiments: small scale

For these tests, the problem instances were generated with the number of vertices ranging from $n = 500$ to $n = 5000$, with increments of 500. The cost matrices were fully dense and the elements were randomly drawn from a uniform distribution of integers in the range $[0, n]$. For each problem size, we generated 3 instances and performed 5 repetitions on each instance (total 15 runs). For each run, we recorded the total execution time (in seconds). We compared OpenMP/CPU implementation of the alternating tree variant (1 thread and 8 thread versions) with the CUDA/GPU versions of classical and alternating tree variants. The average computational results for the two algorithms are shown in Table 1 and Fig. 2. From the results, we can see that the sequential (single thread OpenMP) implementation is much more efficient for problems with $n \leq 1000$, due to the absence of data transfer and thread invocation overheads involved in the multi-threaded OpenMP and CUDA versions. However, the multi-threaded OpenMP and CUDA versions outperform the sequential version for problems with $n \geq 1500$. Moreover, the performance of our CUDA/GPU algorithms is superior to the multi-threaded OpenMP version due to availability of a lot more threads and processor cores on the GPU, as compared to the CPU. We can also see that the classical variant performs better than the alternating tree variant on these small problems, owing to its higher granularity of parallelization. Table 2 shows the number of augmenting paths found during various steps in the GPU accelerated alternating tree variant. We can see that the algorithm finds large number of augmenting paths during initial stages, and the number decreases with the increasing iteration count.

6.2. Single GPU experiments: large scale

For these tests, we followed a slightly modified experimental setup of Balas et al. [2]. The problems in this set were generated with the number of vertices ranging from $n = 5000$ to $n = 20,000$, with increments of 5000. The cost matrices

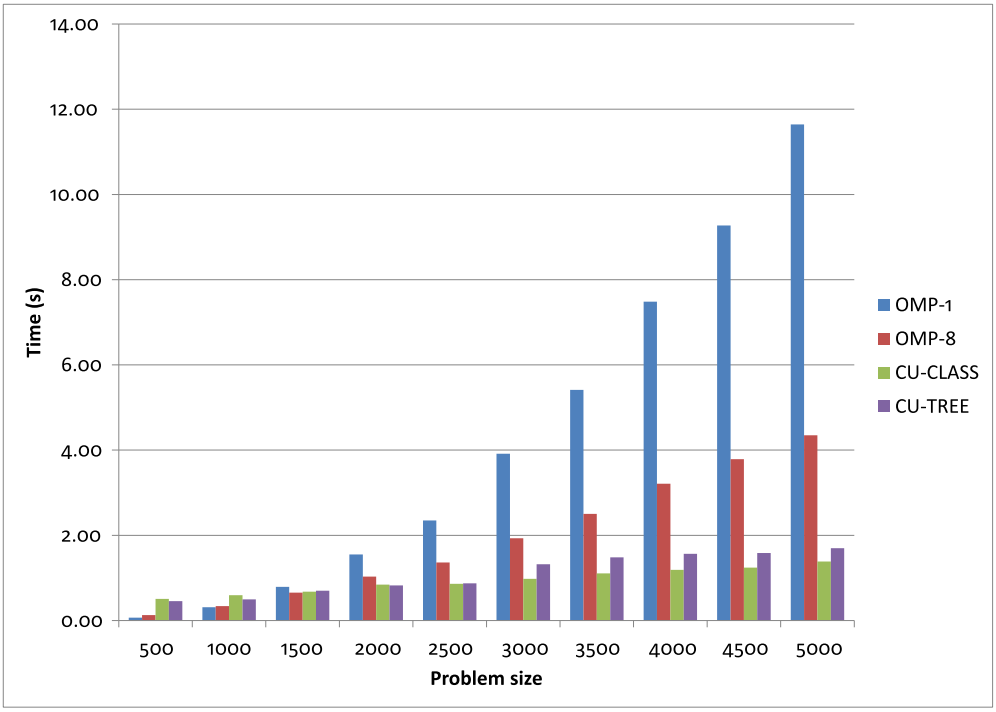

Fig. 2. Comparison chart for execution time (s).

were fully dense and the elements were randomly drawn from a uniform distribution of integers in the ranges $[0, \frac{n}{10}]$, $[0, n]$, and $[0, 10n]$. For each problem size and cost range, we generated 3 instances, and performed 5 repetitions on each instance (total 15 runs). For each run, we recorded the execution times for various steps (in seconds). In Table 3, we have presented the average computational results for the OpenMP/CPU version of the alternating tree variant (with 8 threads) and the CUDA/GPU versions of the two variants. In Fig. 3 we have shown the contribution of individual operations towards the overall execution time, for $n = 20,000$. Again, we can see that the GPU-accelerated versions are much more efficient than the multi-core CPU version. We can also see that, as the zero-elements in the cost matrix become sparser, the number of initial assignments decreases. Additionally, the relative contribution of the matrix compression (in classical variant), forward pass, and dual update steps increases. This is due to the fact that, in a cost matrix with more dissimilar values, fewer augmenting paths are found during each iteration, and the algorithm spends most of the time finding the maximum matching and updating the dual variables. We can also see that in the classical variant, matrix compression step becomes the primary bottleneck for larger problems due to its $O(n^2)$ complexity, and in those cases, the alternating tree variant starts to dominate. For the classical variant, memory is another limiting factor for instances with size $n \geq 24,000$ because we need to store both the cost matrix and the adjacency list. Therefore, we have to resort to the multi-GPU implementation, which can address both these bottlenecks to some extent. On the contrary, the alternating tree variant can handle larger problems with $n \leq 35,000$ without having to go to the GPU cluster.

6.3. Multi-GPU experiments

For the multi-GPU implementation, we conducted weak and strong scalability studies. In weak scalability tests, the problem size is increased in proportion to the number of processing elements. These results are used to demonstrate the scaling behavior of algorithms which are primarily bounded by the memory. In strong scalability tests, the problem size is kept constant and the number of processing elements are increased. These results are used to demonstrate the scaling behavior of algorithms which are primarily bounded by the CPU.

For weak scalability tests, initial problem is generated with $n = 10,000$ vertices, for a single GPU. The number of GPUs is doubled up to 16, and for each increment, the problem size is multiplied by $\sqrt{2}$, ensuring that each GPU contains about 100 million cost elements. The cost elements are randomly drawn from a uniform distribution of integers between $[0, 10n]$. The weak scaling efficiency of the algorithm is calculated as: $E_{\text{weak}} = \frac{t_1}{t_p}$, where t_p represents the execution time observed for p number of GPUs. The results for weak scalability tests are shown in Table 4, and Fig. 4. Ideally, the weak scaling efficiency should remain constant (equal to 1). We can see that the overall efficiency of our parallel algorithm(s) deteriorates with increasing number of GPUs (and problem size). The matrix compression step exhibits very good scaling behavior, which is the primary bottleneck in the classical variant. The multi-GPU version of the alternating tree variant shows significant

Table 3
Test results for large instances.

Cost range $[0, \frac{n}{10}]$[a]												
n	Obj Val	Initial Assignment Count	Time (s)									
			OMP-8			CU-CLASS				CU-TREE		
			Forward pass	Dual update	Total	Matrix comp.	Forward pass	Dual update	Total	Forward pass	Dual update	Total
5000	1.0	5000.0	–	–	1.87	–	–	–	0.44	–	–	0.51
10000	0.7	10000.0	–	–	6.18	–	–	–	0.63	–	–	1.15
15000	0.7	15000.0	–	–	13.99	–	–	–	0.97	–	–	1.55
20000	0.3	20000.0	–	–	26.42	–	–	–	1.41	–	–	2.05
Cost range $[0, n]$												
n	Obj Val	Initial Assignment Count	Time (s)									
			OMP-8			CU-CLASS				CU-TREE		
			Forward pass	Dual update	Total	Matrix comp.	Forward pass	Dual update	Total	Forward pass	Dual update	Total
5000	5795.3	4326.7	3.91	0.00	4.92	0.08	0.81	0.08	1.43	0.90	0.03	1.39
10000	11748.3	8653.3	12.35	0.00	15.83	0.32	1.25	0.22	2.44	1.48	0.05	2.17
15000	17439.0	13032.7	27.98	0.00	36.61	0.69	1.65	0.51	3.85	2.18	0.07	3.21
20000	23393.3	17350.0	49.03	0.00	65.77	1.34	2.03	0.87	5.69	2.75	0.11	4.22
Cost range $[0, 10n]$												
n	Obj Val	Initial Assignment Count	Time (s)									
			OMP-8			CU-CLASS				CU-TREE		
			Forward pass	Dual update	Total	Matrix comp.	Forward pass	Dual update	Total	Forward pass	Dual update	Total
5000	5795.3	4049.3	8.11	0.01	9.00	0.72	2.52	0.66	4.40	3.14	0.22	3.86
10000	11748.3	8140.3	30.59	0.02	33.90	2.81	4.28	1.95	9.74	5.52	0.46	6.67
15000	17439.0	12193.7	68.80	0.03	77.11	8.01	5.79	5.92	20.78	7.96	0.83	9.81
20000	23393.3	16268.7	121.88	0.04	137.98	12.99	7.18	8.36	30.05	10.34	1.06	12.81

[a] Augmenting path search and dual update steps are not required to be executed for these problem instances.

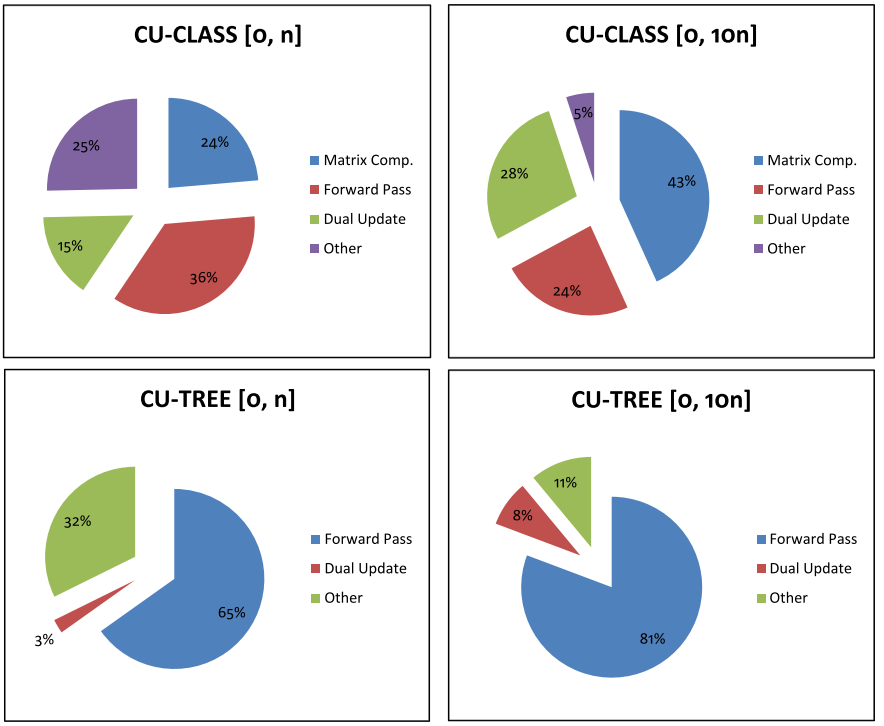

Fig. 3. Contribution of various steps in execution time for $n = 20,000$.

Table 4
Weak scalability results.

n	GPUs	CU-CLASS						CU-TREE			
		Time (s)			Scaling efficiency			Time (s)		Scaling efficiency	
		Matrix comp.	AP search	Overall	Matrix comp.	AP search	Overall	AP search	Overall	AP search	Overall
10,000	1	3.66	4.97	10.42	1.00	1.00	1.00	15.26	16.28	1.00	1.00
14,142	2	3.80	6.70	12.48	0.96	0.74	0.83	18.70	20.26	0.82	0.80
20,000	4	4.09	8.42	15.03	0.90	0.59	0.69	24.50	26.51	0.62	0.61
28,284	8	4.36	11.43	18.58	0.84	0.43	0.56	34.00	37.03	0.45	0.44
40,000	16	4.64	14.91	23.59	0.79	0.33	0.44	52.17	58.32	0.29	0.28

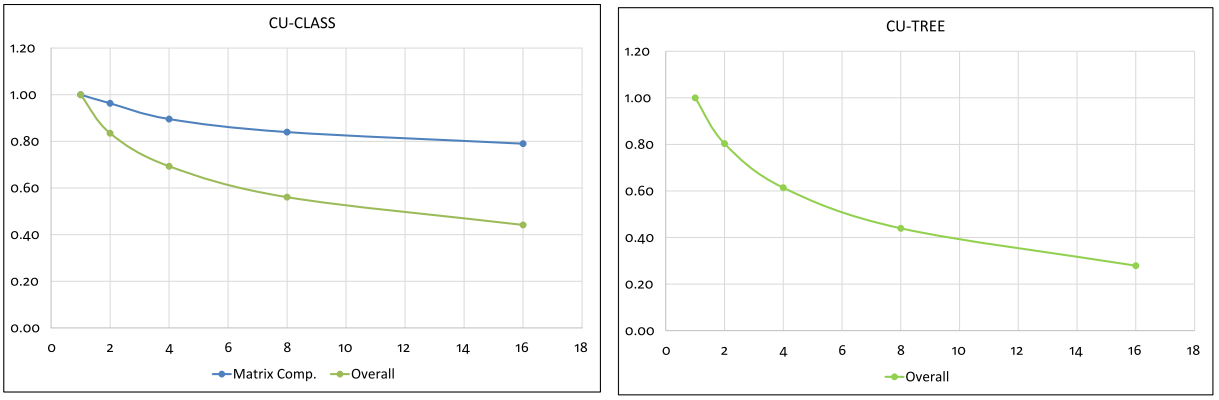

Fig. 4. Weak scaling efficiency.

Table 5
Strong scalability results.

n	GPUs	CU-CLASS						CU-TREE			
		Time (s)			Scaling efficiency			Time (s)		Scaling efficiency	
		Matrix comp.	AP search	Overall	Matrix comp.	AP search	Overall	AP search	Overall	AP search	Overall
10,000	1	3.66	4.93	11.07	1.00	1.00	1.00	15.20	16.52	1.00	1.00
10,000	2	1.89	4.97	8.15	0.97	0.50	0.68	14.14	15.12	0.54	0.55
10,000	4	1.01	5.01	7.84	0.91	0.25	0.35	14.35	15.78	0.26	0.26
10,000	8	0.58	5.01	7.39	0.78	0.12	0.19	14.93	16.61	0.13	0.12
10,000	16	0.38	4.97	7.17	0.60	0.06	0.10	16.15	18.30	0.06	0.06
14,142	1	8.24	6.57	18.25	1.00	1.00	1.00	22.44	24.19	1.00	1.00
14,142	2	4.21	6.68	12.94	0.98	0.49	0.71	19.10	20.65	0.59	0.59
14,142	4	2.18	6.74	11.03	0.94	0.24	0.41	18.91	20.55	0.30	0.29
14,142	8	1.18	6.75	9.98	0.87	0.12	0.23	19.60	21.59	0.14	0.14
14,142	16	0.69	6.76	9.50	0.75	0.06	0.12	20.88	23.50	0.07	0.06
20,000	1	15.49	8.26	28.50	1.00	1.00	1.00	35.73	38.30	1.00	1.00
20,000	2	7.88	8.38	19.47	0.98	0.49	0.73	27.14	29.21	0.66	0.66
20,000	4	4.04	8.49	15.07	0.96	0.24	0.47	24.18	26.22	0.37	0.37
20,000	8	2.13	8.50	13.00	0.91	0.12	0.27	24.96	27.47	0.18	0.17
20,000	16	1.20	8.48	12.03	0.80	0.06	0.15	26.89	30.21	0.08	0.08

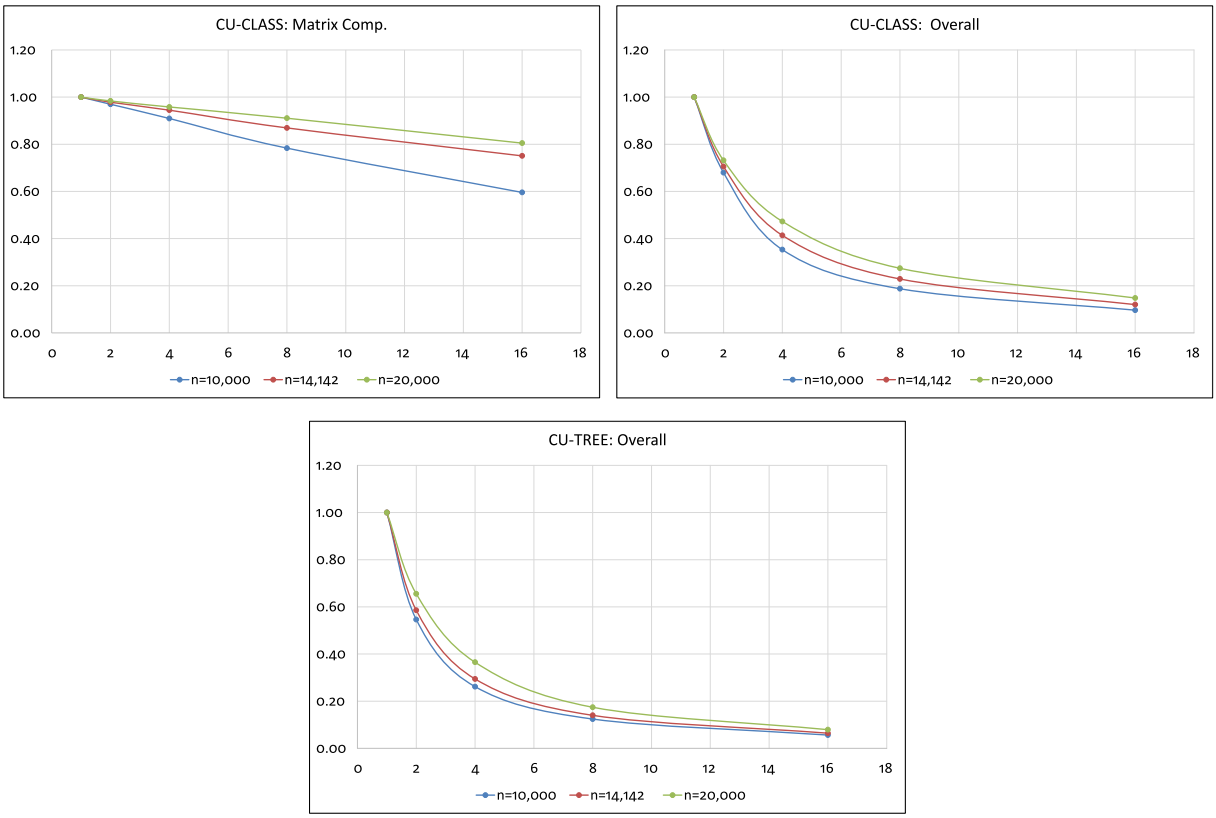

Fig. 5. Strong scaling efficiency.

increase in the execution time (even for a single CPU-GPU pair), due to the MPI directives and communication overhead in the parallel BFS.

For strong scalability tests, problems are generated with $n = 10,000$, $n = 14,142$, and $n = 20,000$ vertices, with cost elements between $[0, 10n]$. For each problem, the number of GPUs is increased from 1 to 16, in powers of 2. The strong scaling efficiency for p number of GPUs is calculated as: $E_{\text{strong}} = \frac{t_1}{p \times t_p}$. The results for strong scalability tests are shown in Table 5, and Fig. 5. We can see that for all problem sizes, the overall efficiency of our parallel algorithm(s) deteriorates

sharply with increasing number of GPUs. Once again, the matrix compression step shows excellent scaling behavior, and the scaling efficiency improves with the increasing problem size.

From the weak and strong scalability results, we can conclude that the multi-GPU implementation of the classical variant is a viable alternative for solving large problems with dense cost structures (e.g., $[0, 10n]$). This is because the matrix compression step is the dominant contributor in the execution time and it is embarrassingly parallelizable. However, for smaller problems with sparse cost structures (e.g., $\left[0, \frac{n}{10}\right]$ and $[0, n]$), scaling efficiency would be poor, due to the fact that the forward and reverse pass steps are the main contributors in the execution time, which are limited to a single GPU. The alternating tree variant is not scalable in a multi-GPU setting, however, its primary advantage is that we can solve truly large sized problems, provided we have sufficient number of GPUs.

7. Conclusions

To summarize, we developed parallel versions of the classical and the alternating tree variants of the Hungarian algorithm, for solving the linear assignment problem on a single GPU, as well as multiple GPUs in a grid setting. Although, the sequential algorithm does not readily lend itself to massive parallelism like the Auction algorithm, each step of the algorithm can be parallelized on the GPU. Our main contribution is an efficient GPU-based parallel algorithm for the augmenting path search, which happens to be the most time intensive step of the Hungarian algorithm. We showed that our algorithm(s) can find multiple vertex-disjoint paths that can be used to augment the solution during each iteration, which drastically reduces the execution time. We conducted extensive numerical tests on the algorithm(s), on small and large scale problems, which reveal that our algorithm(s) are significantly faster than the sequential and OpenMP implementations solved on a multi-core CPU. Although the OpenMP version implemented on a faster CPU with greater number of cores can potentially outperform the GPU version, such CPUs are extremely costly, making them out of reach for an average researcher. On the contrary, a simple gaming graphics card (NVIDIA GeForce GTX 970) contains 1664 CUDA cores and it can be bought for only about $400, which makes our implementation all the more attractive. Out of the two GPU-accelerated algorithms, the alternating tree variant has one order of magnitude smaller asymptotic complexity, therefore, it is bound to outperform the classical variant at some point. Additionally, it is best suited for denser cost matrices and therefore it is an excellent choice for LAPs with non-integral costs. The multi-GPU version of the classical variant can be used to efficiently solve larger problems with dense cost matrix structure. For small problems, however, the multi-GPU version does not provide adequate scaling efficiency, because of the limitation imposed by the augmenting path search step. The alternating tree variant shows poor scaling on multi-GPU setting owing to the fact that there is significant MPI communication during the BFS iterations, however, it can be used to solve problems that are extremely large, if we have the required number of GPUs.

Our algorithm(s) can be implemented in various solution methodologies for important NP-hard problems, such as data association and graph matching, which can benefit from its large parallel speedup. We believe that our parallelization strategy can provide an elegant and cost effective way of solving these problems on a desktop computer, simply equipped with a graphics card, without having to rely on the large computational grids. The multi-GPU versions provide good alternative for solving larger problems which cannot be solved on a single GPU due to memory limitations, subject to the availability of the necessary hardware.

Acknowledgments

This work has been supported by a Multidisciplinary University Research Initiative (MURI) grant (Number W911NF-09-1-0392) for Unified Research on Network-based Hard / Soft Information Fusion, issued by the US Army Research Office (ARO) under the program management of Dr. John Lavery. We gratefully appreciate this support.

Initial development and testing of this work was done using the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1053575; and the computational resources of the Keeneland Computing Facility at the Georgia Institute of Technology, which is supported by the National Science Foundation under Contract OCI-0910735. We gratefully appreciate this support.

Final testing of this work was done using the resources from the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (awards OCI-0725070 and ACI-1238993) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications. We gratefully appreciate this support.

References

[1] R.K. Ahuja, T.L. Magnanti, J.B. Orlin, Network flows: theory, algorithms, and applications, Prentice Hall, Upper Saddle River, NJ, USA, 1993.
[2] E. Balas, D. Miller, J. Pekny, P. Toth, A parallel shortest augmenting path algorithm for the assignment problem, J. ACM (JACM) 38 (4) (1991) 985–1004.
[3] M.S. Bazaraa, J.J. Jarvis, H.D. Sherali, Linear programming and network flows, John Wiley & Sons, 2011.
[4] D.P. Bertsekas, The Auction algorithm for assignment and other network flow problems: a tutorial, Interfaces 20 (4) (1990) 133–149.
[5] D.P. Bertsekas, D.A. Castañon, Parallel synchronous and asynchronous implementations of the Auction algorithm, Parallel Comput. 17 (6) (1991) 707–732.
[6] D.P. Bertsekas, D.A. Castañon, Parallel asynchronous Hungarian methods for the assignment problem, ORSA J. Comput. 5 (3) (1993) 261–274.
[7] G.E. Blelloch, Prefix Sums and Their Applications, Technical Report, Synthesis of Parallel Algorithms, 1990.
[8] R.E. Burkard, E. Çela, Linear assignment problems and extensions, in: D.-Z. Du, P. Pardalos (Eds.), Handbook of Combinatorial Optimization, Springer US, 1999, pp. 75–149, doi:10.1007/978-1-4757-3023-4_2.

[9] L. Buš, P. Tvrdík, Towards Auction algorithms for large dense assignment problems, Comput. Optimization Appl. 43 (3) (2009) 411–436.
[10] M. Harris, S. Sengupta, J.D. Owens, Parallel prefix sum (scan) with CUDA, GPU Gems 3 (39) (2007) 851–876.
[11] J. Hoberock, N. Bell, Thrust: a parallel template library, 2010, http://thrust.github.io/, Version 1.7.0.
[12] R. Jonker, A. Volgenant, Improving the Hungarian assignment algorithm, Operations Res. Lett. 5 (4) (1986) 171–175.
[13] R. Jonker, A. Volgenant, A shortest augmenting path algorithm for dense and sparse linear assignment problems, Computing 38 (4) (1987) 325–340.
[14] R. Jonker, A. Volgenant, Linear assignment procedures, Eur. J. Operational Res. 116 (1) (1999) 233–234.
[15] J.L. Kennington, Z. Wang, An empirical analysis of the dense assignment problem: Sequential and parallel implementations, ORSA J. Comput. 3 (4) (1991) 299–306.
[16] H.W. Kuhn, The Hungarian method for the assignment problem, Nav. Res. Logist. Q. 2 (1-2) (1955) 83–97.
[17] E.L. Lawler, Combinatorial optimization: networks and matroids, Courier Corporation, 1976.
[18] L. Luo, M. Wong, W.-m. Hwu, An effective GPU implementation of breadth-first search, in: Proceedings of the 47th Design Automation Conference. DAC '10, ACM, New York, NY, USA, 2010, pp. 52–55, doi:10.1145/1837274.1837289.
[19] D. Merrill, M. Garland, A. Grimshaw, Scalable GPU graph traversal, in: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, in: PPoPP '12, ACM, New York, NY, USA, 2012, pp. 117–128, doi:10.1145/2145816.2145832.
[20] J. Munkres, Algorithms for the assignment and transportation problems, J. Soc. Ind. Appl. Math. 5 (1) (1957) 32–38.
[21] A. Naiem, M. El-Beltagy, Deep greedy switching: a fast and simple approach for linear assignment problems, in: 7th International Conference of Numerical Analysis and Applied Mathematics, 2009.
[22] A. Naiem, M. El-Beltagy, M. Rasmy, A centrally coordinated parallel Auction algorithm for large scale assignment problems, in: Informatics and Systems (INFOS), 2010 The 7th International Conference on, IEEE, 2010, pp. 1–4.
[23] E.D. Nering, A.W. Tucker, Linear Programs and Related Problems, vol. 1, Access Online via Elsevier, 1993.
[24] NVIDIA, CUDA C programming guide 5.0, NVIDIA Corporation, 2012. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
[25] R. Roverso, A. Naiem, M. El-Beltagy, S. El-Ansary, S. Haridi, A GPU-enabled solver for time-constrained linear sum assignment problems, in: Informatics and Systems (INFOS), 2010 The 7th International Conference on, IEEE, 2010, pp. 1–6.
[26] M. Sathe, O. Schenk, H. Burkhart, An Auction-based weighted matching implementation on massively parallel architectures, Parallel Comput. 38 (12) (2012) 595–614. http://dx.doi.org/10.1016/j.parco.2012.09.001. http://www.sciencedirect.com/science/article/pii/S0167819112000750
[27] S. Sengupta, A.E. Lefohn, J.D. Owens, A work-efficient step-efficient prefix sum algorithm, in: Proceedings of the Workshop on Edge Computing Using New Commodity Architectures, 2006, pp. 26–27.
[28] S. Storøy, T. Sørevik, Massively parallel augmenting path algorithms for the assignment problem, Computing 59 (1) (1997) 1–16.
[29] C.N. Vasconcelos, B. Rosenhahn, Bipartite graph matching computation on GPU, in: Energy Minimization Methods in Computer Vision and Pattern Recognition, Springer, 2009, pp. 42–55.
[30] A. Volgenant, Linear and semi-assignment problems: a core oriented approach, Comput. Operations Res. 23 (10) (1996) 917–932.
[31] J.M. Wein, S. Zenios, Massively parallel Auction algorithms for the assignment problem, in: Frontiers of Massively Parallel Computation, 1990. Proceedings., 3rd Symposium on the, IEEE, 1990, pp. 90–99.