

Improvements to the Hungarian LAP Solver

Samiran Kawtikwar
Dept. of Industrial and
Systems Engineering
samiran2@illinois.edu

Yen-Hsiang Chang
Dept. of Electrical and
Computer Engineering
yhchang3@illinois.edu

Varun Govind
Dept. of Electrical and
Computer Engineering
varung2@illinois.edu

Abstract—The linear assignment problem (LAP) is the foundation for combinatorial optimization with many applications. Many algorithms have been proposed in the literature and the most famous one is the Hungarian algorithm based on a primal-dual formulation. In this report, we compare two state-of-the-art GPU-accelerated Hungarian LAP solvers and present optimizations for the better implementation. Our optimizations explore opportunities for reducing memory contention and improving cache performance, by converting atomic operations to reductions and applying kernel fusion techniques, respectively. We also investigate instruction-level optimizations about register allocations and synchronization strategies. Our evaluation shows that our optimized implementation outperforms the original GPU-accelerated Hungarian LAP solver by a geometric mean of 7.93 times speedup on randomly generated problem instances across various problem sizes.

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

I. INTRODUCTION

The linear assignment problem (LAP) is a fundamental problem in combinatorial optimization and has vast applications, such as designing transportation systems and approximating the quadratic assignment problem. 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 the total cost of the assignment complying with the constraint that a bijection is formed between the resources and the tasks in the assignment.

Many algorithms have been proposed for the LAP in the literature. We focus on the Hungarian algorithm proposed by Kuhn [1] based on a primal-dual formulation of the LAP, since the nature of operating directly on the cost matrix is well suited for parallelization on GPU. There are two variants for the Hungarian algorithm: (1) the classical version developed by Munkres [2]; and (2) the alternating tree version developed by Lawler [3]. Both variants work on augmenting path search phases to adjust the assignment and dual update phases to introduce new candidates. On the one hand, the classical version has a worst-case complexity of $O(n^4)$ with the bottleneck on dual update phases. On the other hand, the alternating tree version has a worst-case complexity of $O(n^3)$ with bottlenecks on both augmenting path search phases and dual update phases. However, both versions are of interests for GPU implementations, since it is rare for the classical version to be bounded by the dual update phases in real word scenarios and the alternating tree version has a disadvantage on irregular

memory accessing patterns. More detailed explanations to the Hungarian algorithm can be found in the appendix.

There are two state-of-the-art GPU-accelerated Hungarian LAP solvers in the community. One was proposed by Date and Nagi [4] in 2016 based on the alternating tree version of the Hungarian algorithm and the other was proposed by Lopes et al. [5] in 2019 based on the classical version of the Hungarian algorithm. We will refer to the solver proposed by Date and Nagi as LAP16 and the solver proposed by Lopes et al. as LAP19 in the rest of this report for simplicity. Both solvers contributed an efficient parallelization of augmenting path search phases in the Hungarian algorithm, where the LAP16 solver exploited race conditions to generate vertex-disjoint augmentation paths and the LAP19 solver developed a fine-grained implementation to alleviate load imbalance by processing edges (dual variables) instead of vertices (primal variables) in parallel.

In this report, we compare the LAP16 solver with the LAP19 solver and choose the LAP19 solver as our baseline. Our optimizations first reduce global memory contention by converting atomic operations to reduction. On top of that, we apply kernel fusion techniques to alleviate kernel launch overheads and explore data locality. We also investigate instruction-level optimizations to avoid inadequate register allocations and synchronization strategies deduced by the compiler. As a result, we achieve a geometric mean of 7.93 times speedup on randomly generated problem instances across various problem sizes by combining these optimizations.

The rest of the report is organized as follows. In Section II, we review the literature of LAP solvers and touch upon other parallel implementations. In Section III, we describe our evaluation methodologies, compare the LAP16 solver and the LAP19 solver, and determine the LAP19 solver as our baseline. In Section IV, we elaborate the intuition, the implementations, and the incremental improvements for our optimizations on the LAP19 solver. In Section V, we present our experiment results on randomly generated problem instances with various problem sizes. Finally, in section VI, we conclude the report with a summary.

II. 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 $C : A \times T \rightarrow \mathbb{R}$, the objective

of the LAP is to find a bijection $f : A \rightarrow T$ 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 optimization problems that can be solved in polynomial time. There have been many efficient algorithms proposed in the literature. According to Burkard and Çela [6], these algorithms can be broadly categorized into three main classes:

- 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 [1] and the Auction algorithm [7]; and
- 3) Dual algorithms such as the successive shortest path algorithm [8].

Due to their polynomial worst-case complexity, the primal-dual and dual algorithms generally outperform the simplex algorithms. The theoretical complexity of most efficient primal-dual or dual algorithms is $O(n^3)$, where n is the number of assignments to be done. Owing to the cubic worst-case complexity, the sequential nature of these algorithms make them less attractive for LAP instances of large sizes. There have been several developments in parallel versions of aforementioned sequential algorithms, including parallel asynchronous version of the Hungarian algorithm [9], parallel version of the shortest path algorithm [10] and parallel version of the Auction algorithm [11], [12]. An empirical analysis of the sequential and parallel versions of the Auction and shortest path algorithms was performed by Kennington and Wang [13] and the above parallel algorithms were shown to achieve significant speedups.

In recent years, the significant advancement in graphics processing hardware and massively parallel architectures provides a cost-effective solution for high performance computing applications. Vasconcelos and Rosenhahn [14] developed a parallel version of the synchronous Auction algorithm for a single GPU. Roverso et al. [15] developed a GPU implementation of the deep greedy switching (DGS) heuristic, which achieved significant speedup by sacrificing optimality and was able to solve problem instances of 100 million variables.

In spite of the flourishing GPU solutions, there were no GPU implementations of the Hungarian algorithm until 2016. When Date and Nagi [4] developed the first known parallel implementation of the Hungarian Algorithm. They parallelized augmenting path search phases in the Hungarian algorithm by exploiting race conditions to generate multiple vertex-disjoint augmenting paths. These augmenting paths can be used to improve the current solution in parallel and lead to significant speedups in the execution time. In 2019, Lopes et al. [5] further improved by distributing augmenting path search phases to multiple blocks in order to minimize global device synchronization. We will refer to the solver proposed by Date

and Nagi as LAP16 and the solver proposed by Lopes et al. as LAP19 in the rest of this report for simplicity.

In this report, we focus on the LAP16 solver and the LAP19 solver since the nature of operating directly on the cost matrix in the Hungarian algorithm is well suited for parallelization on GPU. We will compare these two solvers and optimize the better implementation in the following sections.

III. 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 $n = 256, 512, 1024, 2048$, and 4096 to cover small and large problem instances. The entries in the $n \times n$ cost matrix are stored as double-precision floating-point formats and are generated uniformly randomly in the range $[0, 1000n]$, with a fixed seed in order to achieve reproducibility between experiments.

C. Baseline Selection

The baseline is selected among the LAP16 GPU-solver by Date and Nagi [4], the LAP19 GPU-solver by Lopes et al. [5], and a simple LAP CPU-solver implemented by us using the Gurobi optimizers [16], by comparing their end-to-end execution time on the datasets mentioned above. As Fig. 1 shows, we can see that the existing GPU-solvers (LAP16 and LAP19) both yield a significant amount of speedup versus the CPU-solver. One interesting thing is that the LAP16 solver is actually slower than the CPU-solver for certain small problem instances as shown by the log-scale plot. This is probably due to the difference in execution speed between the GPU and the CPU as well as some GPU overheads in launching kernels and synchronization. Regardless of the small problem instances, for much larger problem instances the GPU-solvers are much faster with the LAP19 solver winning across all problem sizes. Thus, we will use the LAP19 solver as our baseline program and perform all our optimizations on top of it.

D. Optimization Procedure

Our evaluation and experimental methodology consists of several steps with a simple loop for optimization. The loop allows us to consistently evaluate optimizations with respect to previous ones and measure the incremental performance improvement. Consequently, we can clearly measure where

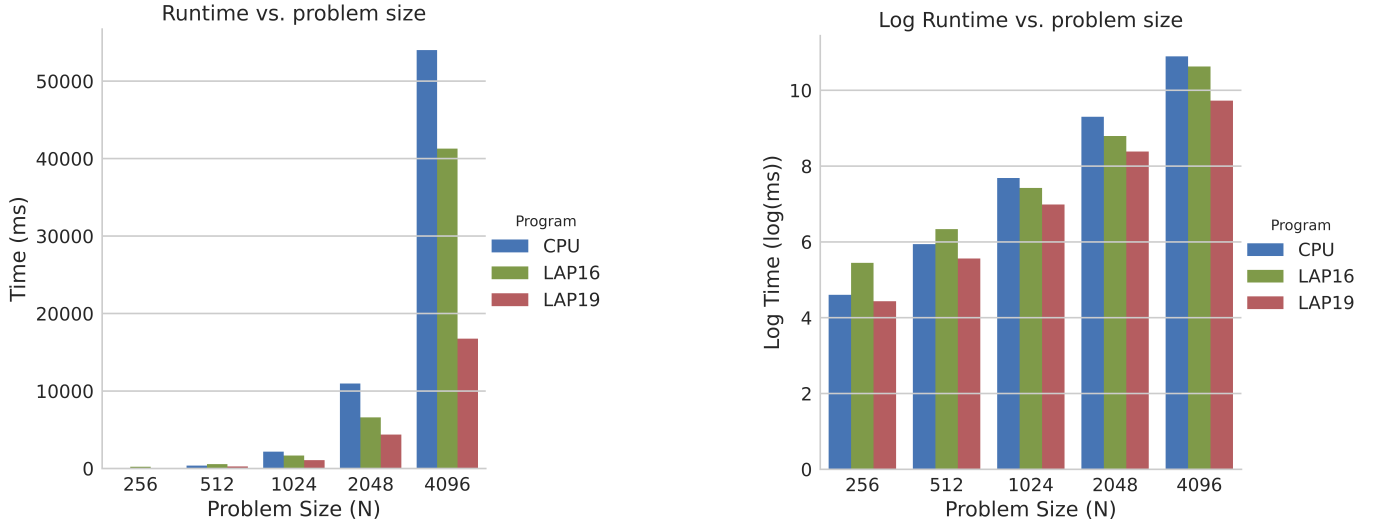


Fig. 1: This figure compares the runtimes of the three LAP solvers where the left sub-figure shows normal runtimes and the right sub-figure shows the log-scale runtime for various problem sizes.

the bottlenecks are shifting to as a result of each optimization. The steps of our methodology are:

- 1) First, we profile the code systematically gathering timings, API calls, and memory usage per kernel to find greatest bottlenecks and determine where the most effective optimizations could be performed.
- 2) Next, we perform the optimization by alleviating bottlenecks.
- 3) Finally, we evaluate the code at a system-level again after adding the optimization and determine if further optimizations can be performed. If so, we return to step 1 to dig deeper into the code.

E. Profiling Tools

We use various tools in our evaluation to gather profiling data. 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.

IV. OPTIMIZATIONS

This section describes our optimizations on the LAP19 solver, including converting atomic operations to reductions, fusing kernels, improving code efficiency and tuning with synchronization barriers. In order to explain how we use profiling tools to identify opportunities for optimizations, we will only focus on the results of the largest problem size we are dealing with, which is $n = 4096$, in this section. The complete evaluation results across varying problem sizes can be found in Section V. In addition, since there are dependencies between optimizations, each optimization is implemented on top of the

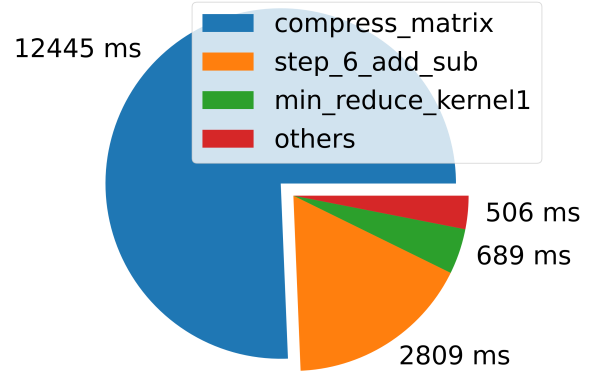


Fig. 2: Execution time of the most time-consuming kernel calls aggregated by name for the baseline implementation of the LAP19 solver for problem size of $n = 4096$.

previous optimization and compare its improvement directly with the previous result.

To understand the bottleneck of the LAP19 solver, we first put timers between kernel calls to obtain the execution time aggregated by kernel names. As Fig. 2 shows, the most time-consuming kernel is the `compress_matrix` kernel which takes about 75% of the execution time.

A. Optimization 0: Atomic Operations to Reductions

Since Fig. 2 shows that the bottleneck of the baseline is the `compress_matrix` kernel, we first delve into this kernel and try to identify the cause of exceptionally long execution time. Listing 1 shows the code snippet of the

```

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.

`compress_matrix` kernel in the baseline implementation. It is easy to see that the exceptionally long execution time is introduced by the two atomic add operations. We then observe that the contention of the first atomic operation is heavier than the second one, since all threads compete for a single global memory address in the first atomic operation while the contention of the second atomic operation is distributed to multiple bins.

In order to improve the performance of this kernel, we need to alleviate global memory contention from the first atomic add operation. From the implementation, we observe that the `zeros_size` variable is just the total sum of the `zeros_size_b` array. Therefore, in optimization 0, we remove the atomic add operations on the `zeros_size` variable and introduce a new `add_reduction` kernel which performs parallel reduction on the `zeros_size_b` array then stores the result to the `zeros_size` variable. As Fig. 3 shows, after alleviating global memory contention in the `compress_matrix` kernel, the execution time of this kernel is reduced by 88%. Also note that the new `add_reduction` kernel is classified as the “others” category in Fig. 3 since the number of bins is not large in the baseline implementation hence it does not take too long to perform reductions.

In summary, since atomic operations are expensive in parallel applications due to the serial nature, we decide to alleviate global memory contention in the `compress_matrix` kernel by converting atomic add operations to reductions. Comparing with the baseline, the result in optimization 0 shows that we achieve a speedup of 8.35 times for the `compress_matrix` kernel and a speedup of 3.07 times for the whole application.

B. Optimization 1: Kernel Fusion

As shown by Fig. 3 that the bottleneck after applying optimization 0 becomes the `step_6_add_sub` kernel, we then shift our focus to this kernel. Listing 2 shows the code snippet of the `step_6_add_sub` kernel. At first glance, there is no optimization we can apply to this kernel alone, since there is no opportunity to reuse memory and the computations are necessary. However, we observe that there is a chance to exploit L1 caches in streaming multiprocessors by fusing the `step_6_add_sub` kernel and the `compress_matrix` kernel, since the `slack[i]` variable appears in both of them. Therefore, we apply kernel fusion techniques by replacing the `step_6_add_sub` kernel and the `compress_matrix` kernel with a new `add_sub_compress_matrix` kernel, in

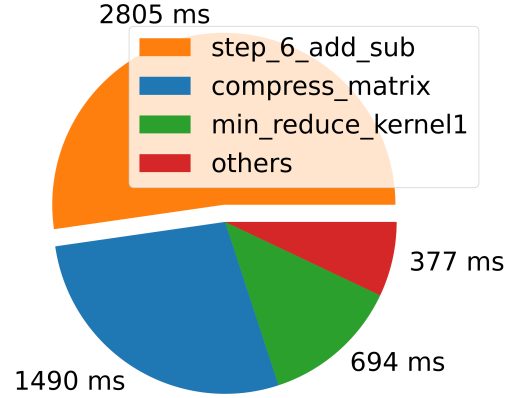


Fig. 3: Execution time of the most time-consuming kernel calls aggregated by name for the implementation up to optimization 0 for problem size of $n = 4096$.

order to reduce access to global memory, as shown in Listing 3.

In order to understand why applying kernel fusion might help in this scenario, we also use NVIDIA Nsight Compute to investigate the difference between the original two kernels and the new fused kernel. Table I shows the profiling results for the kernels focusing on the L1 hit rate, the compute throughput and the memory throughput. We observe that the `step_6_add_sub` kernel has a high L1 hit rate coming from the `cover_row` and `cover_column` arrays instead of the `slack` array. However, it has low compute throughput and memory throughput since the work is not enough to hide latency. On the contrary, the compute and memory units are well-utilized in the `compress_matrix` kernel by having high throughput, but the L1 hit rate is roughly 0% due to global memory access to the `slack` array. As a result, the fused kernel `add_sub_compress_matrix` kernel is balanced between the original two kernels and has moderate L1 hit rate and throughput.

In summary, applying kernel fusion techniques not only takes advantage of L1 caches in multiprocessors, but also generates enough work for the fused kernel to hide latency. Fig. 4 shows the aggregated kernel time after applying kernel fusion. Comparing with optimization 0, the result in optimization 1 shows that we achieve a speedup of 1.45 times for the fused kernel and a speedup of 1.30 times for the whole application.

C. Optimization 2: Cooperative Groups

The idea of cooperative groups was introduced with CUDA 9.0 which enabled ability to synchronize across threads in different thread blocks. This eliminates the old way of using kernel boundaries as means to synchronize across the grid.

As shown in Fig. 7 in the appendix, the Hungarian Algorithm is iterative in nature with the flow moving between

TABLE I: Profiling results using NVIDIA Nsight Compute for the `step_6_add_sub` kernel, the `compress_matrix` and the `add_sub_compress_matrix` kernel, focusing on the L1 hit rate, the compute throughput and the memory throughput.

Kernel Name	L1 Hit Rate	Compute Throughput	Memory Throughput
<code>step_6_add_sub</code>	77%	24%	31%
<code>compress_matrix</code>	0.03%	67%	76%
<code>add_sub_compress_matrix</code>	44%	55%	48%

```
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.

```
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.

different steps until some conditions are met. The baseline implementation uses the host to verify these conditions and aggressively launches small kernels accordingly.

Fig. 5 shows that the kernel launch overhead is about 48% of all CUDA API calls for the problem size of $n = 4096$. Apart from the launch overhead, each kernel calculates global indices for each thread separately as well as the cached data are lost with invocation of new kernels causing higher memory bandwidth. Therefore, there is an opportunity to fuse kernels more aggressively using cooperative groups and get some locality optimization.

To fuse kernels with different requirements for the number of threads per block and the number of blocks in the grid, we need to query the device for the best configuration using the `cudaOccupancyMaxPotentialBlockSize(...)` CUDA API call. It returns the number of threads per block that maximizes occupancy per streaming multiprocessor (SM) and the total number of blocks that can reside concurrently on all SMs. This helps compute units stay active for kernel execution.

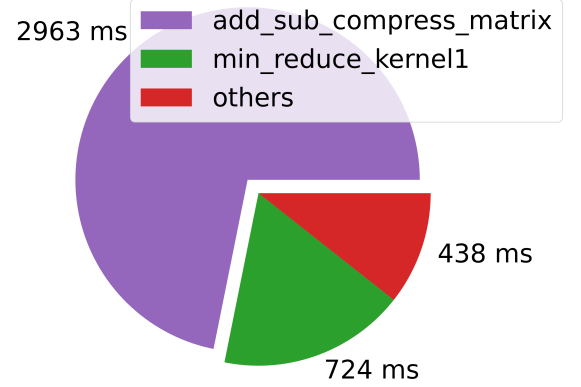


Fig. 4: Execution time of the most time-consuming kernel calls aggregated by name for the implementation up to optimization 1 for problem size of $n = 4096$.

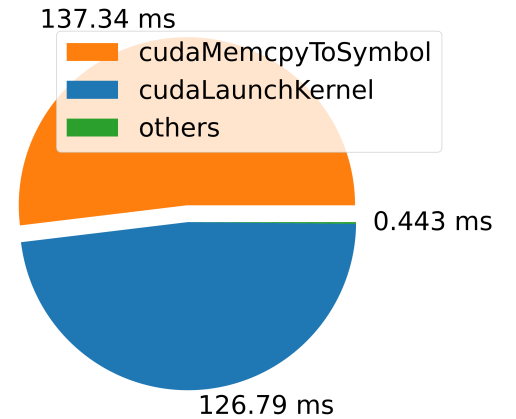


Fig. 5: Execution time for all API calls up to optimization 1 for problem size of $n = 4096$.

As a result, these techniques help to fuse all kernels into one cooperative launch and enables memory reuse as well as better L1 hit rates. Comparing with optimization 1, the result in optimization 2 shows that we achieve a speedup of 1.01 times for the whole application.

D. Optimization 3: Code Efficiency

Although we fused several kernels into one using cooperative groups in optimization 2, there was no significant change to the code body. Therefore, the conclusion from optimization 1 that the `add_sub_compress_matrix` part dominates the application is still valid. Hence, we decide to dig deeper into the code and improve more.

Recall from Listing 3 that one of the bottleneck in the `add_sub_compress_matrix` kernel is the global memory access to the large `slack` array since there is no memory reuse among threads. Unfortunately, this is due to the nature of the algorithm hence we can not improve more from this perspective. Moreover, it seems hard to optimize memory reuse in a single thread because the `slack[i]` variable only appears twice in the code. Even if we try to load the `slack[i]` variable into a register and write it back to the global memory later, we still need to access the memory twice. However, this does not agree with the profiling result from NVIDIA Nsight Compute. In fact, the SASS code from NVIDIA Nsight Compute shows that there are three memory instructions on the `slack[i]` variable, which are two LDG instructions and one STG instruction, instead of only two instructions. The reason is that the compound assignment operator in `slack[i] += d_min_in_mat` is indeed expanded as an addition operator and an assignment operator, which becomes `slack[i] = slack[i] + d_min_in_mat`. Therefore, as shown by the analysis above, using a register to hold the `slack[i]` variable can actually alleviate the bottleneck.

Listing 4 shows the optimized implementation for the `add_sub_compress_matrix` kernel, where we load the `slack[i]` variable into a register, update the register and propagate the value back to the global memory in the switch statement, and only use the register value for the if statement. The profiling result from NVIDIA Nsight Compute further validates our optimization, where the number of global memory load instructions drops from 231 millions to 1.94 millions, which is a reduction of 16%.

In summary, implementing efficient code by having optimizations based on the instruction level is important when tackling with simple code because a small change in the high-level code can significantly affect the number of instructions executed. Since compilers might not have enough hints and knowledge to optimize the code, it is our responsibility to utilize profiling tools to check if instructions executed, branch divergence and register utilization meet our expectation in the low-level code. In addition to the decrease of the number of global memory instructions by applying this optimization, the improvement of the L1 hit rate in the `add_sub_compress_matrix` part from 44% to 54% also contributes to the speedup. Comparing with optimization 2, the result in optimization 3 shows that we achieve a speedup of 1.13 times for the `add_sub_compress_matrix` part and a speedup of 1.05 times for the whole application.

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

if (near_zero(reg)) {
  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 4: Snippet of the `add_sub_compress_matrix` kernel in optimization 3 where we use a register to alleviate access to the global `slack[i]` variable.

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

__syncthreads();

if (near_zero(reg)) {
  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 5: Snippet of the `add_sub_compress_matrix` kernel in optimization 4 where we add an extra synchronization.

E. Optimization 4: Synchronization Strategy

The last optimization is an interesting but counter-intuitive observation we get accidentally when playing with the code, hence there is no intuition on why we want to even try it. As Listing 5 shows, this optimization adds a redundant synchronization barrier, `__syncthreads()`, in the `add_sub_compress_matrix` kernel and this speeds up the whole application by 1%. It is hard to rationalize this behavior by barely looking at the code, since synchronization barriers usually only add overheads and slows down the application as all threads need to wait for the slowest one. Therefore, we again use NVIDIA Nsight Compute to profile the kernel in order to get more information to justify this result.

By comparing the profiling results from NVIDIA Nsight Compute, there is one entry that attracts our attention. We observe that adding this redundant synchronization barrier brings the L1 hit rate from 54% to 58%. We suspect that this behavior is related with the divergent atomic operations. If there is no synchronization, the frontier of warps is not flat

as they are not executed in the same pace, since only some warps in a block need to perform atomic operations while the other do not. This hurts the utilization of L1 caches since data locality only exists for neighboring iterations, but the absence of synchronization leads to an imbalanced pace among warps in a block. This concludes why adding an synchronization barrier slightly improves the performance since the flat frontier of warps is beneficial to L1 caches.

In summary, comparing with optimization 3, the result in optimization 4 shows that we achieve a speedup of a speedup of 1.03 times for the whole application.

V. EXPERIMENTAL RESULTS

In terms of results, each optimization had varying degrees of improvement for different problem sizes. Some optimizations gave more of a performance boost than others and depending on the problem size, some slightly decreased the performance. In fact, we can observe this visually through the log-scale plot at the right hand side of Fig. 6, where optimization 1 slightly hurts performance in problem sizes from 256 to 1024. However, despite hurting performance marginally, nearly all the optimizations gave a sizable boost with the largest speedups occurring in optimization 0.

Table II shows runtimes for each optimization per problem size and additionally shows the incremental time Δt , which is the difference in runtime between the previous optimization and the current optimization. Hence, a positive value shows a performance boost while a negative value shows a loss in performance. If we compare runtimes for optimization 4 with the runtimes for the baseline, we can see that cumulatively the overall program runtime was reduced by a large amount. In fact, the final optimized implementation outperforms the baseline LAP19 solver by a geometric mean of 7.93 times. Table III shows the factor of speedups compared with the baseline. Specifically, the total speedup compares the current optimization with the baseline while the incremental speedup compares the current optimization with the previous. In the total speedup column of optimization 4, we can clearly see that we achieved the highest speedup for the problem size of 1024 by a factor of 11.63, while the lowest factor of speedup occurs for the largest problem size we evaluated, i.e., $n = 4096$. The largest factor of incremental speedups were gained with optimization 0 with a geometric mean of 4.65 times across all problem sizes; this accounts for more than half of our total speedup for our final optimized implementation.

From the tables and figures, it is evident that some optimizations hurt the performance, albeit the performance loss is very small compared to the total reduction in runtime. It can be inferred that because the matrix is smaller, less calculations need to be performed so the costs, in terms of overheads, starts to dominate the runtime. This is because the overhead cost remains relatively the same for all of the problem sizes; thus it becomes a larger factor with respect to total runtime in problems of smaller sizes. In other words, the compute latency is not high enough to hide the overhead latency for smaller problems. Recall in optimization 1, we begin implementing

kernel fusion. While fusing the kernels reduces the overhead of calling multiple kernels to one kernel, there remains overhead latency with respect to thread synchronization. This overhead latency could be causing the marginal increase in runtime for smaller problem sizes. While the kernel fusions do add some extra overhead in terms of synchronization and less control in terms of kernel launch parameters, there is a obvious benefit as we do not have to ping-pong between GPU and CPU to launch and synchronize each kernel. Since the performance decrease is marginal (with respect to the overall runtime improvement) for smaller problem sizes, it is an acceptable trade-off especially because there is a clear benefit for the larger problem sizes.

VI. CONCLUSION

This report improves on the cutting edge GPU implementation of the Hungarian Algorithm by Lopes et al. [5]. Overall, each optimization was implemented as a result of iteratively profiling the code and measuring bottlenecks using classical timing constructs along with available tools from NVIDIA. We introduce five optimizations using simple concepts like privatization, locality optimization and coarsening; as a result, we are able to achieve a geometric mean speedup of 7.93 times for different problem sizes over the original implementation.

Some future directions to explore would be integrating dynamic parallelism or implementing it separately with other optimizations other than kernel fusion and then comparing it with fused kernel optimizations. We may be able to observe faster runtimes if there are opportunities for asynchronous kernels. With current limiting factors on number of floating point operations, any significant improvement needs algorithmic inputs to the algorithm such as heuristics giving better-quality initial solutions and reducing the number of iterations; a closer analysis of the algorithm may yield further optimizations. As the compute capability of GPU and CPU continues to scale up and more and more applications converge into specialized computation, existing GPU LAP solvers would get faster and even more performance could be extracted by designing specialized hardware.

APPENDIX

The classical 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

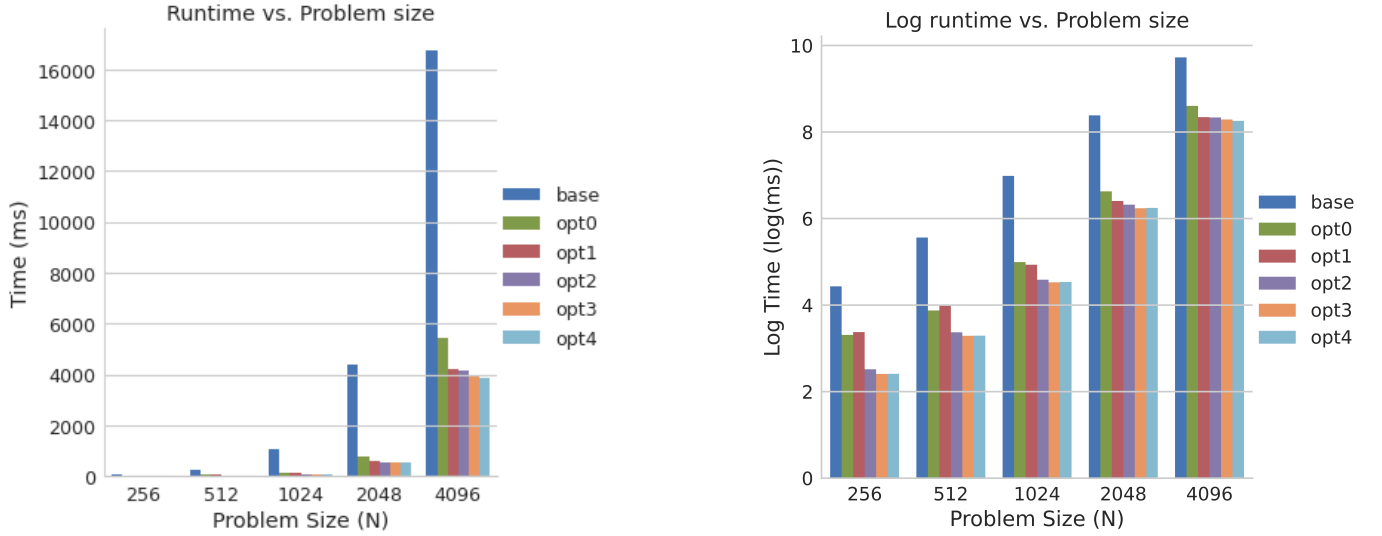


Fig. 6: This figure compares the run times of all our implemented optimizations; the left sub-figure indicates normal run times and the right sub-figure shows the log-scale run times, both with various problem sizes.

TABLE II: This table shows runtime and incremental time improvement for each optimization and respective problem size.

	Baseline	Opt0		Opt1		Opt2		Opt3		Opt4	
Problem Size	t (ms)	t (ms)	incr. Δt (ms)	t (ms)	incr. Δt (ms)	t (ms)	incr. Δt (ms)	t (ms)	incr. Δt (ms)	t (ms)	incr. Δt (ms)
256	84.29	27.38	56.90	29.29	-1.90	12.36	16.92	11.13	1.23	11.15	-0.019
512	260.24	48.21	212.02	54.24	-6.03	29.09	25.15	26.88	2.21	26.92	-0.04
1024	1082.22	147.70	934.52	138.95	8.74	98.32	40.62	92.41	5.91	93.01	-0.60
2048	4378.64	756.90	3621.74	605.97	150.93	556.54	49.42	512.14	44.39	516.76	-4.61
4096	16759.68	5454.24	11305.43	4203.91	1250.33	4171.05	32.86	3984.21	186.83	3857.83	126.38

TABLE III: This table shows incremental speedup and total speedups for each optimization and respective problem size. A value greater than 1 indicates some speedup was gained, while a value less than 1 indicates slowdown.

	Opt0		Opt1		Opt2		Opt3		Opt4	
Problem Size	incr. speedup	total speedup	incr. speedup	total speedup	incr. speedup	total speedup	incr. speedup	total speedup	incr. speedup	total speedup
256	3.08	3.08	0.94	2.88	2.37	6.82	1.11	7.57	1.00	7.56
512	5.40	5.40	0.89	4.80	1.86	8.94	1.08	9.68	1.00	9.67
1024	7.33	7.33	1.06	7.79	1.41	11.01	1.06	11.71	0.99	11.63
2048	5.78	5.78	1.25	7.23	1.09	7.87	1.09	8.55	0.99	8.47
4096	3.07	3.07	1.30	3.99	1.01	4.02	1.05	4.21	1.03	4.34

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.

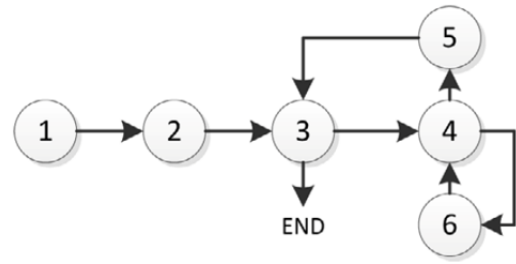


Fig. 7: Flow through steps of the Hungarian Algorithm.

REFERENCES

- [1] H.W. Kuhn, The Hungarian method for the assignment problem, *Nav. Res. Logist. Q.*, 2 (1-2) (1955), pp. 83-97.
- [2] J. Munkres, Algorithms for the assignment and transportation problems, *J. Soc. Ind. Appl. Math.*, 5 (1) (1957), pp. 32-38.
- [3] E.L. Lawler, *Combinatorial optimization: networks and matroids*, Courier Corporation (1976).
- [4] Date K. and Nagi R., GPU-accelerated Hungarian Algorithms for the linear assignment problem, *Parallel Comput.*, 57 (2016), pp. 52-72.
- [5] P. A. C. Lopes, S. S. Yadav, A. Ilic, and S. K. Patra, Fast block distributed CUDA implementation of the Hungarian algorithm, *J. Parallel and Distrib. Comput.*, 130 (2019), pp. 50-62.
- [6] R.E. Burkard and E. Çela, Linear assignment problems and extensions, D.-Z. Du, P. Pardalos (Eds.), *Handbook of Combinatorial Optimization*, Springer US (1999), pp. 75-149.
- [7] D.P. Bertsekas, The Auction algorithm for assignment and other network flow problems: a tutorial, *Interfaces*, 20 (4) (1990), pp. 133-149.
- [8] R. Jonker and A. Volgenant, A shortest augmenting path algorithm for dense and sparse linear assignment problems, *Computing*, 38 (4) (1987), pp. 325-340.
- [9] D.P. Bertsekas and D.A. Castañón, Parallel asynchronous Hungarian methods for the assignment problem, *ORSA J. Comput.*, 5 (3) (1993), pp. 261-274.
- [10] E. Balas, D. Miller, J. Pekny, and P. Toth, A parallel shortest augmenting path algorithm for the assignment problem, *J. ACM (JACM)*, 38 (4) (1991), pp. 985-1004.
- [11] D.P. Bertsekas and D.A. Castañón, Parallel synchronous and asynchronous implementations of the Auction algorithm, *Parallel Comput.*, 17 (6) (1991), pp. 707-732.
- [12] L. Buš and P. Tvrdík, Towards Auction algorithms for large dense assignment problems, *Comput. Optimization Appl.*, 43 (3) (2009), pp. 411-436.
- [13] J.L. Kennington and Wang Z., An empirical analysis of the dense assignment problem: Sequential and parallel implementations, *ORSA J. Comput.*, 3 (4) (1991), pp. 299-306.
- [14] C.N. Vasconcelos and B. Rosenhahn, Bipartite graph matching computation on GPU, *Energy Minimization Methods in Computer Vision and Pattern Recognition*, Springer (2009), pp. 42-55.
- [15] R. Roverso, A. Naiem, M. El-Beltagy, S. El-Ansary, and S. Haridi, A GPU-enabled solver for time-constrained linear sum assignment problems, *Informatics and Systems (INFOS)*, 2010 The 7th International Conference on, IEEE (2010), pp. 1-6.
- [16] Gurobi Optimizer Reference Manual, Gurobi Optimization LLC (2021), <https://www.gurobi.com>.