A Novel Faster APSP Algorithm for GPUs

Yelai Feng\textsuperscript{1,2} and Huaixi Wang\textsuperscript{2,*}

\textsuperscript{1}College of Computer Science and Technology, National University of Defense Technology, Changsha, 410000, China
\textsuperscript{2}College of Electronic Engineering, National University of Defense Technology, Hefei, 230000, China
\textsuperscript{*}wanghuaixi@nudt.edu.cn

ABSTRACT

The shortest path problem is a class of typical problems in graph theory and network science, it has a wide range of application scenarios. At present, the parallel single-source shortest path algorithm is mainly used to solve the all-pair shortest path problems. We propose a novel all-pair shortest path algorithm based on block matrix multiplication via GPUs. Our key advancement is transforming the shortest path problems into the linear algebra problems, taking advantage of the GPUs’ performance ascendancy in this regard. In the experiments, the novel algorithm achieves average of $41.257 \times$ and the maximum of $89.919 \times$ over Dijkstra algorithm which implements the priority queue by the binary heap and optimized via multi-threaded.

keywords: Graph Theory, Network Science, Parallel Computing, Shortest Path, Matrix Multiplication

1 Introduction

The shortest path problem is a basic problem of graph theory, and has attracted the attention of researchers from many disciplines such as transportation planning, computer science, network science, and applied mathematics\textsuperscript{1–5}. As the scale of the graph increases, it is difficult for serial algorithms to adapt to such changes. Researchers hope to tackle this type of problem through parallel computing, and top conferences in the field of parallel and distributed computing have focused on this problem\textsuperscript{6}.

The shortest path problem is generally divided into four categories: point-to-point shortest path problem, single-source shortest path problem, multi-source point shortest path problem, and all-pair shortest path problem. The classic algorithms to solve the shortest path problem are: Dijkstra algorithm\textsuperscript{7}, Bellman-Ford algorithm\textsuperscript{8} and Floyd algorithm\textsuperscript{9}. In the second half of the last century, artificial intelligence began its initial development and brought new ideas to the shortest path problem\textsuperscript{10–13}. The development of genetic algorithm provides a new solution for shortest path problem, and researchers continue to focus on the methods\textsuperscript{14–17}.

Parallel computing, which began to flourish at about the same time as artificial intelligence, provided more possibilities for researchers to solve general problems and became the basis of artificial intelligence. When single-core performance is no longer growing fast, co-processing through multiple cores seems like a perfect solution. Unfortunately, this perfect solution is not widely applicable. Floyd algorithm is based on dynamic programming technology, and it is difficult to accelerate the calculation through parallelism because of the pursuit of the optimal solution at each step. The current best method is to deal with the all-pair shortest path problem by a parallel single-source shortest path algorithm and this has received extensive attention at academic conferences in the field of parallel computing\textsuperscript{18–25}.

Our key contribution is to transform the shortest path problem into a linear algebra problem which can take full using of the huge advantages of GPUs. We propose a novel all-pair shortest path algorithm via aDjacency mAtrix operation Worked on the graphs Networks, which has been named DAWN.

DAWN is a more efficient algorithm based on adjacency matrix operations for solving the all-pair shortest paths problem, which requires $\Theta(n^2)$ space and $\Theta(dim \cdot n^{2.387})$ time, where $dim$ is the diameter of the graph. DAWN can accelerate computing via a multi-GPU system, and its time complexity depends on the number of nodes and is insensitive to graph density. This paper does not distinguish between graphs and networks which have the same meaning.

The main contributions of this work are as follows:

1. We proved mathematically that the shortest path problems can be transformed into the linear algebra problems, which is the theorem has been first systematically described.

2. We propose a novel algorithm based on the theorem which can take full using of the huge advantages of GPUs on the linear algebra problems, and avoid read-write conflicts which had been solved by double buffering, and limits the parallelism of the algorithm.
3. We optimized the novel algorithm by the matrix block multiplications, and found that the matrix can be guaranteed to be reblockable by continuously adding rows and columns with all elements zero, which expands the applicability of the algorithm.

4. We propose a method of path composition to cope with graphs which have a few of longer shortest paths.

In section 2, we introduce the problem of shortest path and its typical algorithms. In section 3, we describe the design of the DAWN algorithm and propose the optimization methods to make it more widely applicable to various graphs. In section 4, we demonstrate the efficiency of DAWN through multiple sets of comparative experiments. In section 5, we conclude the work of this paper and propose future directions.

2 Related Works

The shortest path problem is a classic problem in graph theory and network science. In this section, we introduce Dijkstra algorithm, Bellman-Ford algorithm, Floyd algorithm and their extensions, which are typical algorithms for solving the shortest path problem.

2.1 Dijkstra algorithm

In 1959, Edsger W. Dijkstra et al. proposed and solved two graph problems: construct the tree of minimum total length between the n nodes, and find the path of minimum total length between two given nodes P and Q. There are three main dijkstra optimization methods, priority queue optimization, binary heap optimization and Fibonacci heap optimization. The algorithms improve the priority queue data structure based on Dijkstra algorithm.

Ulrich Meyer et al. proposed an optimized dijkstra algorithm which can be parallel setting for a large class of graphs. Best parallel version of Δ-stepping Dijkstra algorithm takes $\Theta(d \cdot L \cdot \log n + \log^2 n)$ time and $\Theta(n + m + d \cdot L \cdot \log n)$ work on average, where $L$ denotes the maximum shortest path weight from the source node $s$ to any node reachable from $s$, and $d$ represents maximum node degree.

Andrew Davidson et al. proposed an adaptation of Δ-stepping Dijkstra algorithm that makes several simplifications to the worklist which used two buckets(named Near and Far) and double buffering. The near and far buckets implements an approximate priority queue, the barrier synchronization and double buffering technology limit the parallelism of the algorithm. A single core in a GPU is not as powerful as in a cpu, but rather mediocre.

In one of the latest work, researchers demonstrate that sophisticated data structures can be used on these mediocre cores. This is a remarkable research result that defies common sense. Kai Wang et al. proposed a Δ-stepping Dijkstra algorithm based on the GPU, using a phisticated work scheduler to improve work efficiency and parallelism, which named ADDS. They demonstrated that a balance of algorithmic efficiency and GPU fitness can be achieved by an efficient scheduler and showed that sophisticated data structures can be used on GPU via careful design.

2.2 Bellman-Ford algorithm

In 1958, Bellman et al. proposed a new shortest path algorithm, which is suitable for sparse graphs and can handle negative weighted graphs. The Bellman-Ford algorithm needs to perform $n - 1$ relaxation operations on $m$ edges respectively, and requires $\Theta(nm)$ time. Compared with the Dijkstra algorithm, the Bellman-Ford algorithm is more suitable for parallelism, and researchers have been doing work on the parallel Bellman-Ford algorithm.

2.3 Floyd algorithm

In 1962, Robert W. Floyd proposed an algorithm which used uses dynamic programming to solve the shortest path problem and is easy to implement. Compared to Dijkstra algorithm, which uses sophisticated data structures to optimize, Floyd algorithm uses only two-dimensional arrays. However, the pursuit of the optimal solution at each step limits the parallelism of the Floyd algorithm. Researchers have also been working on the optimization of the folyd algorithm and the development of a parallel version. The latest representative work on the Floyd algorithm was published in PPoPP’ 20, Piyush Sao et al. proposed a Floyd algorithm optimized by techniques of fill-in reducing ordering, symbolic analysis, supernodal traversal and elimination tree parallelism.

3 Methods

In this section, we will introduce the design of the DAWN algorithm, which is mainly divided into four aspects: mathematical foundation, implementation on the unweighted graphs, version on the weighted graphs and optimization methods.
3.1 mathematical foundation

We represent the graph $G = (V, E)$ as an adjacency matrix:

$$A = \begin{bmatrix}
    a_{1,1} & a_{1,2} & \cdots & a_{1,n-1} & a_{1,n} \\
    a_{2,1} & a_{2,2} & \cdots & a_{2,n-1} & a_{2,n} \\
    \vdots & \vdots & \ddots & \vdots & \vdots \\
    a_{n-1,1} & a_{n-1,2} & \cdots & a_{n-1,n-1} & a_{n-1,n} \\
    a_{n,1} & a_{n,2} & \cdots & a_{n,n-1} & a_{n,n}
\end{bmatrix}.$$ (1)

**Theorem 1** In the matrices $A^k$, the $a_{i,j}^{(k)}$ is the number of paths in $G$ of length $k$ from $v_i$ to $v_j$.

Theorem 1 gives us ideas for designing shortest path algorithms, which reveal the relationship between paths and matrix multiplication. In the past representative work, the methods are based on path search and dynamic programming, which have been detailed in Section 2. To design the shortest path algorithm through the relationship between paths and matrix multiplication, we need to solve three main problems:

1. how to identify the shortest paths;
2. how to operate on the weighted or directed graphs;
3. how to reduce the time complexity of the algorithm.

In the following, we will illustrate the methods in three subsections.

3.1.1 Identify the Shortest Paths

**Theorem 2** In the graphs, arbitrary shortest path of length $k$ can be expressed as the connection of two shortest paths with length $k_1$ and $k_2$, where $k = k_1 + k_2$.

It’s an obvious theory, and we propose its inferences:

**Corollary 1** The first occurrence of the paths from $v_i$ to $v_j$ are the shortest path, which can be described as $a_{i,j}^{(k)} \neq 0 \land i \neq j \land \sum_{1\leq p \leq k-1} a_{i,j}^{(p)} = 0$.

**Corollary 2** Arbitrary shortest path of length $k$ contains one or more shortest path of length $k-1$.

Corollary 1 states the sufficient condition for the shortest paths, and reveals the relationship between shortest paths and matrix multiplication. Thus, we transform the shortest path problem into the linear algebra problem, which is the first to be rigorously proposed and implemented.

There are two conditions for breaking the loop and ending,

1. DAWN already found the shortest paths between all pairs of nodes in the graph;
2. the filling rate of the amount matrix does not change when a loop ends, which means no new paths were found in the loop.

Corollary 2 proves the stopping condition 2 of the DAWN. If no new shortest path is found in $A^n$, then no new paths will be found in $A^{n+1}$, and ending the DAWN.

3.1.2 Version on the Weighted Graphs

There is no difference in the operation of DAWN on directed and undirected graphs, so this section mainly discusses the version applicable to weighted graphs. We represent the graph $G = (V, E, W)$ as an adjacency matrix:

$$A_w = \begin{bmatrix}
    w_{1,1} & w_{1,2} & \cdots & w_{1,n-1} & w_{1,n} \\
    w_{2,1} & w_{2,2} & \cdots & w_{2,n-1} & w_{2,n} \\
    \vdots & \vdots & \ddots & \vdots & \vdots \\
    w_{n-1,1} & w_{n-1,2} & \cdots & w_{n-1,n-1} & w_{n-1,n} \\
    w_{n,1} & w_{n,2} & \cdots & w_{n,n-1} & w_{n,n}
\end{bmatrix}.$$ (2)

**Theorem 3** In the matrices $A_w^{(k_w)}$, the $w_{i,j}^{(k_w)}$ is the number of paths in $G = (V, E, W)$ of weighted $k_w$ from $v_i$ to $v_j$.

Theorem 3 is an extension of Theorem 1 on the weighted graphs. We notice a phenomenon that $A_w^{(k_w)}$ would not include edges and paths with weights greater than 1. Therefore, we obtain a corollary based on Theorem 3 and assume that $k_w \in \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers.
Corollary 3 Matrix $A_w^{(k_w)}$ would not include edges and paths with weights greater than $k_w$.

Via the Corollary 3, we need to process the matrix $A_w$ which contains edges with the weighted greater than 1 to obtain the matrix $A_w^{1}$. We define the matrix $H$ that contains edges with weighted greater than $k_w$, and these edges are gradually added to the matrix in the iterative computation. So, we obtain the iterative equations on the weighted graphs:

$$A_w^{(k_w)} = \sum_{1 \leq i \leq n} \sum_{1 \leq j \leq n} a_{i,j}^{(k_w)} + h_{i,j}^{(k_w)},$$  \hspace{1cm} (3)

where $h_{i,j}^{(k_w)}$ represents the element in matrix $H$ which are the amount of paths, and $k$ represents the weight of the edges. In the next section, we would discuss that if there are some paths and edges with weight greater than $n$, which is not considered in unweighted graphs.

3.1.3 Time Complexity of the DAWN

In this section, we will discuss the time complexity of the DAWN in detail. First, the time complexity of the unoptimized DAWN algorithm is as follows:

$$T(n) = \Theta(dim \cdot n^3),$$  \hspace{1cm} (4)

where $dim$ represents the diameter of the graph. The diameter of the graph can be up to $n - 1$ on the unweighted graphs, which will cause the DAWN time complexity increasing to $n^3$. But there have been many well-established studies showing that the diameter of a graph/network is $\mu \cdot \log n$\(^{36-43}\), where $\mu$ is a constant parameter.

Volker Strassen proposed the matrix block multiplication algorithm required $\Theta(n \log^7)$ time\(^{44}\), which used 7 multiplications and 18 additions. In 1987, Coppersmith and Winograd proposed a method accelerating matrix multiplication, which required $\Theta(n^{2.376})$ time\(^{45}\). Virginia Vassilevska Williams has contributed the latest representative work on the matrix block multiplication algorithm\(^{46}\), which running in the time of $\Theta(n^{2.3727})$. We use Coppersmith-Winograd algorithm to reduce the time complexity of the algorithm and speed up the computation. So we get the time complexity of the DAWN:

$$T(n) = \Theta(n^{2.376} \cdot \log n),$$  \hspace{1cm} (5)

and we will introduce targeted optimization methods in Section 3.4, which reduce the time complexity of the DAWN.

3.2 DAWN on the Unweighted Graphs

We define the algorithm on the unweighted graphs as:

$$A_{ij} = \sum_{1 \leq i \leq n} \sum_{1 \leq j \leq n} a_{i,j}^{(k)} ,$$  \hspace{1cm} (6)

and formula holds under the condition $a_{ij}^{(k)} \neq 0 \wedge i \neq j \wedge \sum_{1 \leq p \leq k-1} a_{ij}^{(p)} = 0$, which guaranteed to filter out the shortest paths.

And calculation result can be expressed as:

$$\begin{align*}
amount[i][n] &= \sum_{1 \leq i \leq n} \sum_{1 \leq j \leq n} a_{i,j}^{(k)}, \\
length[i][n] &= \sum_{1 \leq i \leq n} \sum_{1 \leq j \leq n} k_{i,j}.
\end{align*}$$  \hspace{1cm} (7)

We use pseudocode in the appendix to demonstrate, and $amount_{\text{fill}}$ represents the number of filling in the matrix $amount$. We use $\text{tmp}$ to store the value of $amount_{\text{fill}}$ at the end of the previous loop.

3.3 DAWN on the Weighted Graphs

We describe the relationship between the adjacency matrix $A_w$ and matrix $H$ in detail:

$$A_w = \sum_{k=1}^{w_0} H^{(k)},$$  \hspace{1cm} (9)

where $w_0$ represents the maximum weight of the edge in the graph. First, we define the left and right multiplication of the matrix respectively:

$$A_1^{(k)} = A^{(k-1)} \times H^{(1)} + H^{(k)},$$  \hspace{1cm} (10)

$$A_2^{(k)} = H^{(1)} \times A^{(k-1)} + H^{(k)},$$  \hspace{1cm} (11)
and the matrix $H^{(k)}$ will not affect the matrix relationship in the undirected graph and can be ignored. We transpose the matrix $A^{(k)}_2$:

$$[A^{(k)}_2]^T = [H^{(1)} \times A^{(k-1)}]^T = [A^{(k-1)}]^T \times [H^{(1)}]^T = A^{(k-1)} \times H^{(1)} = A^{(k)}_1,$$

and the equation means that we don’t need to compute the matrix multiplication twice, just transpose the matrix which can be obtained the result of matrix left and right multiplication.

Obviously, paths can be combined by shorter paths. We provide an equation:

$$x_1 + 2x_2 + 3x_3 \cdots + (w_i - 1)x_{w_i-1} + w_i x_{w_i} = w_i,$$

and $w_i$ represents the maximum weight of the path to be solved in this time. This is a multivariate linear equation with many solutions, and each solution is a combination of paths. We calculate all path combinations via:

$$A_w^{(k)} = \sum_{i+j=k,1 \leq i,j \leq k} \left[ A_w^{(i)} \times H^{(j)} + A_w^{(j)} \times H^{(i)} \right],$$

and it requires a huge computational cost.

In the DAWN, we adopt a simpler solution that directly takes the maximum value of $A^{(k)}_1$ and $A^{(k)}_2$ positions, which includes most paths. This approximation methods requires $\text{dim}$ multiplications on undirected networks and $2 \cdot \text{dim}$ multiplications on directed networks, which would not cause a significant increase in time complexity of the DAWN.

There is a close relationship between the accuracy of the approximation method and the characteristics of the graph. Approximate method is suitable for the graphs, which mainly contains edges with smaller weights and a few edges with larger weights, and has good accuracy on these graphs. In the graph, if arbitrary path with weight $k$ can be obtained by combining the path with weight $k - 1$ and an edge of weight 1, no matter how the weighted edges are distributed, the DAWN can accurately calculate the result without losing any paths.

We provide the weight transfer function to accommodate graphs which the weight distribution does not meet the requirements:

$$w' = 1 + \log w,$$

where $w'$ represents the transformed weight and it needs to be rounded according to the rounding method. We give an example to facilitate the understanding of DAWN on the weighted graphs which provided in the appendix.

### 3.4 Optimization Methods

In 1999, Albert-László Barabási et al. found scale-free properties in real networks\(^{39,47}\), and degree distribution obeys a power-law distribution. We found that there are some graphs that obey scale-free properties with very long diameters, and show one of real network used in the study which is a collaboration network\(^{38}\). The method of path combination can be used to reduce the number of matrix multiplication calculations on the graph.

The time complexity of the path combination method is $\Theta(n)$, and $x, y$ represents the pair of nodes which the shortest path has not yet occurred in the previous calculation. When the number of non-occurring node pairs is reduced to $\mu \cdot n \log n$, this method can be used instead of matrix multiplication which reduces time complexity and is parallelizable. A necessary condition for parallelism is that most pairs of nodes in the graph have already found the shortest path. When the scale of the graph is relatively small, the threshold of the path combination method can be limited to $\mu n$.

Optimized via path combination method, the time complexity of DAWN is reduced from $\Theta(n^{2.376} \cdot \text{dim})$ to $\Theta(n^{2.376} \cdot (\text{dim} - \eta) + \mu n^2 \cdot \log n)$, where $\eta$ represents the reduced times of matrix multiplications. In the collaboration network, the path combination method reduces matrix multiplications to 9, which originally required 16 multiplications. We show the pseudocode in the appendix.

The matrix block algorithm is only suitable for even-order matrices, and if the matrix cannot be continuously divided into blocks, it will greatly limit the performance of the algorithm. The graphs which has $\mu \cdot 2^n$ nodes is very rare, and we propose a new theory to expand the scope of application of the algorithm.

**Theorem 4** In the graphs, adding the orphaned nodes would not affect the shortest paths in them.

**Theorem 5** For a matrix that qualifies for partitioning, the results of multiplication and block multiplication are consistent. Obviously, we get the corollaries:

**Corollary 4** If the order of the adjacency matrix is odd, adding a new row and column to the matrix in which all elements are set to 0 would not affect the shortest paths in them.

**Corollary 5** If the order of the adjacency matrix is odd, adding a new row and column to the matrix in which all elements are set to 0 would not affect the result of matrix block multiplications.

This operation, which requires at most $\log n$ time, does not affect the result of matrix block multiplication and extends the applicability of the DAWN. The temporal complexity of the algorithm slightly increases to $\Theta((n + \log n)^\alpha)$.
4 Results

In this section, the real networks comes from the Stanford University public dataset\cite{49}, the Barabási–Albert models and the Erdős–Rényi models graphs are generated using NetworkX. NetworkX is a Python package for complex networks analysis; it has excellent performance and a wide range of application scenarios\cite{50}. Paul Erdos et al. proposed the concept of random graph in 1960\cite{51}.

We noticed that the maximum of matrices order in multiplications that TASLE V100 can support is about 45000 when DAWN is running, and this upper limit is slightly floating. When the scale of the network is above the upper limit, continuously partitioning the matrix becomes an inevitable choice. Table 1 shows some of the parameters of the test machine.

Table 1. Partial Parameter Table of the Test Machine

| Hardware | Parameters          |
|----------|---------------------|
| CPU      | Intel Xeon Gold 6151|
| RAM      | 512GB               |
| GPU      | NVIDIA TESLA V100   |
| IDE      | Visual Studio 2019  |
| OS       | Windows Server 2019 |
| Toolkit  | CUDA 11.4           |

We show the running time of DAWN on Erdős–Rényi graphs and Barabasi-Albert graphs with 10000 nodes which are the random graphs generated using NetworkX. The unit for the running time is seconds, with three decimals reserved. The speedup shown in the table is the average value of 5 tests. We compared the naive dijkstra(Dijkstra), the version optimized by sequential priority queue(DOQ), the version optimized via binary heap to implement priority queue(DOBH), DAWN and the parallel versions based on OpenMP of the above algorithms.

Katzav et al. proposed the phenomenon of network diameter shrinkage\cite{52}. As the density increases, the diameter of the network shrinks and the calculation time of DAWN on the networks decreases. In these tests, the speedup ratio can reach up to 709. Compared with the dijkstra algorithm optimized by the binary heap and its version of running in parallel, DAWN can achieve a speedup of $352.479 \times$ and $19.272 \times$, respectively.

We tested the performance of on the some real networks: General Relativity and Quantum Cosmology collaboration network\cite{48}, Autonomous systems - Oregon-1\cite{53}, Enron email network\cite{54}, High-energy physics citation network\cite{55}, Brightkite\cite{56},
Figure 2. Experiment on the Barabasi-Albert graphs with 10000 nodes and a connection probability of 0.05. We take the DOBH with parallel accelerated as the baseline and the vertical axis is the logarithm of the speedup.

Figure 3. Experiment on the Erdős–Rényi graphs with 10000 nodes and a connection probability of 0.5. We take the naive dijkstra(Dijkstra) as the baseline and the vertical axis is the speedup.

Graph Embedding with Self Clustering: Facebook-Artist. We use tables to show the detailed parameters of these networks.

| Index | Name   | Nodes | Edges  | Diameter | Nodes in largest SCC |
|-------|--------|-------|--------|----------|----------------------|
| 1     | GRQC   | 5242  | 14496  | 17       | 0.793                |
| 2     | ASO1   | 10670 | 22002  | 9        | 1.000                |
| 3     | EE     | 36692 | 183831 | 11       | 0.918                |
| 4     | HEPCN  | 34546 | 421578 | 12       | 0.368                |
| 5     | Brightkite | 58228 | 214078 | 16       | 0.974                |
| 6     | GESCFA | 50515 | 819306 | 11       | 1.000                |

On the graph with tens of thousands nodes, the naive Dijkstra algorithm and the serial version of the algorithm are extremely computationally expensive. We replace the baseline algorithm with a parallel version of the binary heap optimized Dijkstra algorithm. On the graph generated by computing, the diameter of the graph is strongly correlated with the number of nodes and edges in the graph. On the real networks, the diameter is related to the characteristics of itself, and even on a small-scale network, the diameter may be large. GRQC is one such example, and DAWN does not perform as well as we would expect. The results prove that on sparse and small-scale graphs, the transmission overhead greatly limits the performance of DAWN.

The problem of long data transmission time is a problem that has not been solved, and we did not consider at the beginning of the DAWN design. We use the cudastream to asynchronously transfer data to the GPU, and hoping to reduce the waiting time. In the experiment, the average waiting time is still more than 40% of the operation time. We have to admit that in addition to the graph diameter, the preformance of the test machine' memory also affects the operation time of DAWN.

Wang Kai et al. mentioned that many GPU SSSP algorithms use double buffering, which reduces the concurrency of the algorithm. On a experimental graph, the average work count per iteration is only 800, while a RTX 2080 GPU has 68K hardware threads. DAWN does not have the problem of read-write conflicts, and the GPU can call almost threads to calculate matrix multiplication.

Compared with the parallel binary heap optimization dijkstra algorithm, the DAWN achieves the speedup of 1.038×, 5.749×, 39.031×, 39.184×, 72.619×, and 89.919× on the real networks shown by Fig.4-9, respectively.
Figure 4. GRQC is from the e-print arXiv and covers scientific collaborations between authors papers submitted to General Relativity and Quantum Cosmology category, has 5242 nodes and 14496 edges. We take the DOBH with parallel accelerated as the baseline and the vertical axis is the logarithm of the speedup.

Figure 5. ASO1 is a graph of the Autonomous Systems (AS) peering information inferred from Oregon route-views, has 10670 nodes and 22002 edges. We take the DOBH with parallel accelerated as the baseline and the vertical axis is the logarithm of the speedup.

Figure 6. Enron email communication network posted to the web by the Federal Energy Regulatory Commission during its investigation, which consists of 36692 nodes and 183831 edges. We take the DOBH with parallel accelerated as the baseline and the vertical axis is the speedup.

Figure 7. HEPCN is the high energy physics phenomenology citation graph from the e-print arXiv and covers all the citations within a dataset of 34546 nodes with 421578 edges. We take the DOBH with parallel accelerated as the baseline and the vertical axis is the speedup.
5 Conclusion

In this paper, we proposed a novel all-pair shortest path algorithm based on the matrix operations, which has been named DAWN and requires $\Theta(n^2)$ space and $\Theta(dim \cdot n^{2.387})$ time. We transform the shortest path problem into a linear algebra problem which can take full using of the huge advantages of GPUs. This transformation method is first systematically described, and we developed the feasible solution based on mathematical principles. Sufficient independence between data allows DAWN to accelerate computing via the multi-GPU systems. DAWN is sensitive to graph diameter, whose time complexity depends on the number of nodes. Compared with the dijkstra algorithm optimized by the binary heap with 32 threads in parallel computing, the DAWN achieves the average of 41.257×, and the maximum of 89.919×.

References

1. Dreyfus, S. E. An appraisal of some shortest-path algorithms. *Oper. research* 17, 395–412 (1969).
2. Bertsekas, D. *Network optimization: continuous and discrete models*, vol. 8 (Athena Scientific, 1998).
3. Tarjan, R. E. *Data structures and network algorithms* (SIAM, 1983).
4. Waissi, G. R. Network flows: Theory, algorithms, and applications (1994).
5. Bertsekas, D. P. & Tsitsiklis, J. N. An analysis of stochastic shortest path problems. *Math. Oper. Res.* 16, 580–595 (1991).
6. Wang, K., Fussell, D. & Lin, C. A fast work-efficient sssp algorithm for gpus. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 133–146 (2021).
7. Dijkstra, E. W. *et al.* A note on two problems in connexion with graphs. *Numer. mathematik* 1, 269–271 (1959).
8. Bellman, R. On a routing problem. *Q. applied mathematics* 16, 87–90 (1958).
9. Floyd, R. W. Algorithm 97: shortest path. *Commun. ACM* 5, 345 (1962).
10. Hart, P. E., Nilsson, N. J. & Raphael, B. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Syst. Sci. Cybern.* 4, 100–107 (1968).
11. Chang, C. & Lee, R. Problem-solving methods in artificial intelligence (nils j. nilsson) (1972).
12. Newell, A., Simon, H. A. *et al.* *Human problem solving*, vol. 104 (Prentice-hall Englewood Cliffs, NJ, 1972).
13. Heuristics, P. Intelligent search strategies for computer problem solving (1984).
14. Gen, M., Cheng, R. & Wang, D. Genetic algorithms for solving shortest path problems. In *Proceedings of 1997 IEEE International Conference on Evolutionary Computation (ICEC’97)*, 401–406 (IEEE, 1997).
15. Davies, C. & Lingras, P. Genetic algorithms for rerouting shortest paths in dynamic and stochastic networks. *Eur. J. Oper. Res.* **144**, 27–38 (2003).
16. Sharma, Y., Saini, S. C. & Bhandhari, M. Comparison of dijkstra’s shortest path algorithm with genetic algorithm for static and dynamic routing network. *Int. J. Electron. Comput. Sci. Eng.* **1**, 416–425 (2012).
17. Lin, L., Wu, C. & Ma, L. A genetic algorithm for the fuzzy shortest path problem in a fuzzy network. *Complex & Intell. Syst.* **7**, 225–234 (2021).
18. Crauser, A., Mehlhorn, K., Meyer, U. & Sanders, P. A parallelization of dijkstra’s shortest path algorithm. In *International Symposium on Mathematical Foundations of Computer Science*, 722–731 (Springer, 1998).
19. Martín, P. J., Torres, R. & Gavilanes, A. Cuda solutions for the sssp problem. In *International Conference on Computational Science*, 904–913 (Springer, 2009).
20. Kumar, S., Misra, A. & Tomar, R. S. A modified parallel approach to single source shortest path problem for massively dense graphs using cuda. In *2011 2nd International Conference on Computer and Communication Technology (ICCCCT-2011)*, 635–639 (IEEE, 2011).
21. Ortega-Arranz, H., Torres, Y., Llanos, D. R. & Gonzalez-Escribano, A. A new gpu-based approach to the shortest path problem. In *2013 International Conference on High Performance Computing & Simulation (HPCS)*, 505–511 (IEEE, 2013).
22. Davidson, A., Baxter, S., Garland, M. & Owens, J. D. Work-efficient parallel gpu methods for single-source shortest paths. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 349–359 (IEEE, 2014).
23. Busato, F. & Bombieri, N. An efficient implementation of the bellman-ford algorithm for kepler gpu architectures. *IEEE Transactions on Parallel Distributed Syst.* **27**, 2222–2233 (2015).
24. Surve, G. G. & Shah, M. A. Parallel implementation of bellman-ford algorithm using cuda architecture. In *2017 International conference of Electronics, Communication and Aerospace Technology (ICECA)*, vol. 2, 16–22 (IEEE, 2017).
25. Wang, H. *et al.* Sep-graph: finding shortest execution paths for graph processing under a hybrid framework on gpu. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, 38–52 (2019).
26. Johnson, D. B. Efficient algorithms for shortest paths in sparse networks. *J. ACM (JACM)* **24**, 1–13 (1977).
27. Cherkassky, B. V., Goldberg, A. V. & Radzik, T. Shortest paths algorithms: Theory and experimental evaluation. *Math. programming* **73**, 129–174 (1996).
28. Raman, R. Priority queues: Small, monotone and trans-dichotomous. In *European Symposium on Algorithms*, 121–137 (Springer, 1996).
29. Raman, R. Recent results on the single-source shortest paths problem. *ACM SIGACT News* **28**, 81–87 (1997).
30. Thorup, M. Undirected single-source shortest paths with positive integer weights in linear time. *J. ACM (JACM)* **46**, 362–394 (1999).
31. Thorup, M. On rand priority queues. *SIAM J. on Comput.* **30**, 86–109 (2000).
32. Meyer, U. & Sanders, P. δ-stepping: a parallelizable shortest path algorithm. *J. Algorithms** **49**, 114–152 (2003).
33. Wei, D. An optimized floyd algorithm for the shortest path problem. *J. Networks* **5**, 1496–1504 (2010).
34. Pradhan, A. & Mahinthakumar, G. Finding all-pairs shortest path for a large-scale transportation network using parallel floyd-warshall and parallel dijkstra algorithms. *J. Comput. Civ. Eng.* **27**, 263–273 (2013).
35. Yuan, H., Hu, J., Song, Y., Li, Y. & Du, J. A new exact algorithm for the shortest path problem: An optimized shortest distance matrix. *Comput. Ind. Eng.* **158**, DOI: 10.1016/j.cie.2021.107407 (2021).
36. Sao, P., Kannan, R., Gera, P. & Vuduc, R. A supernodal all-pairs shortest path algorithm. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 250–261 (2020).
37. Harary, F., Norman, R. Z. & Cartwright, D. *Structural models: An introduction to the theory of directed graphs* (Wiley, 1965).
38. Bollobás, B. The diameter of random graphs. *Transactions Am. Math. Soc.* **267**, 41–52 (1981).
39. Albert, R., Jeong, H. & Barabási, A.-L. Diameter of the world-wide web. *nature* **401**, 130–131 (1999).
40. Cohen, R. & Havlin, S. Scale-free networks are ultrasmall. *Phys. Rev. Lett.* **90**, 058701, DOI: 10.1103/PhysRevLett.90.058701 (2003).
41. Riordan, O. et al. The diameter of a scale-free random graph. *Combinatorica* **24**, 5–34 (2004).
42. Ganesh, A. & Xue, F. On the connectivity and diameter of small-world networks. *Adv. Appl. Probab.* **39**, 853–863, DOI: 10.1239/aap/1198177228 (2007).
43. Takes, F. W. & Kosters, W. A. Determining the diameter of small world networks. CIKM ’11, 1191–1196, DOI: 10.1145/2063576.2063748 (Association for Computing Machinery, New York, NY, USA, 2011).
44. Strassen, V. et al. Gaussian elimination is not optimal. *Numer. mathematik* **13**, 354–356 (1969).
45. Coppersmith, D. & Winograd, S. Matrix multiplication via arithmetic progressions. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC ’87, 1–6, DOI: 10.1145/28395.28396 (Association for Computing Machinery, New York, NY, USA, 1987).
46. Williams, V. V. Multiplying matrices faster than coppersmith-winograd. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, 887–898 (2012).
47. Barabási, A.-L. & Albert, R. Emergence of scaling in random networks. *science* **286**, 509–512 (1999).
48. Leskovec, J., Kleinberg, J. & Faloutsos, C. Graph evolution: Densification and shrinking diameters. *ACM transactions on Knowl. Discov. from Data (TKDD)* **1**, 2–11 (2007).
49. Leskovec, J. & Krevl, A. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data (2014).
50. Hagberg, A., Swart, P. & S Chult, D. Exploring network structure, dynamics, and function using networkx. Tech. Rep., Los Alamos National Lab.(LANL), Los Alamos, NM (United States) (2008).
51. Erdos, P., Rényi, A. et al. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci* **5**, 17–60 (1960).
52. Katzav, E. et al. Analytical results for the distribution of shortest path lengths in random networks. *EPL (Europhysics Lett.)* **111**, 26006 (2015).
53. Leskovec, J., Kleinberg, J. & Faloutsos, C. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, 177–187 (2005).
54. Leskovec, J., Lang, K. J., Dasgupta, A. & Mahoney, M. W. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Math.* **6**, 29–123 (2009).
55. Gehrke, J., Ginsparg, P. & Kleinberg, J. Overview of the 2003 kdd cup. *Acm Sigkdd Explor. Newsl.* **5**, 149–151 (2003).
56. Cho, E., Myers, S. A. & Leskovec, J. Friendship and mobility: Friendship and mobility: User movement in location-based social networksfriendship and mobility. In *User Movement in Location-Based Social Networks ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)* (2011).
57. Rozemberczki, B., Davies, R., Sarkar, R. & Sutton, C. Gemsec: Graph embedding with self clustering. In *Proceedings of the 2019 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2019*, 65–72 (ACM, 2019).
Appendix

Example

Figure 10. Example figure of directed and weighted graph $G_8 = (8, 10, w)$

There is a directed and weighted graph $G_8 = (8, 10, w)$:

$$G = \begin{bmatrix}
0 & 1 & 0 & 2 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 4 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 2 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix},$$

and the example figure is shown in the appendix. The weighted edges start to affect the matrix product in $G^{(3)}$:

$$G_1^{(3)} = \begin{bmatrix}
0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}, \quad G_2^{(3)} = \begin{bmatrix}
0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

In this example, the missing term of the approximation method is exactly 0, so it shows good accuracy. Let’s discuss this issue more generally. The definitions of matrices $A_1$ and $A_2$ can be found in Section 3.3 of the paper.
We get these path combination, \((H^{(1)})^4, H^{(2)} \times (H^{(1)})^2, H^{(3)} \times H^{(1)}, H^{(1)} \times H^{(2)} \times H^{(1)}, H^{(1)} \times H^{(3)}, H^{(4)}\), and \((H^{(2)})^2\) is not included.

Obviously, the DAWN algorithm can only consider one of weighted edge in a path at most, such as \(3 + 1\), and would lose the path like \(2 + 2\). In the graph, if arbitrary path with weight \(k\) can be obtained by combining the path with weight \(k - 1\) and an edge of weight 1, no matter how the weighted edges are distributed, the DANW can accurately calculate the result without losing any paths.

There is a close relationship between the accuracy of the approximation method and the characteristics of the graph. Approximate method is suitable for the graphs, which mainly contains edges with smaller weights and a few edges with larger weights, and has good accuracy on these graphs.
Algorithm 1 Path Combination

**Input:** length[n][n], {x, y}

**Output:** length[n][n]

1: function MAIN(length[n][n])
2:     for i = 0 → n - 1 do
3:         if length[x][i] ≠ -1 ∧ length[i][y] ≠ -1 then
4:             tmp ← length[x][i] + length[i][y]
5:         if length[x][y] = -1 ∨ length[x][i] > tmp then
6:             length[x][y] ← tmp
7:     end if
8:     end if
9:     end for
10:    return length[n][n]
11: end function
Algorithm 2 Unweighted Graphs

Input: $A[n][n], B[n][n], amount[n][n], length[n][n]$

Output: $amount[n][n], length[n][n]$

1: function MAIN($A[n][n]$)
2: for $i = 0 \rightarrow n - 1$, $j = 0 \rightarrow n - 1$ do
3:   if $A[i][j] = 1$ then
4:     $B[i][j] \leftarrow 1$
5:     $amount[i][j] \leftarrow 1$
6:     $length[i][j] \leftarrow 1$
7:   else
8:     $B[i][j] \leftarrow 0$
9:     $amount[i][j] \leftarrow -1$
10:    $length[i][j] \leftarrow -1$
11:  end if
12:  $amount_{fill} \leftarrow 0$
13:  $tmp \leftarrow 0$
14: end for
15: for $k = 2 \rightarrow \text{diameter}$, $i = 0 \rightarrow n - 1$, $j = 0 \rightarrow n - 1$ do
16:   $B[n][n] \leftarrow B[n][n] \times A[n][n]$
17:   if $B[i][j] \neq 0 \wedge length[i][j] = -1$ then
18:     $amount[i][j] \leftarrow B[i][j]$
19:     $length[i][j] \leftarrow k$
20:     if $amount_{fill} > n \times (n - 1) - 1$ then
21:       break
22:     end if
23:   end if
24:   if $tmp = amount_{fill}$ then
25:     break
26:   end if
27:   $tmp \leftarrow amount_{fill}$
28: end for
29: return $amount[n][n], length[n][n]$
30: end function
Algorithm 3 Weighted Graphs

Input: $A[n][n], B[n][n], \text{amount}[n][n], \text{length}[n][n]$  
Output: $\text{amount}[n][n], \text{length}[n][n]$

1: function MAIN($A[n][n]$)
2:   for $i = 0 \rightarrow n - 1, j = 0 \rightarrow n - 1$ do
3:     if $A[i][j] = 1$ then
4:       $B[i][j] \leftarrow 1$
5:       $\text{amount}[i][j] \leftarrow 1$
6:       $\text{length}[i][j] \leftarrow 1$
7:     else
8:       $B[i][j] \leftarrow 0$
9:       $\text{amount}[i][j] \leftarrow -1$
10:      $\text{length}[i][j] \leftarrow -1$
11:      map($\{i,j\}$) $\leftarrow A[i][j]$
12:     end if
13:     $\text{amount}_\text{fill} \leftarrow 0$
14:     $\text{tmp} \leftarrow 0$
15:     $C[n][n] \leftarrow 0$
16:   end for
17:   for $k = 2 \rightarrow \text{diameter}, i = 0 \rightarrow n - 1, j = 0 \rightarrow n - 1$ do
18:     if Graphs is directed then
19:       $C[n][n] \leftarrow A[n][n] \times B[n][n]$
20:       $B[n][n] \leftarrow B[n][n] \times A[n][n]$
21:       $B[i][j] \leftarrow \max(B[i][j], C[i][j])$
22:     else
23:       $B[i][j] = \max(B[i][j], B[j][i])$
24:     end if
25:     if $\text{map}(\{i,j\}) = k$ then
26:       $B[i][j] \leftarrow B[i][j] + 1$
27:     end if
28:     if $B[i][j] \neq 0 \land \text{length}[i][j] = -1$ then
29:       $\text{amount}[i][j] \leftarrow B[i][j]$
30:       $\text{length}[i][j] \leftarrow k$
31:       $k \leftarrow k + 1$
32:       if $\text{amount}_\text{fill} > n \times (n - 1) - 1$ then
33:         break
34:     end if
35:   end if
36:   if $\text{tmp} = \text{amount}_\text{fill}$ then
37:     break
38:   end if
39:   $\text{tmp} \leftarrow \text{amount}_\text{fill}$
40: end for
41: return $\text{amount}[n][n], \text{length}[n][n]$
42: end function