Improvements to Hungarian Solver

Samiran Kawtikwar
Industrial and Systems Enterprise Engineering
samiran2@illinois.edu

Wes Cravens
Industrial and Systems Enterprise Engineering
wbc3@illinois.edu

Yuechun (Nick) Yang
Computing Systems and Software Computer Engineering
yyang192@illinois.edu

***Abstract*—The linear assignment problem (LAP), or bipartite matching, is a foundational problem in combinatorial optimization with many applications. Many algorithms have been proposed in the literature. The most famous one is the Hungarian algorithm based on a primal-dual formulation. Being a polynomial time algorithm, it has scalability issues and poses challenges on sequential processors in settings with high demand for large instances. General purpose Graphics Processing Units (GPGPUs) have shown promise meeting data scalability and compute bandwidth requirements in recent times. Some preliminary work in the HPC domain has revealed significant parallelism available in the problem structure and shown performance improvements. This work is mainly focused on cost matrix preprocessing and smart row/column cover choices that will increase parallel processing opportunities resulting in faster convergence and reduced computation times.**

***Index Terms*—Graphics processing unit, Hungarian algorithm, Linear assignment problem, Parallel algorithms, Preprocessing.**

I. Introduction

The linear assignment problem is a fundamental problem in combinatorial optimization and has vast applications ranging from transportation systems, personal assignments, graph association. The problem is popularly known as the perfect matching problem in bipartite graphs. Given an LAP with n resources, n tasks and a cost matrix $C_{n\times n} = [c_{ij}]$. Where c_{ij} denotes the cost to assign resource i to task j. The objective of the LAP is to minimize total assignment cost by complying with the constraint that a bijection is formed between resources and tasks in the assignment. Many algorithms have been proposed for the LAP in the literature. With the pioneering Hungarian algorithm proposed by Kuhn based on a primal-dual formulation of the LAP, this algorithm is highly favored for implementing on GPUs since the nature of operating directly on cost matrix is well suited.There are two variants for the Hungarian Algorithm: 1) The classical version developed by Munkers; and 2) the alternating tree version developed by Lawler. Both variants work on the augmenting path search phases to adjust the assignment and dual update phase to introduce new candidates for augmenting. The classical algorithm has worst case complexity $O(N^4)$ while alternating tree version is $O(N^3)$. Both algorithms naturally have bottlenecks on the path augmenting search and dual update phase.

The Hungarian algorithm can be visualized as a six step iterative algorithm. We describe each step in brief as follows:

1) Subtract the row minimum from each row. Subtract the column minimum from each column.
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.
3) Cover each column with a starred zero. If all the columns are covered then the matching is maximum.
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.
5) Construct a series of alternating primed and starred zeros as follows: Let Z_0 represent the uncovered primed zero found in step 4. Let Z_1 denote the starred zero in the column of Z_0 (if any). Let Z_2 denote the primed zero in the row of Z_1 (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.
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.

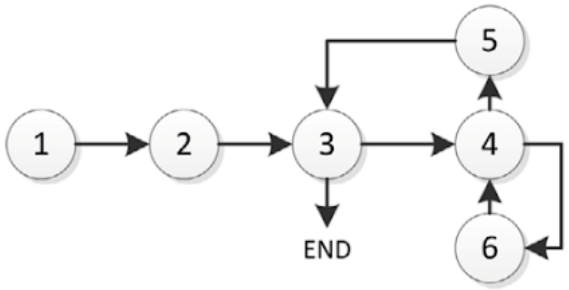

Fig. 1: The flow of Hungarian Algorithm.

II. Motivation

Perfect matching in bipartite graphs is a fundamental problem problem in graph theory, it is also a very important problem in Operations research with applications from

warehouse matching to aircraft scheduling. It is also used as a heuristic for obtaining lower bounds in the Quadratic Assignment Problem (QAP), Traveling Salesman Problem (TSP), Multi-Dimensional Assignment Problem (MDAP) etc. These instances demand a large number of LAP instances. Which naturally motivates use of parallel throughput oriented computational hardware such as General Purpose Graphics Processing Units (GPGPUs). The project lead has done some previous work in this domain and found that the [1] implementation is faster than [2]. They also made some computational contributions to improve the performance. The summary of this work was presented as a report [3]. Some of the important findings from this work revealed no huge scope of further computational improvements and that the performance was mostly limited by the number of iterations and computational steps. This motivates us to utilize the course project and witness the effect of preprocessing techniques and dual update improvements on total execution time to measure their benefit.

III. Literature Review

The Linear Assignment Problem (LAP) can be formally described as follows. Given two sets A and T of equal sizes, together with a cost function $C : A \times T \to \mathbb{R}$, the objective of the LAP is to find a bijection $f : A \to T$ such that the objective function,

$$\sum_{\alpha \in A} C(\alpha, f(\alpha)),$$

is minimized. The problem is called Linear Assignment Problem as the objective function and constraints are both linear. LAP is one of the most well studied combinatorial optimization problems that can be solved in polynomial time. According to [4] LAP solving techniques can be broadly classified into three categories, these are based on algorithms for:

1) Maximum flow
2) Shortest Path
3) Linear Programming

Most algorithms based on *maximum flow* are primal-Dual methods, The Hungarian Algorithm (explained in I given by the Hungarian mathematicians Dénes Kőnig and Jenő Egerváry [5] was actually developed before the primal-dual techniques and served as a building block for the general primal-dual techniques. The first polynomial implementation of this algorithm developed by James Munkers and Harold kuhn [5] was $\mathbb{O}(n^4)$ and is also known as *Munker's* implementation. Later $\mathbb{O}(n^3)$ version was developed by Lawler [6] and is famously known as *Lawler's* implementation. Bertsekas [7] also developed a similar primal-dual algorithm and his best performing version even switches to the Hungarian Algorithm itself as soon as the original method becomes less effective.
Jonker and Volgenant [8] suggested 3 improvements to the Hungarian algorithm given in [7] and [6] by reducing additional steps while:

1) Scanning rows and columns together to find zero cost augmenting path.
2) Finding local minimum cost instead of global minimum for Dual update.
3) During the dual cost update step, instead of updating costs for whole row, the column corresponding to minimum cost is marked covered.

These improvements further helped develop shortest path based algorithms.
The methods based on *shortest path* are dual algorithms, in these methods dual feasibility exists and primal feasibility has to be reached. This is achieved by modelling the problem as a minimum cost flow problem [4]. Edmonds [9] and Tomizawa [10] independently developed $\mathbb{O}(n^3)$ algorithms for LAP based on shortest path techniques. These algorithms use flow augmentation along shortest paths in *auxiliary* network, that can be constructed based on original network and current flow. According to [11] Further development in shortest path algorithms for LAP mostly differs in these aspects:

1) Procedure used for determining shortest paths
2) *sparsification* techniques
3) preprocessing methods to determine feasible dual solutions

Linear Programming based algorithms are specialized versions of *simplex*. Some of the best published results are from Barr et al. [12]. An inherent problem with simplex based techniques is the phenomenon of *degenerate pivots* and relatively complicated implementation. Computational experiments in [13] and [14] show that these algorithms are outperformed by the max flow based (Primal-Dual) and shortest path based (Dual) methods.
Further development in linear assignment problems was fueled by graph theory and max cardinality matching techniques. The Hopcroft-Karp algorithm [15] is the first known "multi-threaded" algorithm for max cardinality matching that introduced vertex disjoint paths to simultaneously perform multiple pivots. The algorithm has a complexity of $\mathbb{O}(n^{5/2})$for dense bipartite graphs. This was further improved by using fast adjacency techniques in [16] to achieve $\mathbb{O}(n^{1.5}\sqrt{m/\log(n)}$. However, closing the gap between complexity for maximum cardinality matching and linear assignment problem proved to be difficult. The theoretical result by [17] finally closed the gap in 2003. Table I shows a summary of these algorithms by complexity and category.
With the development of *multi cor*e processors many sequential methods for LSAP were parallelized and computationally tested on parallel machines. Most of the theoretical contributions in this domain are concerned with implementations on *parallel random access machines* PRAM. PRAM is a parallel computer with p processing units working synchronously on a shared memory with small constant access time. Each processor has its own program control (MIMD principle) but the whole machine is clocked globally and works synchronously such that each PRAM processor executes exactly one PRAM operation (arithmetic, memory access, branch etc.) per clock cycle.
[23] introduced priority queue data structures for parallel

TABLE I: Time complexity of sequential algorithms for Linear Assignment Problem

Year	Reference	Time complexity	Category
1946	Easterfield	exponential	Combinatorial
1955	Kuhn [5]	$O(n^4)$	Primal-dual
1964	Balinski and Gomory [18]	$O(n^4)$	Primal
1969	Dinic and Kronro [19]	$O(n^3)$	Dual
1971	Tomizawa [10]	$O(n^3)$	Shortest path
1972	Edmonds and Karp [9]	$O(n^3)$	Shortest path
1976	Lawler [6]	$O(n^3)$	Primal-dual
1977	Barr, Glover, and Klingman [12]	exponential	Primal simplex
1978	Cunningham and Marsh [20]	$O(n^3)$	Primal
1980	Hung and Rom [13]	$O(n^3)$	Dual
1993	Akgül [21]	$O(n^3)$	Primal simplex
1995	Goldberg and Kennedy [22]	$O(\sqrt{n}\, m \log(n\mathcal{C}))$	Pseudoflow
2001	Kao, Lam, Sung, and Ting [17]	$O(\sqrt{n}\, \mathcal{W} \log(\frac{n^2}{\mathcal{W}/\mathcal{C}})/\log n)$	Decomposition

TABLE II: PRAM implementations for LAP

Year	Reference	Complexity
1988	Driscoll, Gabow, Shrairman, and Tarjan [23]	$\mathbb{O}(nm/p)$
1988	Gabow and Tarjan [24]	$\mathbb{O}(\sqrt{n}m.\log(nC)\log(2p)/p)$
1988	Goldberg, Plotkin and Vaidya [22]	$\mathbb{O}(n^3/p + n^2p)$
1988	Schwiegelshohn and Thiele [25]	$\mathbb{O}(n^4/p)$
1993	Hoffman and Markowitz [26]	$\mathbb{O}(n^4/p)$
2004	Fayyazi, Kaeli and Meleis [27]	$\mathbb{O}(\sqrt{n}\log^2 n)$

implementation of shortest path algorithms for LSAP to get $\mathbb{O}(nm/p)$ time complexity with the assumption that the costs are non-negative, (p is the number of processors). We summarize parallel implementations of LSAP in Table II.

IV. Evaluation methodology

In this section, we will describe the evaluation platform, the datasets, the baseline selection, the optimization procedure, and the profiling tools we use when evaluating the GPU implementations in this report.

A. Evaluation Platform

We evaluate the GPU implementations on a single socket machine with 16 GB of main memory. The socket has a 4-core Intel i7-4790 CPU and two NVIDIA GeForce RTX 2080 Ti GPUs with 11 GB of global memory each, while we only use one GPU in our evaluation. We compile the code with NVCC (CUDA 11.5) and GCC 7.5.0. The CUDA driver version used is 495.29.05.

B. Datasets

The problem sizes of the LAP in our evaluation are chosen among $n = \{64, 128, \ldots, 8192\}$. Since, the performance of these methods varies with sparsity, we test each problem sizes for range from 0 to $0.001n, 0.01n, \ldots, 1000n$. The entries in the $n \times n$ cost matrix are generated as integers but are stored as double-precision floating-point formats and are generated uniformly randomly in the range defined above with a fixed seed using in order to achieve consistency between experiments. The random data is generated on host using $minstd_rand0$, the c++ default random generator. It is then copied to the device memory and the respective algorithm is executed. Naturally, we do not report the time for data generation, transfer etc. for all instances.

C. Baselines

We did an extensive research on purpose build lap solvers and found 5 most prominant gpu and cpu solutions. These solvers cover all approaches used for developing algorithms for Linear Assignment Problems including state of the art GPU solvers. The list of these solvers is as follows:

- Gurobi (Generic LP solver)
- JVC-sequential (due to [4], taken from scipy.org)
- LAP16 (due to [2], GPU accelerated Hungarian solver using classical implementation)
- LAP19 (due to [1], GPU accelerated solver using classical hungarian algorithm)
- JVC-parallel (due to [28], GPU accelerated JVC algorithm)

Note that, we purposefully exclude sequential implementations of Hungarian algorithm and all auction algorithms to restrict the scope of experiments and their slow performance and poor scaling.

V. Attempted Improvements

In this section we discuss all attempted and proposed improvements.

A. Improvement Attempt 1

After an in depth review of all GPU implementations [2] , [1] and [28] we found out that the initial zero covering for all GPU implementations is done using race conditions. Using insights from the transportation problem, we attempted using Vogel's regret rule for maximal zero cover.

B. Improvement Attempt 2

Following a similar theme of smart preprocessing techniques for maximal zero coverage and reduced iterations, we

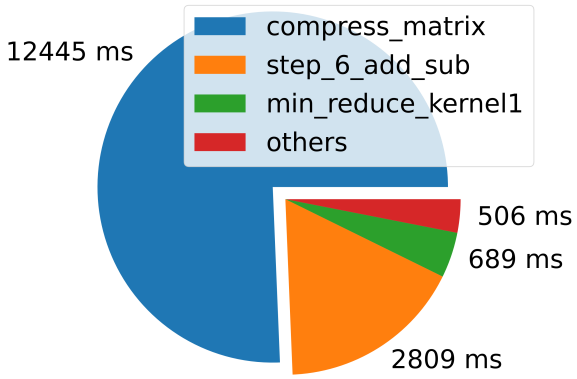

Fig. 2: Most time-consuming kernel calls aggregated by name for [1] for problem size of $n = 4096$ and range = 100

```
if (near_zero(slack[i])) {
  atomicAdd(&zeros_size, 1);
  int b = i >> log2_data_block_size;
  int i0 = b << log2_data_block_size;
  int j = atomicAdd(zeros_size_b + b, 1);
  zeros[i0 + j] = i;
}
```

Listing 1: Snippet of the `compress_matrix` kernel in the baseline implementation.

```
switch(cover_row[l] + cover_column[c]) {
  case 2:
    slack[i] += d_min_in_mat;
    break;
  case 0:
    slack[i] -= d_min_in_mat;
    break;
}
```

Listing 2: Snippet of the `step_6_add_sub` kernel in the baseline implementation.

implemented the preprocessing ideas from [29]. Though the paper claimed it to be doing multiple pivots the techniques explained restrict the scope for parallel augmentations that are readily provided by the Hungarian algorithm. From the known version of the Hungarian algorithm, in the worst case, a single uncovered element is selected to create a single new zero element in each iteration. The Accelerating Hungarian method slightly modifies the number of the smallest elements in each iteration which is advantageous in converging to the perfect match. The improvement is well suited for CPU solver and sequential operations. However, our project aims to find improvements that can benefit from GPU parallelism. Munapo's method does not bring many advantages in the aspect of parallelism. Hence, the idea had to be discarded as sequential augmentations would not be able to benefit from the massive parallelism provided by GPGPUs.

C. Improvement Attempt 3

Using tools provided by NVIDIA for system and kernel profiling, we profiled the [1] code to find bottlenecks. At a high level, we first put timers around kernel calls to find the most time-consuming kernels aggregated by name. Then, we also use NVIDIA Nsight Systems to get kernel launching statistics for API calls if necessary. Finally, we use NVIDIA Nsight Compute to understand low-level metrics such as L1 hit rate, throughput, and number of instructions executed.

1) Improvement Attempt 3.1: Fig. 2 shows that the most time consuming kernel for [1] is the `compress_matrix` kernel. On investing reasons for an exceptionally long runtime of this kernel shown in Listing 1. We observed huge contention for the first atomic operation since all threads compete for a single memory location in global memory. We reduced this contention by simple privatization technique. In this technique, each thread, instead of atomically adding to the global memory, atomically adds to shared memory. The ultimate addition is then found using reduction. With this simple optimization, the performance for the `compress_matrix` kernel improved by 8.35 times while the runtime for whole application improved by 3.07 times for problem size 4096 with range 100.

2) Improvement Attempt 3.2: After applying V-C1, the most time consuming kernel switches from `compress_matrix` to `step_6_add_sub`. 2 shows the code snippet of the `step_6_add_sub` kernel. On investigating, we observe that the kernel provides no scope for any optimization within the kernel. But on taking a broader look, it was clear that the caching performance of this kernel was really bad due to the kernel being small and previous kernels being separated from this. Since, the `slack[i]` variable appears in both `step_6_add_sub` kernel and the `compress_matrix` kernel and the number of threads required by both kernels being same. A simple kernel fusion helped improve caching performance. The updated fused kernel is shown in 3. With this improvement, we achieved a further speedup of 1.45 times in kernel execution and 1.30 times for total runtime.

D. Improvement Attempt 4

With V-C, we made sure that the CUDA implementation of the Hungarian Algorithm itself does not cause huge impact that could hamper the most time consuming steps. The observations for most expensive operations, now match with claims made in [4]. The dual update step of the algorithm turns out to be the most expensive step, even being embarrassingly parallelizable, the sheer number of floating point operations ($\mathcal{O}(n^2)$/iteration) make it an expensive step. The authors in [8] propose techniques to improve these operations from ($\mathcal{O}(n^2)$/iteration) to ($\mathcal{O}(n)$/iteration), the improvement restricts augmentation using shortest path instead any path in Hungarian Algorithm. This also restricts the augmentation to one path as opposed to possibly many disjoint paths in Hungarian. As the figure 3 shows, slight differences between both

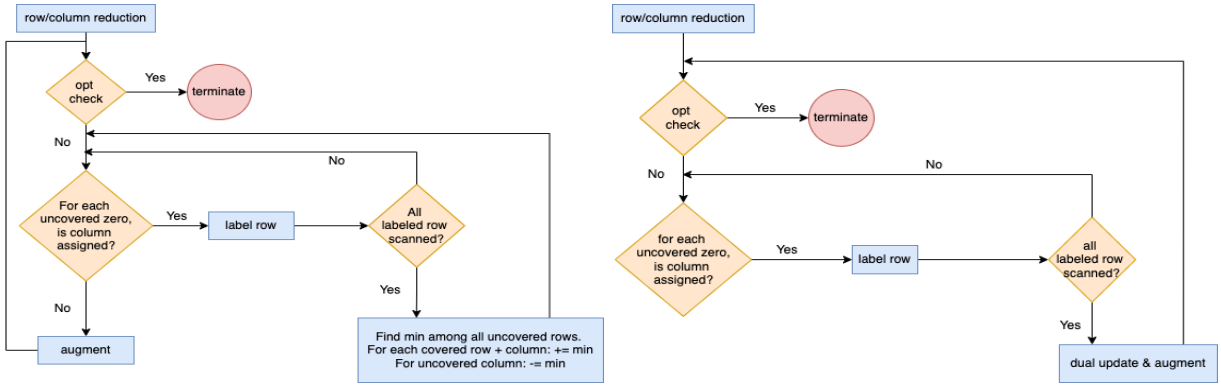

Fig. 3: Control flowchart of Hungarian ALgorithm (left) vs JVC (right)

```
switch(cover_row[l] + cover_column[c]) {
  case 2:
    slack[i] += d_min_in_mat;
    break;
  case 0:
    slack[i] -= d_min_in_mat;
    break;
}

if (near_zero(slack[i])) {
  int b = i >> log2_data_block_size;
  int i0 = b << log2_data_block_size;
  int j = atomicAdd(zeros_size_b + b, 1);
  zeros[i0 + j] = i;
}
```

Listing 3: Snippet of the `add_sub_compress_matrix` kernel in optimization 1 by fusing the `step_6_add_sub` kernel and the `compress_matrix` kernel.

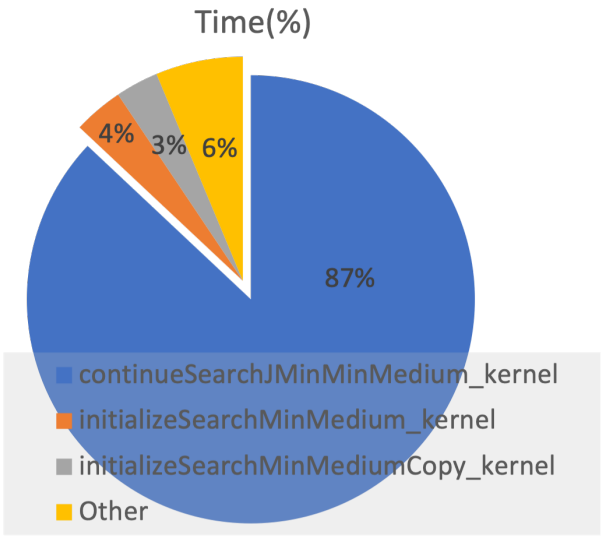

Fig. 4: Most time-consuming kernel calls aggregated by name for [28] with problem size of $n = 8192$ and range = 100

algorithms. Clearly, finding any augmenting path is easier than finding shortest augmenting path. [1] managed to implement very light kernel for finding any augmenting paths, though we could not find any method for finding shortest augmenting paths on GPU's. The authors in [28] have used GPUs for this approach but their implementation is slower than serial cpu versions mainly due to converting matrix format dataset to graph in CSR format and performing BFS with significant synchronization overhead and memory transfers with host. Fig. 4 clearly shows the shortest augmenting path kernel dominating total runtime in [28]. With this improvement attempt, we concluded that JVC is inherently sequential with CPU implementations clearly outperforming GPU implementations.

VI. Results

As discussed in sec. IV, We test our implementation with 5 baselines. As expected from literature, Hungarian algorithm has strong correlation with range values while JVC has weak correlation. The reason for JVC being correlated is the initial covering of zero values, more zeros covered in the beginning results in lesser number of iterations which reflects in faster runtime. As for Hungarian Algorithm, since multiple pivots are possible and all implementations taking advantage from those there is a strong correlation between range and runtimes. Fig. 5 shows the correlation between runtime and range

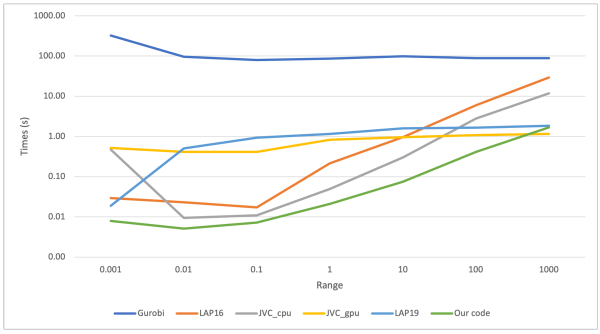

Fig. 5: Variation of runtime with range for problem size N = 4096

for different problem sizes.

Another supporting parameter for testing the expected slow-

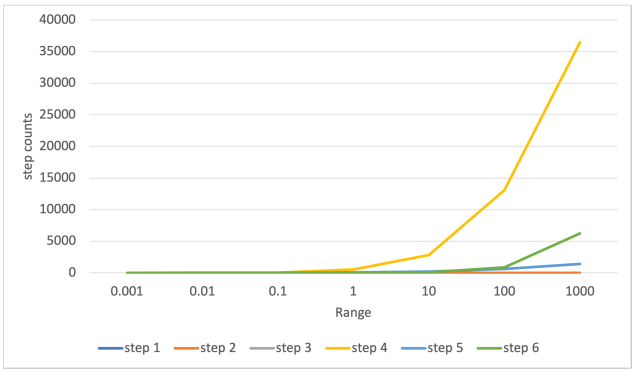

Fig. 6: step counts with range for Hungarian Algorithm

down with range is the number of iterations. Since, there are less zeros generated every iteration it takes dual update step each time to force generate zeros in order to proceed with the algorithm.

Step #	Range						
	0.001	0.01	0.1	1	10	100	1000
step 1	1	1	1	1	1	1	1
step 2	2	2	2	2	2	2	2
step 3	4	7	12	80	200	619	1399
step 4	3	6	14	544	2806	13106	36493
step 5	3	6	11	79	199	618	1398
step 6	0	0	0	10	82	844	6210

TABLE III: step counts with range for Hungarian Algorithm

Another important aspect of our Hungarian implementation is it's performance as compared to JVC algorithm. It is clear from tab. IV - tab. X that our implementation is clearly the best Hungarian algorithm implementation in terms of runtime speedup. Unfortunately due to strong relation of Hungarian algorithm with range the runtime drastically increases and performs slower than JVC_cpu [8] and JVC_gpu [28].
As discussed in sec. V the most time consuming step for Hungarian algorithm is the dual update step i.e. step 6. The same can be seen in fig 7. Dual update is indeed a weakness for Hungarian algorithm $\mathcal{O}(n^2)$ steps while it is a strength for JVC $\mathcal{O}(n)$ steps. While JVC seeming beneficial, the minimum shortest path requirement for JVC is very demanding specially on GPUs hence being difficult to accelerate and causing insufficient GPU utilization and slower times as compared to CPU. A hybrid approach which can switch between Hungarian and JVC shortest path algorithm can be thought of but it still needs $\mathcal{O}(n^2)$ operations causing similar overheads. Though the exact performance impact still remains unknown.

VII. Conclusion

We developed a state of the art CUDA implementation of Hungarian Algorithm which handsomely beats existing Hungarian Algorithm implementations. We achieve speedups of upto $4.41x$ compared to the best baseline. While we achieve geometric mean speedup of $7.8x$ compared to the best Hungarian Algorithm implementation. We use a harsh

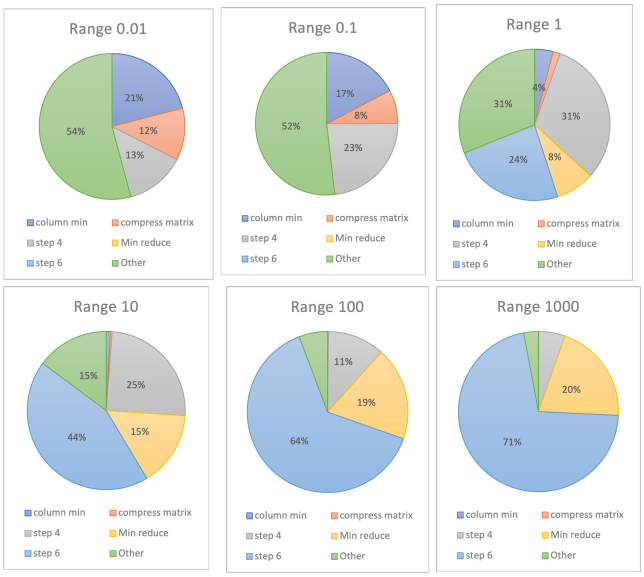

Fig. 7: split up of different dominant kernels with increasing range for problem-size N = 8192

process of comparing with the best baseline solver (though not one solver best performs to all problems) and still achieve a geometric mean speedup of $2.62x$ for problem size $N = 8192$ with range from $0.001to100$. We also experiment with smart initialization strategies like [29] and Vogel's regret strategies to improve initial coverage. We showed that these strategies fail to deliver performance gains due to constraints/opportunities provided by the underlying massively parallel hardware and end up being overheads.
This report also provides an insight of most time taking steps for different algorithms. We use all cuda optimizations in [1] to make sure that there is no significance performance impact of cuda implementation itself which can show a different image of times to the system profiler. We also survey different implementations of linear assignment problem solvers including Hungarian, JVC, auction and flow based implementations both on CPU and GPU (if available). Though we kindle the idea of a hybrid Hungarian-JVC implementation the details of it still need to be clearly worked out. We keep this as scope for future work.

APPENDIX

LAP - Linear Assignment Problem
GPU - Graphics Processing unit
CPU - Central Processing Unit
GPGPU - General Purpose Graphics Processing Unit
CUDA - Compute Unified Device Architecture
SM - Streaming Multiprocessor
BFS - Breadth First Search
HA - Hungarian Algorithm
JVC - Jonker, Volgenant and Castanon Algorithm

REFERENCES

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

[2] K. Date and R. Nagi, "Gpu-accelerated hungarian algorithms for the linear assignment problem," *Parallel Computing*, vol. 57, pp. 52–72, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016781911630045X

[3] V. G. Samiran Kawtikwar, Yen-Hsiang Chang, "Improvements to hungarian lap solver," 2021, unpublished.

[4] R. Jonker and A. Volgenant, "A shortest augmenting path algorithm for dense and sparse linear assignment problems," *Computing*, vol. 38, no. 4, pp. 325–340, Dec 1987. [Online]. Available: https://doi.org/10.1007/BF02278710

[5] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, vol. 2, no. 1-2, pp. 83–97, 1955. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/nav.3800020109

[6] E. L. Lawler, *Combinatorial optimization : networks and matroids*. New York: Holt, Rinehart and Winston, 1976. [Online]. Available: http://catdir.loc.gov/catdir/enhancements/fy0636/76013516-t.html

[7] D. P. Bertsekas, "A new algorithm for the assignment problem," *Mathematical Programming*, vol. 21, pp. 152–171, 1981.

[8] R. Jonker and T. Volgenant, "Improving the hungarian assignment algorithm," *Operations Research Letters*, vol. 5, no. 4, pp. 171–175, 1986. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0167637786900738

[9] J. Edmonds and R. M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems," *J. ACM*, vol. 19, no. 2, p. 248–264, apr 1972. [Online]. Available: https://doi.org/10.1145/321694.321699

[10] N. Tomizawa, "On some techniques useful for solution of transportation network problems," *Networks*, vol. 1, no. 2, pp. 173–194, 1971. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/net.3230010206

[11] R. E. Burkard, *Assignment problems*. Philadelphia, Pa: Society for Industrial and Applied Mathematics SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104, 2009.

[12] R. S. Barr, F. Glover, and D. Klingman, "The alternating basis algorithm for assignment problems," *Mathematical Programming*, vol. 13, no. 1, pp. 1–13, Dec 1977. [Online]. Available: https://doi.org/10.1007/BF01584319

[13] M. S. Hung and W. O. Rom, "Solving the assignment problem by relaxation," *Operations Research*, vol. 28, no. 4, pp. 969–982, 1980. [Online]. Available: http://www.jstor.org/stable/170335

[14] L. F. McGinnis, "Implementation and testing of a primal-dual algorithm for the assignment problem," *Operations Research*, vol. 31, no. 2, pp. 277–291, 1983. [Online]. Available: https://doi.org/10.1287/opre.31.2.277

[15] J. E. Hopcroft and R. M. Karp, "An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs," *SIAM Journal on Computing*, vol. 2, no. 4, pp. 225–231, 1973. [Online]. Available: https://doi.org/10.1137/0202019

[16] H. Alt, N. Blum, K. Mehlhorn, and M. Paul, "Computing a maximum cardinality matching in a bipartite graph in time $o(n^{1.5}\sqrt{(m/\log(n))}$," *Inf. Process. Lett.*, vol. 37, pp. 237–240, 1991.

[17] M.-Y. Kao, T.-W. Lam, W.-K. Sung, and H.-F. Ting, "A decomposition theorem for maximumweight bipartite matchings with applications to evolutionary trees," in *Algorithms - ESA' 99*, J. Nešetřil, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 438–449.

[18] M. Balinski and R. E. Gomory, "A primal method for the assignment and transportation problems," *Management Science*, vol. 10, pp. 578–593, 1964.

[19] E. Dinic and M. cronrod, "An algorithm for solution of the assignment problem," *Soviet Math Dokl.*, vol. 10, pp. 1324–1326, 1969.

[20] W. H. Cunningham, "A network simplex method," *Mathematical Programming*, vol. 11, no. 1, pp. 105–116, Dec 1976. [Online]. Available: https://doi.org/10.1007/BF01580379

[21] M. Akgül, "A genuinely polynomial primal simplex algorithm for the assignment problem," *Discrete Applied Mathematics*, vol. 45, no. 2, pp. 93–115, 1993. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0166218X9390054R

[22] A. V. Goldberg, S. A. Plotkin, and P. M. Vaidya, "Sublinear-time parallel algorithms for matching and related problems," 1988.

[23] J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan, "Relaxed heaps: An alternative to fibonacci heaps with applications to parallel computation," *Commun. ACM*, vol. 31, no. 11, p. 1343–1354, nov 1988. [Online]. Available: https://doi.org/10.1145/50087.50096

[24] H. N. Gabow and R. E. Tarjan, "Almost-optimum speed-ups of algorithms for bipartite matching and related problems," in *STOC '88*, 1988.

[25] U. Schwiegelshohn and L. Thiele, "A systolic array for the assignment problem," *IEEE Transactions on Computers*, vol. 37, no. 11, pp. 1422–1425, 1988.

[26] A. J. Hoffman and H. M. Markowitz, "A note on shortest path, assignment, and transportation problems," *Naval Research Logistics Quarterly*, vol. 10, no. 1, pp. 375–379, 1963. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/nav.3800100132

[27] M. Fayyazi, D. Kaeli, and W. Meleis, "Parallel maximum weight bipartite matching algorithms for scheduling in input-queued switches," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, 2004, pp. 4–.

[28] S. Guthe and D. Thuerck, "Algorithm nbsp;1015: A fast scalable solver for the dense linear (sum) assignment problem," *ACM Trans. Math. Softw.*, vol. 47, no. 2, apr 2021. [Online]. Available: https://doi-org.proxy2.library.illinois.edu/10.1145/3442348

[29] E. Munapo, "Development of an accelerating hungarian method for assignment problems," *Eastern-European Journal of Enterprise Technologies*, vol. 4, no. 4 (106), p. 6–13, Aug. 2020. [Online]. Available: http://journals.uran.ua/eejet/article/view/209172

Name	Size							
	64	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**
Gurobi	8.439	32.645	151.949	706.992	3614.36	58170.1	324376	out of mem
LAP16	0.198	0.253	0.413	0.742	1.464	10.682	29.264	58.208
LAP19	0.369	0.436	0.561	1.031	2.198	3.64	18.786	43.105
JVC_cpu	0.162	1.338	2.691	4.407	26.324	125.492	474.599	1842.94
JVC_gpu	174	164	177	190	201	279	521	1452
Our code	0.414	0.568	0.622	3.487	4.813	3.412	7.888	16.642

TABLE IV: Time (in ms) for different problem sizes. Range = 0.001

Name	Size							
	64	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**
Gurobi	10.933	33.165	258.186	509.853	2466.77	11718	95954.1	out of mem
LAP16	0.512	0.245	0.419	3.23	4.75	8.539	22.936001	56.57
LAP19	0.313076	0.410895	0.493	2.763	6.456	6.44	9.448	18.126
JVC_cpu	0.086	0.299	1.71	6.205	19.586	92.64	411.712	1617.43
JVC_gpu	223	134	134	140	164	242	503	1436
Our code	1.497	0.884	1.453	2.375	2.64	6.262	5.091	9.958

TABLE V: Time (in ms) for different problem sizes. Range = 0.01

Name	Size							
	64	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**
Gurobi	6.971	23.527	137.993	461.581	2012.14	10436	79257.6	out of mem
LAP16	2.248	2.488	3.407	5.021	6.091	9.495	17.159	42.937
LAP19	1.76927	2.60547	4.08282	6.794	7.068	8.084	10.885	19.465
JVC_cpu	0.074	0.251	0.73	4.625	17.477	86.663	415.379	1751.18
JVC_gpu	219	130	139	149	189	375	928	3319
Our code	2.098	2.433	3.538	4.74	4.216	5.996	7.191	12.235

TABLE VI: Time (in ms) for different problem sizes. Range = 0.1

Name	Size							
	64	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**
Gurobi	7.59	19.29	107.97	470.41	2008.00	11160.60	86774.40	out of mem
LAP16	9.56	14.93	24.09	44.49	61.72	102.96	213.45	519.70
LAP19	7.09	6.90	10.75	10.35	15.34	24.72	49.26	130.65
JVC_cpu	0.07	0.26	1.54	6.18	32.10	151.60	826.08	4668.32
JVC_gpu	228.00	138.00	144.00	176.00	237.00	539.00	1158.00	3680.00
Our code	8.63	8.76	9.30	11.31	11.47	15.61	21.15	44.04

TABLE VII: Time (in ms) for different problem sizes. Range = 1

Name	Size							
	64	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**
Gurobi	7.65	18.05	114.89	503.53	2030.60	9837.59	98982.40	out of mem
LAP16	22.55	29.75	85.13	143.32	294.63	478.55	957.58	2628.37
LAP19	11.72	11.81	25.39	33.85	84.32	141.73	300.13	644.31
JVC_cpu	0.10	0.24	1.27	7.13	40.37	196.20	952.54	5503.75
JVC_gpu	225.00	135.00	154.00	204.00	311.00	701.00	1585.00	4071.00
Our code	9.91	10.19	13.21	14.59	20.87	33.98	74.77	192.19

TABLE VIII: Time (in ms) for different problem sizes. Range = 10

Name	Size							
	64	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**
Gurobi	8.11	20.19	94.84	499.53	2059.73	11479.50	88761.10	out of mem
LAP16	40.61	54.81	175.97	410.31	895.52	2385.90	5998.52	23084.58
LAP19	12.88	19.58	70.89	188.67	439.78	1152.71	2781.98	5784.26
JVC_cpu	0.13	0.26	2.72	6.75	31.00	191.67	1079.73	6294.39
JVC_gpu	224.00	135.00	151.00	217.00	316.00	740.00	1651.00	5756.00
Our code	10.74	12.43	21.50	31.28	56.64	141.30	412.98	1306.27

TABLE IX: Time (in ms) for different problem sizes. Range = 100

Name	Size							
	64	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**
Gurobi	5.54	18.14	90.49	472.63	2034.38	12428.00	89106.20	out of mem
LAP16	41.87	56.88	219.22	579.17	1690.49	5666.52	29388.12	149267.97
LAP19	14.35	20.65	79.03	252.65	1073.51	3919.37	11793.90	36263.90
JVC_cpu	0.20	0.27	2.08	7.45	35.62	196.41	1153.80	4961.64
JVC_gpu	230.00	138.00	160.00	194.00	353.00	686.00	1848.00	5147.00
Our code	11.80	12.92	23.87	40.15	107.67	384.54	1687.54	8669.43

TABLE X: Time (in ms) for different problem sizes. Range = 1000