Parallelizing the Hungarian Method on GPU

Andrew Kope
Western University

The Assignment Problem

Problem Description

- There are a number of *agents* and *tasks*
- Any agent can be assigned to perform any task, incurring some *benefit* that may vary depending on the agent-task assignment
- It is required to perform all tasks by assigning exactly one agent to each task

- **Agents are assigned in such a way that the total benefit of the assignment is maximized**

(http://en.wikipedia.org/wiki/Assignment_problem)

Example

		Tasks		
		Task 1	Task 2	Task 3
Agent	1	250	300	150
	2	300	700	250
	3	100	200	150

Example

		Tasks		
		Task 1	Task 2	Task 3
Agent	1	250	300	150
	2	300	700	250
	3	100	200	150

The optimal assignment is 250 + 700 + 150 = 1100

Kuhn's Hungarian Method

History

- Published in 1955
- Based on the work of two Hungarian mathematicians

  “one can always construct an optimal assignment by a succession of transfers followed by additional assignments until this is no longer available”

Description

- Step 1. Subtract the smallest entry in each row from all entries in its row
- Step 2. Subtract the smallest entry in each column from all entries in its column
- Step 3. Draw lines through appropriate rows and columns such that:
  - All the zero entries of the cost matrix are covered
  - The minimum number of such lines is used
- Step 4. Test for Optimality:
  - (i) If the minimum number of covering lines is n, an optimal assignment of zeros is possible and we are finished
  - (ii) If the minimum number of covering lines is less than n, an optimal assignment of zeros is not yet possible; proceed to Step 5
- Step 5. Determine the smallest entry not covered by any line. Subtract this entry from each uncovered row, and add it to each covered column; return to Step 3

(Bruff, 2005)

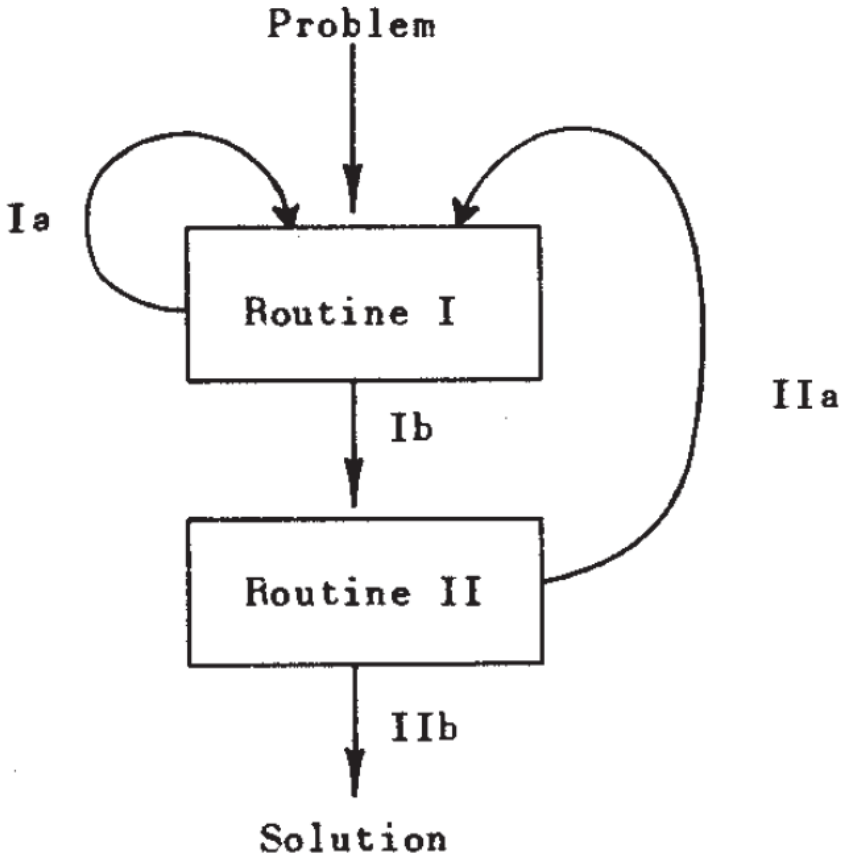
Algorithm

I: Routine 1; assign agents

Ia: Increase number of assignments

II: Routine 2; test for optimality

IIa: Decrement current cover



(Kuhn, 1955)

The Cover

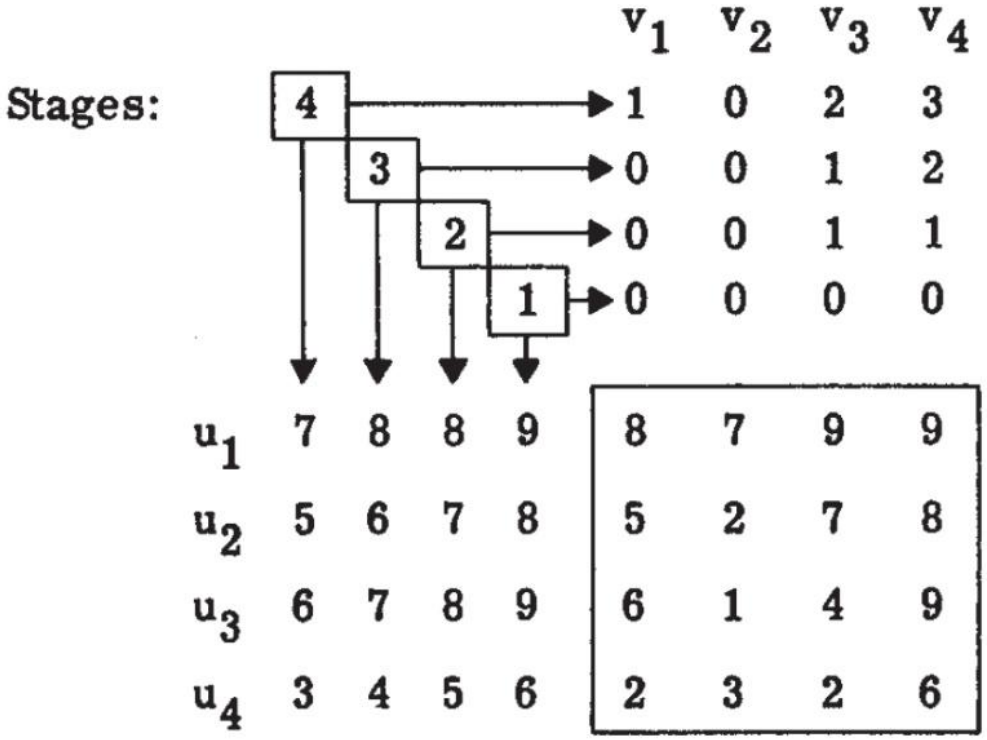
Description

- A set of lines is a cover if it contains all the zeroes in the cost/benefit matrix
- Composed of two vectors $\{U_i, V_j\}$
- With each iteration of the algorithm, the cover decreases in size
  - The minimal cover is an optimal solution

Example

Sum of row maxima = 9 + 8 + 9 + 6 = 32

Sum of column maxima = 8 + 7 + 9 + 9 = 33



(Kuhn, 1955)

Calculating U and V

1. Find the maximum value of each row
2. Find the maximum value of each column
3. Sum the maximum values of every row
4. Sum the maximum values of every column
5. Compare the two sums
   - If the sum of the row maxima is least, the maximum values of each row are assigned to U
   - Conversely, if the sum of the column maxima is least, the maximum values of each column are assigned to V

Parallelization

Calculating U and V

1. Find the maximum value of each row
2. Find the maximum value of each column
3. Sum the maximum values of every row
4. Sum the maximum values of every column
5. Compare the two sums
   - If the sum of the row maxima is least, the maximum values of each row are assigned to U
   - Conversely, if the sum of the column maxima is least, the maximum values of each column are assigned to V

Finding Row Maxima

```
//Launch as many kernels as there are rows
__global__ void FindMaxRow(int *d_r, int *d_row_max, int num_rows, int num_col)  {
    int currentRow = blockIdx.x*blockDim.x + threadIdx.x;
    if (currentRow < num_rows) {
            int currentCol;
            //loop through columns
            for (currentCol = 0; currentCol<num_col; currentCol++) {
                    //check to see if we have a new leader for the row
                    if (d_row_max[currentRow] < d_r[currentRow*num_col + currentCol]) {
                            d_row_max[currentRow] = d_r[currentRow*num_col + currentCol];
                    }
            }
    }
}
```

Finding Column Maxima

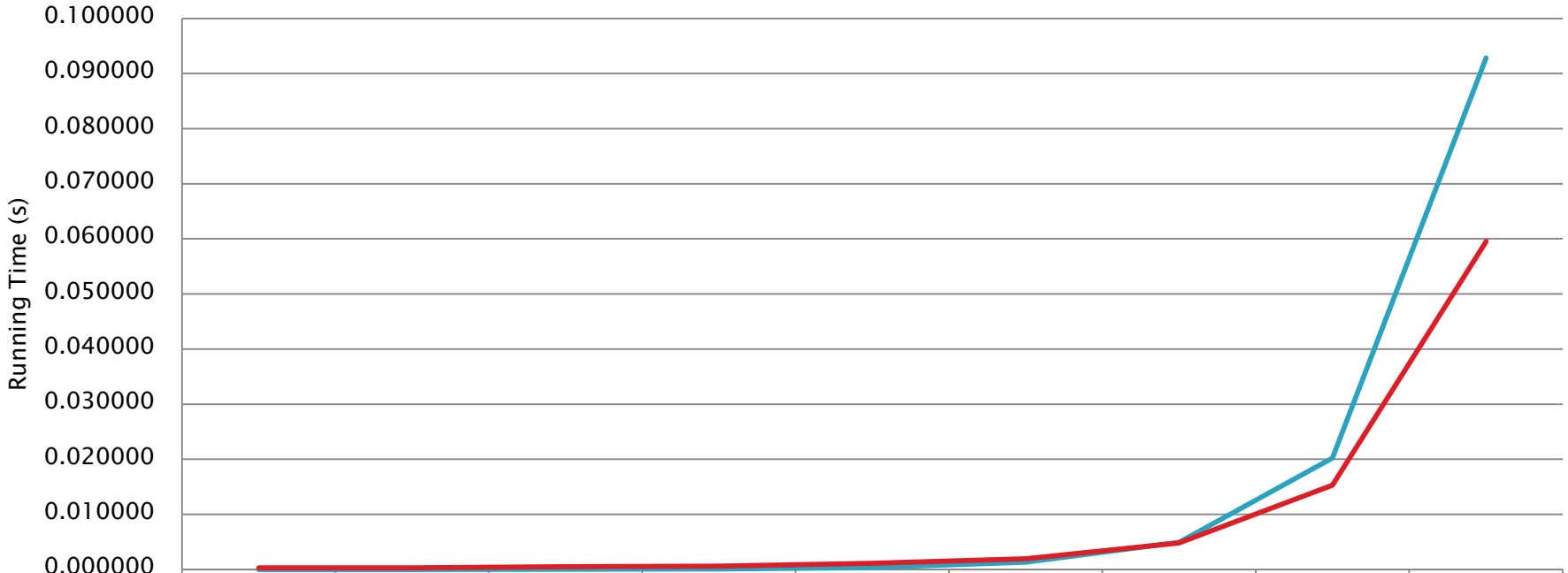
```
//Launch as many kernels as there are columns
__global__ void FindMaxCol(int *d_r, int *d_col_max, int num_rows, int num_col)  {
    int currentCol = blockIdx.x*blockDim.x + threadIdx.x;
    if (currentCol < num_col) {
            int currentRow;
            //loop through rows
            for (currentRow = 0; currentRow<num_row; currentRow++) {
                    //check to see if we have a new leader for the column
                    if (d_col_max[currentCol] < d_r[currentRow*num_col + currentCol]) {
                            d_col_max[currentCol] = d_r[currentRow*num_col + currentCol];
                    }
            }
    }
}
```

Results

Running Time vs. Input Size

	2	4	8	16	32	64	128	256	512	1024	2048	4096
Serial	0.00000	0.00000	0.00000	0.00001	0.00003	0.00013	0.00050	0.00196	0.00795	0.03820	0.16359	0.72237
Parallel	0.00029	0.00029	0.00030	0.00031	0.00052	0.00063	0.00096	0.00201	0.00666	0.02196	0.07818	0.31150

Running Time vs. Input Size



	(2,20)	(4,40)	(8,80)	(16,160)	(32,320)	(64,640)	(128, 1280)	(256, 2560)	(512, 5120)
Serial	0.000003	0.000009	0.000024	0.000088	0.000330	0.001323	0.004920	0.020214	0.092821
Parallel	0.00029922	0.00030767	0.000503	0.00060456	0.00109522	0.00193911	0.00479389	0.01527856	0.05952889

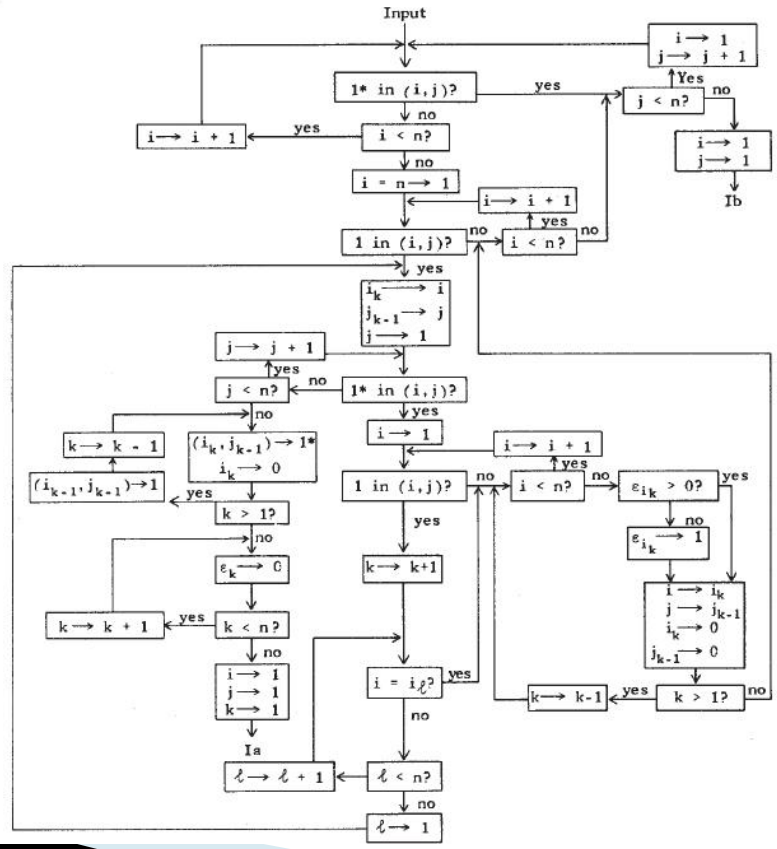
Limitations and Future Directions

Limitations

- No use of GPU shared memory
- Input size

Future Directions

- Use GPU to perform reductions
- Parallelize the transfer algorithm



(Kuhn, 1955)

References

Bruff, D. (2005). The assignment problem and the hungarian method. Retrieved from: http://www.math.harvard.edu/archive/20_spring_05/handouts/assignment_overheads.pdf

Kuhn, H. W. (1955). The hungarian method for the assignment problem. 83-97.

Kuhn, H. W. (1960). The hungarian method.