# Parallel Programming Assignment 2
# Parallel Delta Stepping Algorithm Implementation and Experimentation using MPI

Ayush Raina, 22148

April 25, 2025

## Delta Stepping Algorithm

Delta-Stepping is a parallel algorithm for solving the **Single-Source Shortest Path (SSSP)** problem, particularly effective on graphs with both **light** and **heavy** edges. It was introduced by Ulrich Meyer and Peter Sanders to achieve high performance on shared and distributed memory systems by exploiting parallelism in edge relaxation.

### Problem Definition

Given a graph $G = (V, E)$, where $V$ represents the set of vertices and $E$ represents the set of weighted edges with non-negative weights, and a source vertex $s \in V$, the objective of the algorithm is to compute the shortest path distance $d(v)$ from the source $s$ to every vertex $v \in V$.

### Key Concept: Light and Heavy Edges

Edges are categorized based on a threshold parameter $\Delta > 0$:

- **Light Edges**: Edges with weight $w(e) \leq \Delta$
- **Heavy Edges**: Edges with weight $w(e) > \Delta$

This distinction allows the algorithm to process light edges concurrently while deferring the processing of heavy edges, thereby enhancing parallel efficiency.

### Algorithm Workflow

1. **Initialization:**
   - Set the distance to the source vertex $s$ as $d(s) = 0$, and to all other vertices as infinity $d(v) = \infty$ for $v \neq s$.
   - Initialize a collection of buckets $B[i]$, where each bucket stores vertices with distances in the range:

$$B[i] = \{v \mid i \times \Delta \leq d(v) < (i + 1) \times \Delta\}$$

2. **Bucket Processing Loop:**
   - While there are non-empty buckets:
     - Extract and process the **smallest non-empty bucket** $B[i]$.
     - Relax **light edges** of vertices in the bucket and update distances.
     - Move vertices to appropriate future buckets based on updated distances.
3. **Handling Heavy Edges:**
   - After all light edges are processed, relax heavy edges.
   - If the distance of a vertex changes, place it in the appropriate bucket.
4. **Termination:**
   - The algorithm terminates when all buckets are empty, ensuring all shortest paths are computed.

### Mathematical Formulation

For a vertex $v$ with current distance $d(v)$, when relaxing an edge $(v, u)$ with weight $w$:

$$d(u) = \min(d(u), d(v) + w)$$

If $w \leq \Delta$, the edge is considered **light** and processed immediately. Otherwise, it is **heavy** and deferred.

# Methodology

## Parallel Delta-Stepping Algorithm Implementation

The parallel implementation of the Delta-Stepping algorithm aims to efficiently compute single-source shortest paths in weighted graphs by distributing computation across multiple processes using MPI (Message Passing Interface). This section details the key components, data structures, and methods used in our implementation.

### Data Structures

I have used the following user defined data structures to support the distributed computation:

- **Edge**: A basic structure representing graph edges.

```
struct Edge {
    int Source;
    int Destination;
    int Weight;
};
```

- **DistributedVertexMap**: Maps between global and local vertex IDs, essential for distributed processing.

```
struct DistributedVertexMap {
    unordered_map<int, int> LocalToGlobalMap;
    unordered_map<int, int> GlobalToLocalMap;
};
```

- **VertexInfo**: Stores adjacency information for each vertex.

```
struct VertexInfo {
    vector<pair<int, int>> VertexEdges; // (destVertex, weight)
};
```

- **VertexDistance**: Stores the current shortest path distance to a vertex.

```
struct VertexDistance {
    int distance;
};
```

- **DistanceUpdate**: Structure for communicating distance updates between processes.

```
struct DistanceUpdate {
    int globalVertex;
    int newDistance;
};
```

- **Buckets**: A vector of vectors where each inner vector contains vertices with similar tentative distances.

```
vector<vector<int>> buckets;
```

### Communication Management

Inter-process communication is handled by the **CommunicationBuffer** class, which:

- Maintains a buffer of distance updates to be exchanged between processes
- Handles the transmission of updates using MPI
- Processes incoming updates and applies them to the local state
- Maintains consistency between distance values and bucket assignments

```
class CommunicationBuffer {
    // Stores pending distance updates
    vector<DistanceUpdate> updates;

    // References to shared data structures
    vector<VertexDistance>& LocalDistances;
    DistributedVertexMap& VertexMap;
    vector<vector<int>>& Buckets;
    int Delta;

    // Maps vertices to their current bucket
    unordered_map<int, int> vertexToBucketMap;

    // Methods for managing updates
    void addUpdate(int globalVertex, int newDistance);
    void SendUpdates(MPI_Comm comm, int DestinationRank);
    void ReceiveUpdates(MPI_Comm comm, int sourceRank);
    void ProcessAllUpdates(MPI_Comm comm);
};
```

### Algorithm Implementation

The core algorithm is encapsulated in the **DeltaSteppingAlgorithm** class:

```
class DeltaSteppingAlgorithm {
    // References to data structures
    vector<VertexInfo>& localGraph;
    vector<VertexDistance>& localDistances;
    vector<vector<int>>& buckets;
    CommunicationBuffer& commBuffer;
    DistributedVertexMap& vertexMap;
    int delta;
    int rank;
    int size;
    MPI_Comm comm;

    // Performance metrics
    struct AlgorithmMetrics {
        int totalBucketsProcessed;
        int totalVerticesProcessed;
        int totalLightEdgesProcessed;
        int totalHeavyEdgesProcessed;
        int totalRelaxations;
        int totalCommunicationRounds;
    } metrics;

    // Methods
    void initializeBuckets();
    EdgeType classifyEdge(int weight);
    int findNextNonEmptyBucket(int startBucket);
    int findGlobalNextBucket(int localNextBucket, MPI_Comm comm);
    void relaxEdge(int sourceLocalIndex, int destGlobalVertex, int edgeWeight);
    void processLocalBuckets();
    void processLightEdges(int vertexIndex);
    void processHeavyEdges(int vertexIndex);
};
```

## Algorithm Phases

The parallel Delta-Stepping algorithm execution involves several key phases:

### Initialization

1. **Graph Distribution**: The input graph is read from a file and distributed among processes.

```
vector<Edge> globalGraph = parseGraphFile("SyntheticGraph.txt");
DistributedVertexMap vertexMap = createVertexMapping(globalGraph, rank,
 numProcesses);
vector<VertexInfo> localGraph = distributeGraph(globalGraph, rank, numProcesses,
 vertexMap);
```

2. **Distance Initialization**: Distances are initialized (0 for source, infinity for others).

```
vector<VertexDistance> localDistances = initializeDistances(sourceVertex, vertexMap,
```

3. **Bucket Setup**: The source vertex is placed in bucket 0, and other structures are initialized.

```
vector<vector<int>> buckets;
```

**Main Processing Loop**

1. **Finding Active Buckets**: Processes collaborate to identify the next bucket to process.

```
int localNextBucket = findNextNonEmptyBucket(currentBucketIndex);
int globalNextBucket = findGlobalNextBucket(localNextBucket, comm);
```

2. **Bucket Processing**: Each process processes vertices in the current bucket.

```
if (globalNextBucket < buckets.size() && !buckets[globalNextBucket].empty()) {
    vertices = std::move(buckets[globalNextBucket]);
    buckets[globalNextBucket].clear();

    // Process light edges first
    for (int vertex : vertices) {
        processLightEdges(vertex);
    }

    // Process heavy edges next
    for (int vertex : vertices) {
        processHeavyEdges(vertex);
    }
}
```

3. **Edge Classification and Relaxation**:
   - Light edges (weight $\leq \delta$) are processed first.
   - Heavy edges (weight $> \delta$) are processed after light edges.
   - Edge relaxation may update distances and bucket assignments.

```
EdgeType classifyEdge(int weight) {
    return (weight <= delta) ? Light : Heavy;
}

void relaxEdge(int sourceLocalIndex, int destGlobalVertex, int edgeWeight) {
    int currentDistance = localDistances[sourceLocalIndex].distance;
    if (currentDistance == numeric_limits<int>::max())
        return;

    int newDistance = currentDistance + edgeWeight;
    // Update distances if better path found
    // ...
}
```

4. **Communication and Synchronization**: After processing, processes exchange distance updates.

```
commBuffer.ProcessAllUpdates(comm);
```

**Result Collection and Performance Measurement**

1. **Timer Implementation**: Performance is measured using MPI timing functions.

   ```
   double StartTime = MPI_Wtime();
   // Algorithm execution
   double EndTime = MPI_Wtime();
   double elapsedTime = EndTime - StartTime;
   ```

2. **Performance Metrics**: Several metrics are collected for analysis.

   ```
   struct AlgorithmMetrics {
       int totalBucketsProcessed = 0;
       int totalVerticesProcessed = 0;
       int totalLightEdgesProcessed = 0;
       int totalHeavyEdgesProcessed = 0;
       int totalRelaxations = 0;
       int totalCommunicationRounds = 0;
   } metrics;
   ```

3. **Result Aggregation**: Final results are collected from all processes to rank 0.

   ```
   gatherAndSaveFinalResults(localDistances, vertexMap, sourceVertex, comm);
   ```

## Key Implementation Considerations

### Load Balancing

Vertices are distributed among processes using a modulo-based approach, where vertex $v$ is assigned to process $v \bmod p$ (where $p$ is the number of processes). This simple scheme aims to balance the computational load.

### Delta Parameter

The $\delta$ parameter (set to 1 in our implementation) determines the bucket size and influences:

- Edge classification into light and heavy edges
- Granularity of parallel execution
- Communication frequency between processes

### Termination Condition

The algorithm terminates when no process has more vertices to process, detected through a collective operation:

```
int hasMoreWork = (localNextBucket != -1) ? 1 : 0;
int totalWork = 0;
MPI_Allreduce(&hasMoreWork, &totalWork, 1, MPI_INT, MPI_SUM, comm);
if (totalWork == 0) {
    // Algorithm terminates
    break;
}
```

## Verification and Validation

The implementation includes functions to:

- Verify correct graph distribution across processes
- Validate results by comparing against sequential Dijkstra's algorithm output
- Collect and analyze performance metrics

```
bool isResultMatch = compareOutputFiles("dijkstra_output.txt",
                                        "ParallelDeltaSteppingOutput.txt");
```

## Experimental Setup

All the experiments are done on teaching cluster provided to us. The graph used in experiementation contains 1,971,281 vertices and 5,533,214 edges with randomly assigned weights $w(e) \in [1, 100]$. We are choosing source vertex to be $0$. Following is the execution times for 1,2,4,8,16,32,64 processors along with Speedup and Efficiency for different Delta Values

| Delta | 1 Proc | 2 Proc | 4 Proc | 8 Proc | 16 Proc | 32 Proc | 64 Proc |
|---|---|---|---|---|---|---|---|
| 1 | 4.503 | 3.515 | 3.100 | 3.026 | 2.939 | 3.467 | 8.079 |
| 5 | 4.536 | 3.523 | 3.091 | 2.825 | 2.947 | 3.399 | 8.108 |
| 10 | 4.530 | 3.631 | 3.090 | 2.908 | 3.053 | 3.412 | 7.903 |
| 20 | 4.507 | 3.536 | 3.042 | 2.892 | 2.990 | 3.421 | 8.117 |
| 30 | 4.493 | 3.619 | 3.121 | 2.942 | 2.983 | 3.434 | 7.930 |
| 40 | 4.500 | 3.525 | 3.120 | 2.938 | 2.973 | 3.490 | 8.353 |
| 50 | 4.504 | 3.538 | 3.186 | 2.965 | 3.000 | 3.420 | 7.992 |
| 60 | 4.507 | 3.516 | 3.185 | 2.928 | 2.916 | 3.583 | 8.651 |
| 70 | 4.498 | 3.455 | 3.096 | 2.877 | 2.971 | 3.563 | 8.428 |
| 80 | 4.536 | 3.459 | 3.064 | 3.042 | 2.922 | 4.855 | 8.156 |
| 90 | 4.505 | 3.440 | 3.259 | 3.032 | 2.945 | 3.570 | 8.059 |
| 100 | 4.511 | 3.543 | 3.148 | 2.955 | 2.964 | 3.376 | 8.425 |

Table 1: Execution times (in seconds) for different processor counts and Delta values

In the above table, the execution time for involving more than 1 processor is reported for the processor which took the maximum time. Here are some plots to visualize the above data.

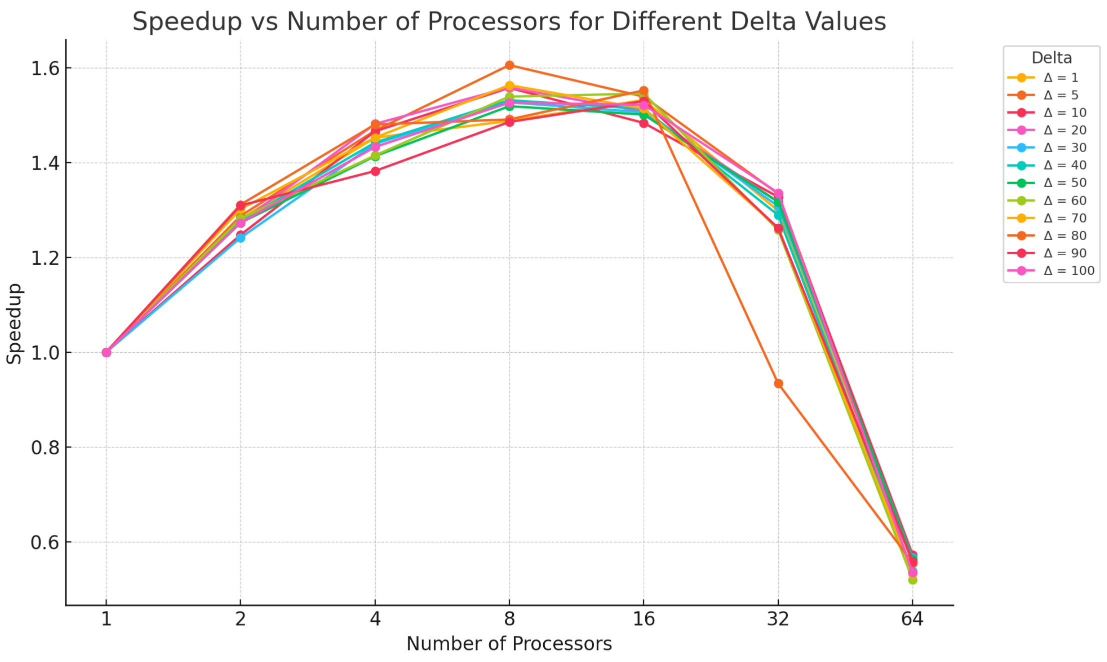

Figure 1: Speedup vs Number of Processors for Different Delta Values

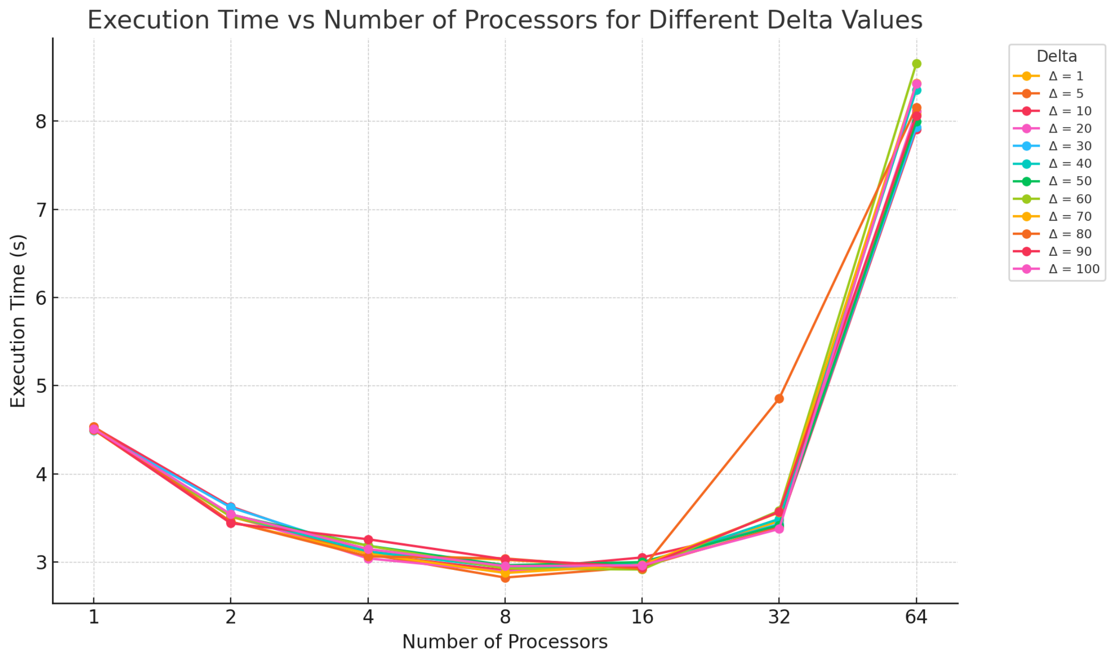

Figure 2: Execution Time vs Number of Processors for Different Delta Values

## Parallel Overhead Calculation

We can also calculate Parallel Overhead which quantifies extra work incurred due to parallelization - such as communication, synchronization and load imbalance. It is defined as $PO = T_N \times P - T_1$ where $T_N$ is the execution time for N processors and $T_1$ is the execution time for 1 processor. Here is the plot for Parallel Overhead for different Delta Values.

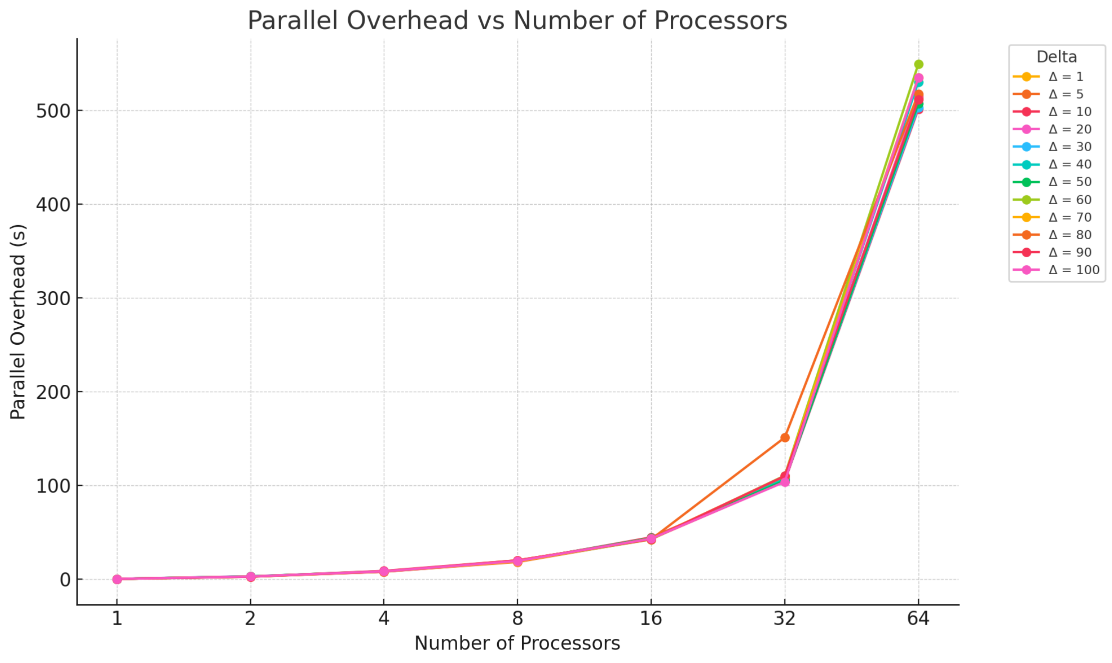

Figure 3: Parallel Overhead vs Number of Processors for Different Delta Values

One more reason, I think for huge increase in overhead when we move to 64 processor from 32 processors is because the machine in which we are running the code has 32 cores. I tried all these experiments with my 24 core machine also in which same pattern followed. Initially execution time kept decreasing until 16 cores, then if I use **–oversubscribe** flag to run with 32 and 64 processors, the execution time increased.

Here is efficiency plot for different Delta Values.

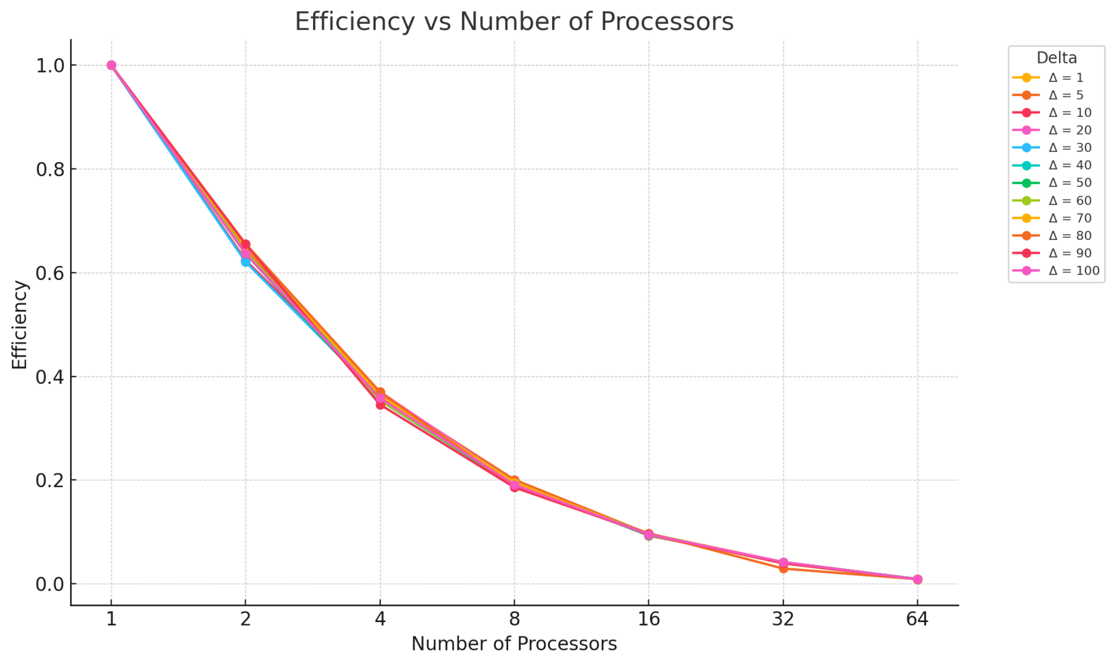

Figure 4: Efficiency vs Number of Processors for Different Delta Values

From above plot we can see that even we are lowering the execution time till 8 processors, but still the resource utilization is not up to the mark. This is because of communication overheads, synchronization overheads as Delta Stepping Algorithm is a bulk synchronization algorithm, and non trivial programming using MPI. My implementation is around 1000 lines of code and I used **modulus** operator to distribute the vertices among processors, due to which all processors have got almost equal number of vertices to process. So **load imbalance** is not an issue, **communication and synchronization overheads** are the main reasons for low efficiency.

## Hybrid CPU-GPU Implementation Methodology

Our hybrid implementation combines CUDA for GPU-accelerated edge relaxation with CPU-based bucket management to achieve high-performance shortest path computation. This section details the design, data structures, and algorithms used in our CUDA-based Delta-Stepping implementation.

### System Architecture

The implementation follows a heterogeneous computing model:

- **CPU**: Manages algorithm control flow, bucket processing, and data orchestration
- **GPU**: Accelerates the computationally intensive edge relaxation operations through CUDA kernels

### Data Structures

#### Graph Representation

The graph is stored in multiple formats to optimize different operations:

```
struct Graph {
    int num_nodes = 0;
    int num_edges = 0;
    vector<vector<pair<int, int>>> adj_list; // [node] -> list of (dest, weight)
    vector<int> src, dst, weight; // Edge list for CUDA
};
```

- **Adjacency List**: Traditional representation for CPU operations
- **Edge List Arrays**: Three parallel arrays (source, destination, weight) optimized for GPU memory access patterns

**Distance Array**

A single array tracks the tentative shortest distance to each node:

```
vector<int> dist(num_nodes, DIST_INF);
```

**Bucket Structure**

Buckets are implemented as a vector of sets, where each set contains vertices with distances in a specific range:

```
const int MAX_BUCKET_COUNT = 1000;
vector<set<int>> buckets(MAX_BUCKET_COUNT);
```

**Performance Metrics**

A dedicated structure collects various algorithm performance metrics:

```
struct PerformanceMetrics {
    double total_time_seconds;
    int iteration_count;
    vector<double> per_bucket_time;
    vector<int> bucket_sizes;
    double data_transfer_time;
    double kernel_execution_time;
    double bucket_processing_time;
};
```

## GPU Kernel Design

The core of the computation is performed by the CUDA kernel that relaxes edges for vertices in the current bucket:

```
__global__ void relaxEdgesKernel(
    int* d_edges_src, int* d_edges_dst, int* d_edges_weight, int num_edges,
    int* d_dist, bool* d_changed, int* d_bucket_nodes, int num_bucket_nodes,
    int delta, int current_bucket) {

    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < num_bucket_nodes) {
        int node = d_bucket_nodes[idx];

        // Only process if the node's distance is valid
        if (d_dist[node] != DIST_INF) {
            // Iterate through edges
            for (int e = 0; e < num_edges; e++) {
                if (d_edges_src[e] == node) {
                    int v = d_edges_dst[e];
                    int w = d_edges_weight[e];

                    // Calculate new distance
                    int new_dist = d_dist[node] + w;

                    // Check for overflow and only update if new_dist is less
                    if (new_dist >= 0 && (d_dist[v] == DIST_INF || new_dist < d_dist[v])) {
                        atomicMin(&d_dist[v], new_dist);
                        *d_changed = true;
                    }
                }
            }
        }
    }
}
```

Key features of the kernel:

- **Thread Structure**: Each thread processes a single vertex from the current bucket

- **Atomic Operations**: Uses `atomicMin()` to safely update distances in parallel
- **Change Detection**: Sets a global flag when any distance is updated
- **Edge Traversal**: Searches through all edges to find those originating from the current node

## Algorithm Implementation

The hybrid Delta-Stepping algorithm follows these key steps:

### Initialization Phase

1. **Graph Loading**: Read graph from file into memory

```
Graph graph = readGraph("SyntheticGraph.txt");
```

2. **Distance Array Initialization**: Set source node distance to 0, all others to infinity

```
vector<int> dist(num_nodes, DIST_INF);
dist[source] = 0;
```

3. **Bucket Initialization**: Place source node in bucket 0

```
buckets[0].insert(source);
```

4. **GPU Memory Allocation**: Allocate and initialize device memory

```
checkCudaError(cudaMalloc(&d_edges_src, num_edges * sizeof(int)),
 "allocate d_edges_src");
checkCudaError(cudaMalloc(&d_edges_dst, num_edges * sizeof(int)),
 "allocate d_edges_dst");
checkCudaError(cudaMalloc(&d_edges_weight, num_edges * sizeof(int)),
"allocate d_edges_weight");
checkCudaError(cudaMalloc(&d_dist, num_nodes * sizeof(int)),
 "allocate d_dist");
checkCudaError(cudaMalloc(&d_changed, sizeof(bool)),
 "allocate d_changed");
checkCudaError(cudaMalloc(&d_bucket_nodes, num_nodes * sizeof(int)),
"allocate d_bucket_nodes");
```

5. **Data Transfer to GPU**: Copy graph structure and initial distances to GPU memory

```
checkCudaError(cudaMemcpy(d_edges_src, graph.src.data(), num_edges * sizeof(int),
                          cudaMemcpyHostToDevice), "copy edges_src");
checkCudaError(cudaMemcpy(d_edges_dst, graph.dst.data(), num_edges * sizeof(int),
                          cudaMemcpyHostToDevice), "copy edges_dst");
checkCudaError(cudaMemcpy(d_edges_weight, graph.weight.data(),
 num_edges * sizeof(int), cudaMemcpyHostToDevice), "copy edges_weight");
checkCudaError(cudaMemcpy(d_dist, dist.data(), num_nodes * sizeof(int),
                          cudaMemcpyHostToDevice), "copy initial dist");
```

### Main Processing Loop

1. **Bucket Selection**: Find the smallest non-empty bucket

```
int smallest_bucket = -1;
for (int b = current_bucket; b < buckets.size(); b++) {
    if (!buckets[b].empty()) {
        smallest_bucket = b;
        break;
    }
}
```

2. **Bucket Extraction**: Move nodes from current bucket to a working set

   ```
   vector<int> bucket_nodes(buckets[smallest_bucket].begin(),
    buckets[smallest_bucket].end());
   buckets[smallest_bucket].clear();
   ```

3. **Kernel Preparation**: Copy current bucket nodes to GPU

   ```
   checkCudaError(cudaMemcpy(d_bucket_nodes, bucket_nodes.data(),
                             bucket_nodes.size() * sizeof(int),
                             cudaMemcpyHostToDevice), "copy bucket nodes");
   ```

4. **Kernel Execution**: Launch CUDA kernel to relax edges

   ```
   int threadsPerBlock = 256;
   int blocksPerGrid = (bucket_nodes.size() + threadsPerBlock - 1) / threadsPerBlock;

   relaxEdgesKernel<<<blocksPerGrid, threadsPerBlock>>>(
       d_edges_src, d_edges_dst, d_edges_weight, num_edges,
       d_dist, d_changed, d_bucket_nodes, bucket_nodes.size(),
       delta, smallest_bucket
   );
   ```

5. **Result Retrieval**: Copy updated distances and change flag back from GPU

   ```
   checkCudaError(cudaMemcpy(&changed, d_changed, sizeof(bool),
                             cudaMemcpyDeviceToHost), "copy changed flag");
   checkCudaError(cudaMemcpy(dist.data(), d_dist, num_nodes * sizeof(int),
                             cudaMemcpyDeviceToHost), "copy distances");
   ```

6. **Bucket Update**: Reassign nodes to buckets based on updated distances

   ```
   if (changed) {
       for (int i = 0; i < num_nodes; i++) {
           // If distance was updated and is now finite
           if (dist[i] != DIST_INF) {
               int bucket_id = dist[i] / delta;

               // Only insert if not already processed in current iteration
               if (i != source && find(bucket_nodes.begin(), bucket_nodes.end(), i)
                == bucket_nodes.end()) {
                   buckets[bucket_id].insert(i);
               }
           }
       }
   }
   ```

**Termination and Results**

1. **Algorithm Termination**: The algorithm terminates when either:
   - No more non-empty buckets exist
   - No distance updates were made in the last iteration
   - Maximum iteration count is reached (safety measure)
2. **Resource Cleanup**: Free GPU memory

   ```
   cudaFree(d_edges_src);
   cudaFree(d_edges_dst);
   cudaFree(d_edges_weight);
   ```

```
    cudaFree(d_dist);
    cudaFree(d_changed);
    cudaFree(d_bucket_nodes);
```

3. **Result Validation**: Compare results with sequential Dijkstra implementation

```
    bool isResultMatch = compareOutputFiles("dijkstra_output.txt",
                                             "HybridDeltaSteppingResults.txt");
```

### Performance Analysis

Several performance metrics are collected during the algorithm execution:

#### Timing Metrics

- **Total Execution Time**: End-to-end runtime of the algorithm
- **Data Transfer Time**: Time spent transferring data between CPU and GPU
- **Kernel Execution Time**: Time spent in GPU computation
- **Per-Bucket Processing Time**: Processing time for each individual bucket

## Experimental Results

We implemented the same Delta Stepping Algorithm using Open MP and CUDA. We used Open MP for parallelizing the bucket processing loop and CUDA for parallelizing the edge relaxation step. We used 8 threads for Open MP and 256 threads per block for CUDA.Here is the execution times for different Delta Values.

Table 2: Execution Times for Different Delta Values (OpenMP + CUDA)

| Delta Value | Execution Time (s) |
|---|---|
| 1 | 6.1021 |
| 5 | 3.4281 |
| 10 | 3.3989 |
| 20 | 1.5955 |
| 30 | 2.4969 |
| 40 | 1.5924 |
| 50 | 1.3124 |
| 60 | 1.3066 |
| 70 | 1.61707 |
| 80 | 1.5882 |
| 90 | 2.2041 |
| 100 | 1.3141 |

Assuming Sequential Execution Time (T_seq): 4.503 seconds

$$\text{Speedup} = \frac{T_{seq}}{T_{par}} \tag{1}$$

and

$$\text{Efficiency} = \frac{\text{Speedup}}{P} \tag{2}$$

Where $T_{par}$ is the execution time for parallel implementation and $P$ is the number of processors which is 8 in this case.

Here are the plots for speedup, execution times and efficiency for OpenMP + CUDA implementation for different Delta Values.

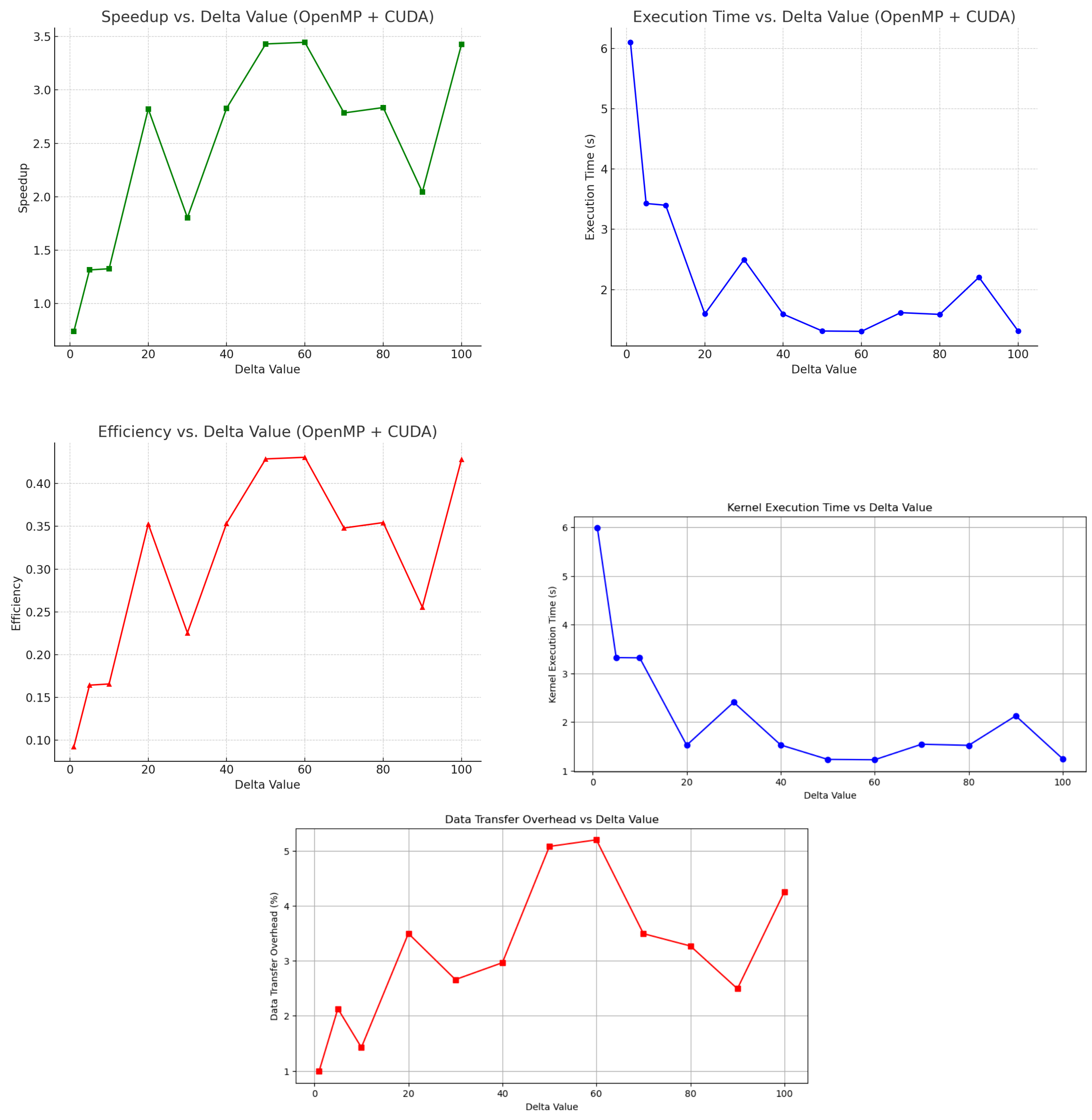

Figure 5: Speedup vs Delta Values for OpenMP + CUDA Implementation

## Results Analysis

### Effect of Delta Parameter on Performance

The delta parameter in the Delta-Stepping algorithm determines bucket width and has a significant impact on performance. Our experiments with the hybrid OpenMP-CUDA implementation reveal several key insights about how varying delta affects different performance aspects.

#### Execution Time Analysis

Table 3 and Figure 6 show the relationship between delta values and execution time. Several observations can be made:

- **Small Delta Values (1-10):** With small delta values, execution times are significantly higher (6.10s for $\delta = 1$, 3.43s for $\delta = 5$). This occurs because small buckets result in frequent GPU kernel launches and memory transfers, increasing overhead.
- **Medium Delta Values (50-60):** The optimal performance is achieved at $\delta = 60$ with 1.31 seconds, representing a 3.44× speedup over sequential execution. This represents the sweet spot where buckets contain enough vertices to fully utilize GPU parallelism without excessive bucket management overhead.
- **Non-Monotonic Behavior:** Interestingly, the relationship is not monotonic. Performance decreases at $\delta = 30$ (2.50s) and $\delta = 90$ (2.20s), showing that the relationship between delta and performance is complex and depends on the graph structure.

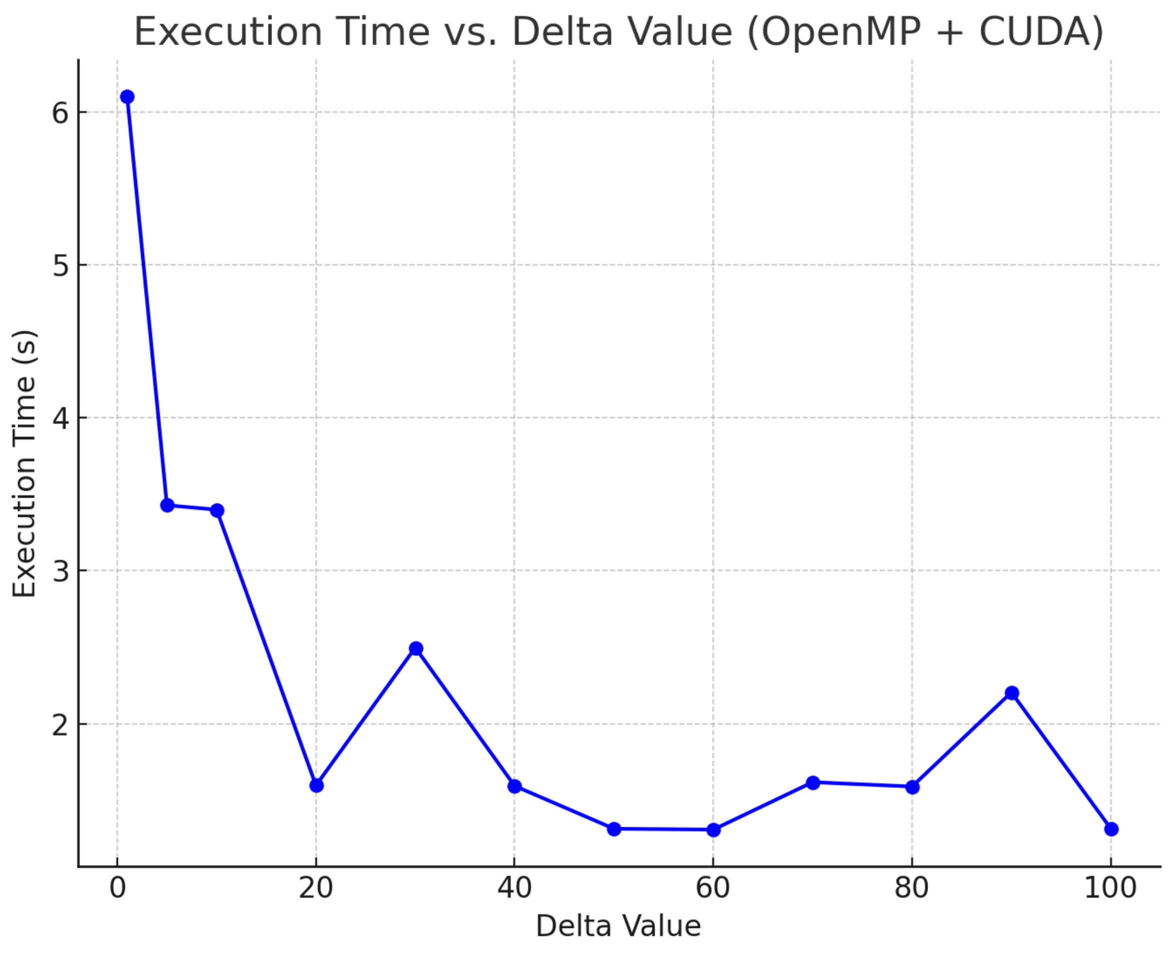

Figure 6: Execution Time vs. Delta Value showing how bucket width affects overall performance

### Kernel Execution Time

As shown in Figure 7, kernel execution time generally decreases as delta increases, with some exceptions:

- At $\delta = 1$, kernel execution time peaks at approximately 6 seconds due to a large number of small buckets requiring many kernel invocations.
- Performance improves significantly until $\delta = 20$, then follows a pattern similar to overall execution time.
- The relationship between kernel time and delta exhibits local minima at $\delta = 60$ and $\delta = 100$, suggesting these values create efficiently sized work units for the GPU.

### Data Transfer Overhead

Figure 8 shows the data transfer overhead as a percentage of total execution time:

- Data transfer overhead is lowest at $\delta = 1$ (approximately 1%), as each bucket contains fewer vertices requiring less data transfer.
- The overhead peaks at approximately 5% for $\delta = 50$ and $\delta = 60$, when larger buckets require more data to be transferred between CPU and GPU.
- The fluctuating pattern suggests that data transfer overhead is influenced by both bucket size and the number of iterations required for convergence.

## Parallel Performance Metrics

### Speedup Analysis

Figure 9 illustrates the speedup achieved by our hybrid implementation compared to sequential Dijkstra's algorithm:

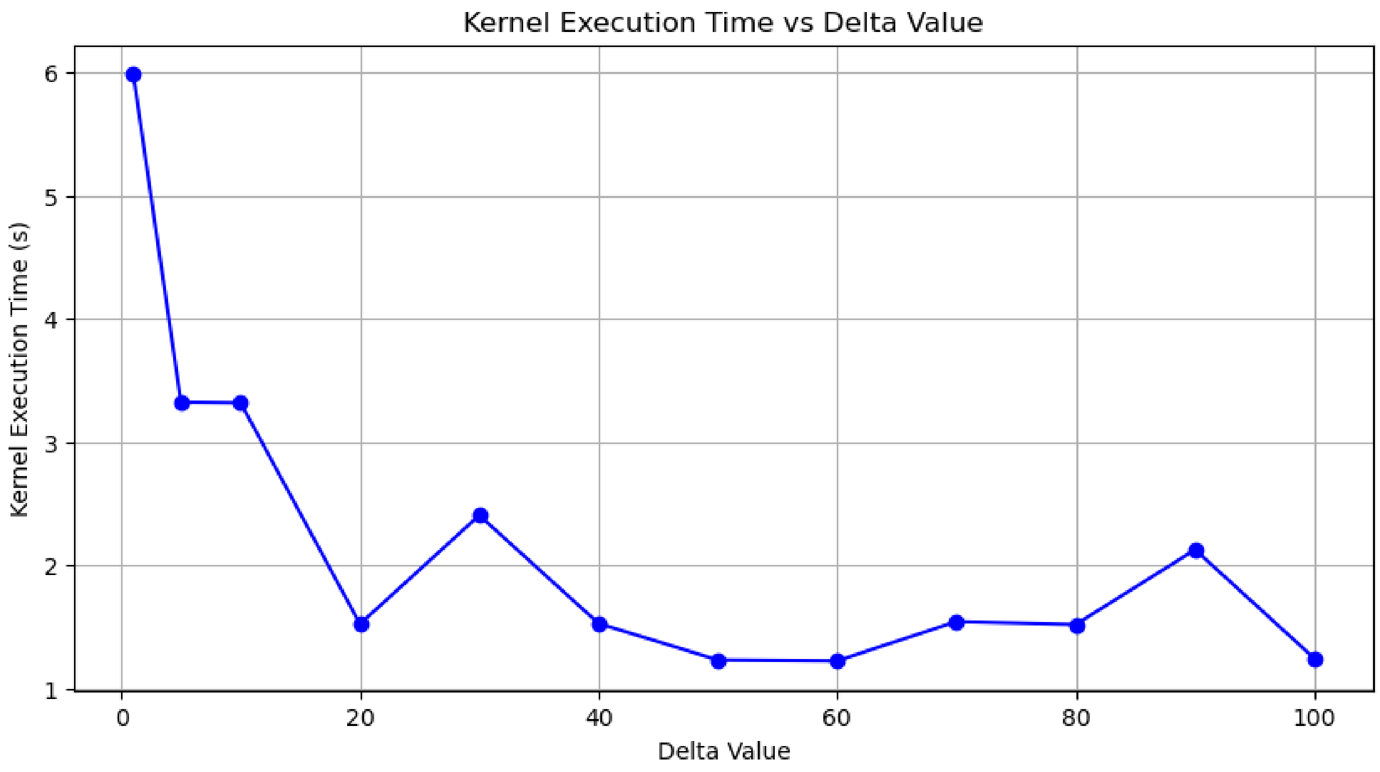

Figure 7: Kernel Execution Time vs. Delta Value showing GPU processing time for different bucket widths

- Speedup improves dramatically from $\delta = 1$ (0.74×) to $\delta = 60$ (3.44×), representing a 4.6× improvement in parallel performance through delta tuning alone.
- Maximum speedup of 3.44× is achieved at $\delta = 60$, with similar performance at $\delta = 100$ (3.43×).
- The speedup curve closely mirrors the inverse of execution time, confirming that delta tuning directly impacts parallel efficiency.

**Efficiency Analysis**

Figure 10 shows the parallel efficiency (speedup divided by processor count):

- Efficiency at $\delta = 1$ is only 9%, demonstrating the critical importance of delta tuning for efficient resource utilization.
- The efficiency curve shows that delta values between 50 and 60 offer the best resource utilization for our test graph.

## Performance Bottleneck Analysis

By comparing kernel execution time against data transfer overhead across different delta values, we can identify key performance bottlenecks:

- For small delta values ($\delta < 20$), performance is primarily limited by **excessive kernel launches** and **serialization** due to small bucket sizes, even though data transfer overhead is minimal.
- For medium delta values ($40 \leq \delta \leq 60$), performance is optimal despite increased data transfer overhead, as efficient GPU utilization compensates for the increased communication costs.
- For certain delta values (notably $\delta = 30$ and $\delta = 90$), performance anomalies occur that correlate with the specific graph structure, suggesting that these values create particularly inefficient bucket distributions for the test graph.

## Practical Implications

Our experimental results lead to several practical recommendations for implementing hybrid CPU-GPU Delta-Stepping:

- **Delta Parameter Selection:** For graphs similar to our test case, setting $\delta$ between 50 and 60 provides optimal performance. For other graphs, delta should be tuned based on average edge weight distribution.

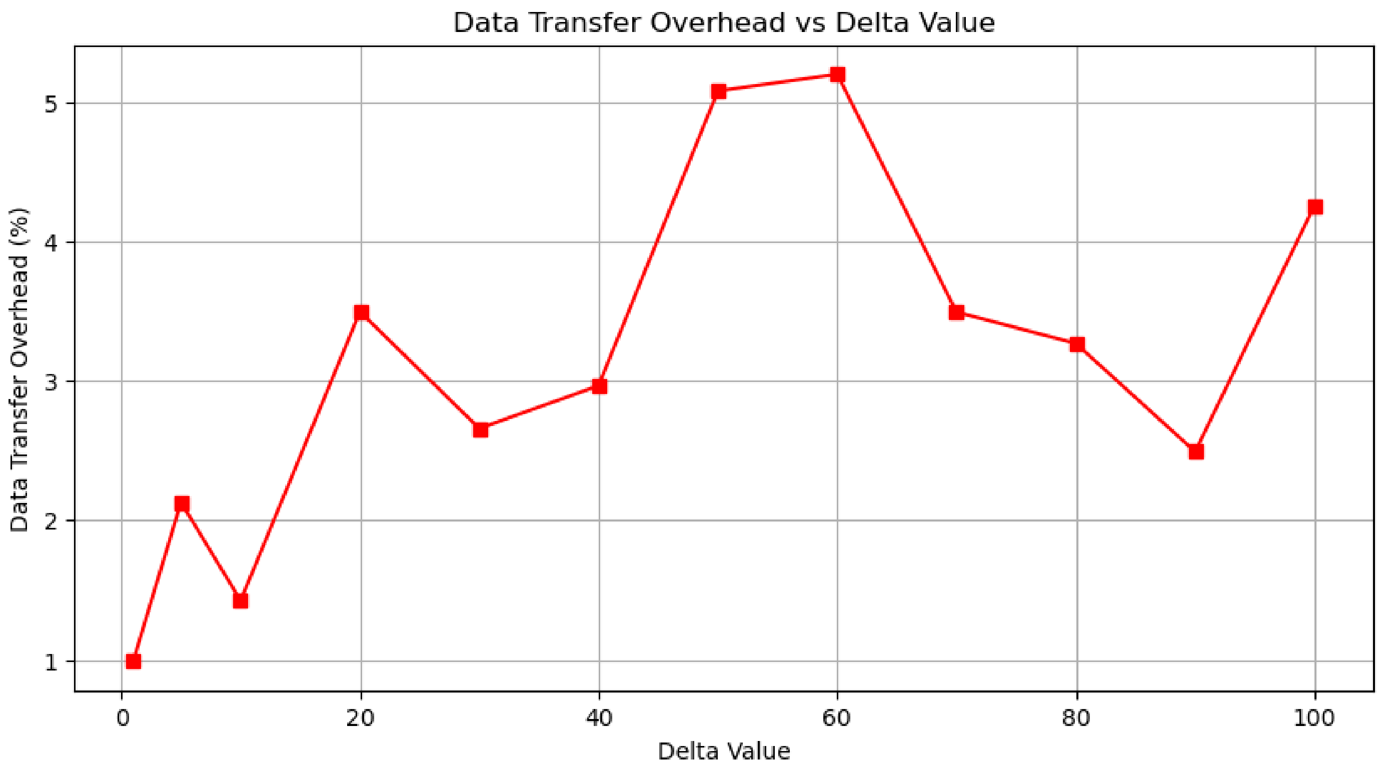

Figure 8: Data Transfer Overhead vs. Delta Value showing communication costs as percentage of total time

- **Memory Transfer Optimization:** While memory transfers contribute to overhead, they do not dominate performance (max 5% overhead), suggesting that our hybrid design effectively balances computation and communication.
- **Scalability:** The achieved efficiency of 43% with 8 CPU threads indicates good scalability potential, though further optimizations in bucket management could improve this further.

## Conclusion

The experimental results demonstrate that our hybrid OpenMP-CUDA implementation of the Delta-Stepping algorithm achieves significant speedups compared to sequential execution, with performance heavily dependent on the delta parameter. The optimal delta value of 60 achieves a 3.44× speedup and 43% efficiency on an 32-core system, confirming the effectiveness of the GPU-accelerated edge relaxation combined with CPU-based bucket management.

The non-monotonic relationship between delta and performance highlights the complexity of parallel shortest path algorithms, where work distribution, communication patterns, and graph characteristics all interact to determine overall efficiency. Our analysis provides valuable insights for tuning delta-stepping implementations on heterogeneous computing platforms, demonstrating that careful parameter selection can lead to substantial performance gains.

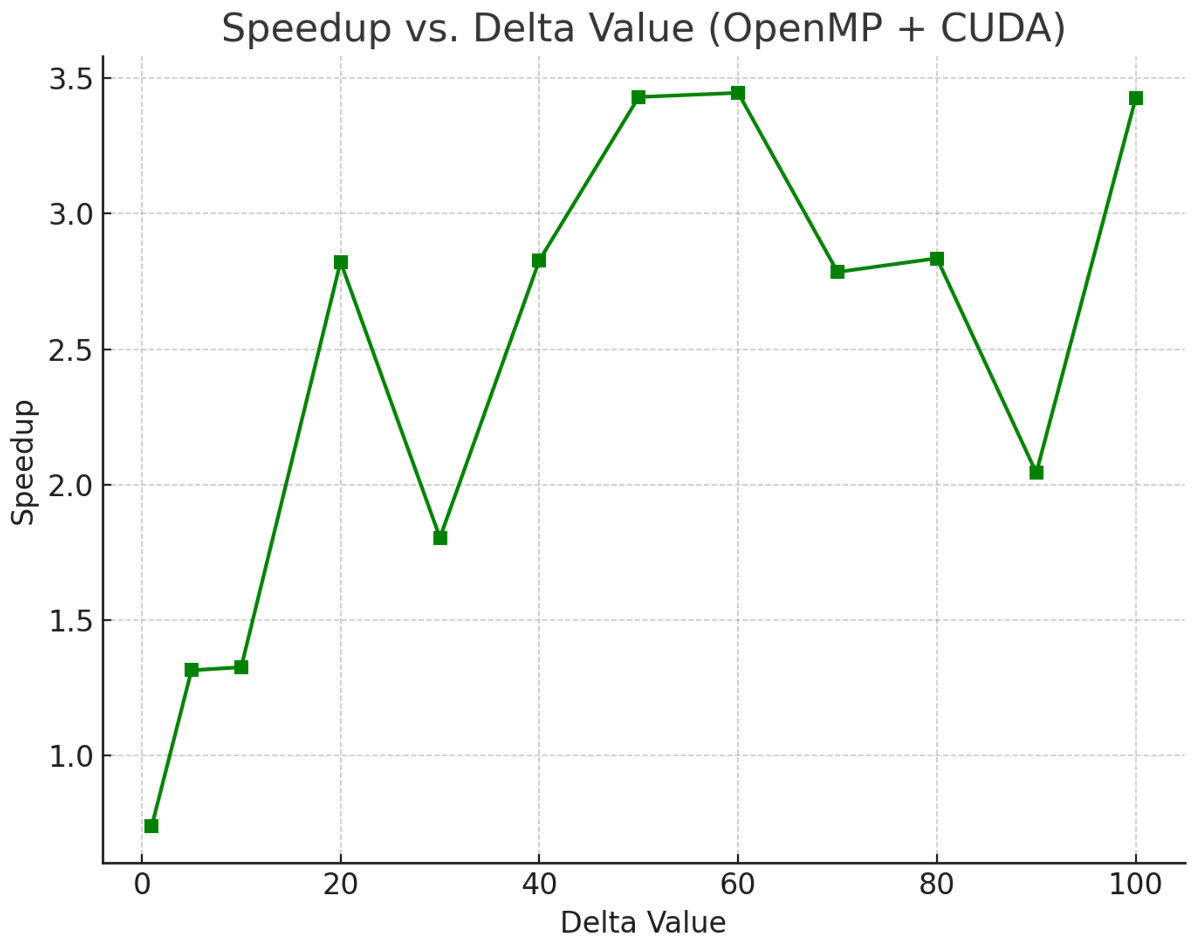

Figure 9: Speedup vs. Delta Value showing performance improvement relative to sequential algorithm

Table 3: Execution Times for Different Delta Values (OpenMP + CUDA)

| **Delta Value** | **Execution Time (s)** |
|---|---|
| 1 | 6.1021 |
| 5 | 3.4281 |
| 10 | 3.3989 |
| 20 | 1.5955 |
| 30 | 2.4969 |
| 40 | 1.5924 |
| 50 | 1.3124 |
| 60 | 1.3066 |
| 70 | 1.61707 |
| 80 | 1.5882 |
| 90 | 2.2041 |
| 100 | 1.3141 |

Assuming Sequential Execution Time (T_seq): 4.503 seconds

$$\text{Speedup} = \frac{T_{seq}}{T_{par}} \tag{3}$$

and

$$\text{Efficiency} = \frac{\text{Speedup}}{P} \tag{4}$$

Where $T_{par}$ is the execution time for parallel implementation and $P$ is the number of processors which is 8 in this case.

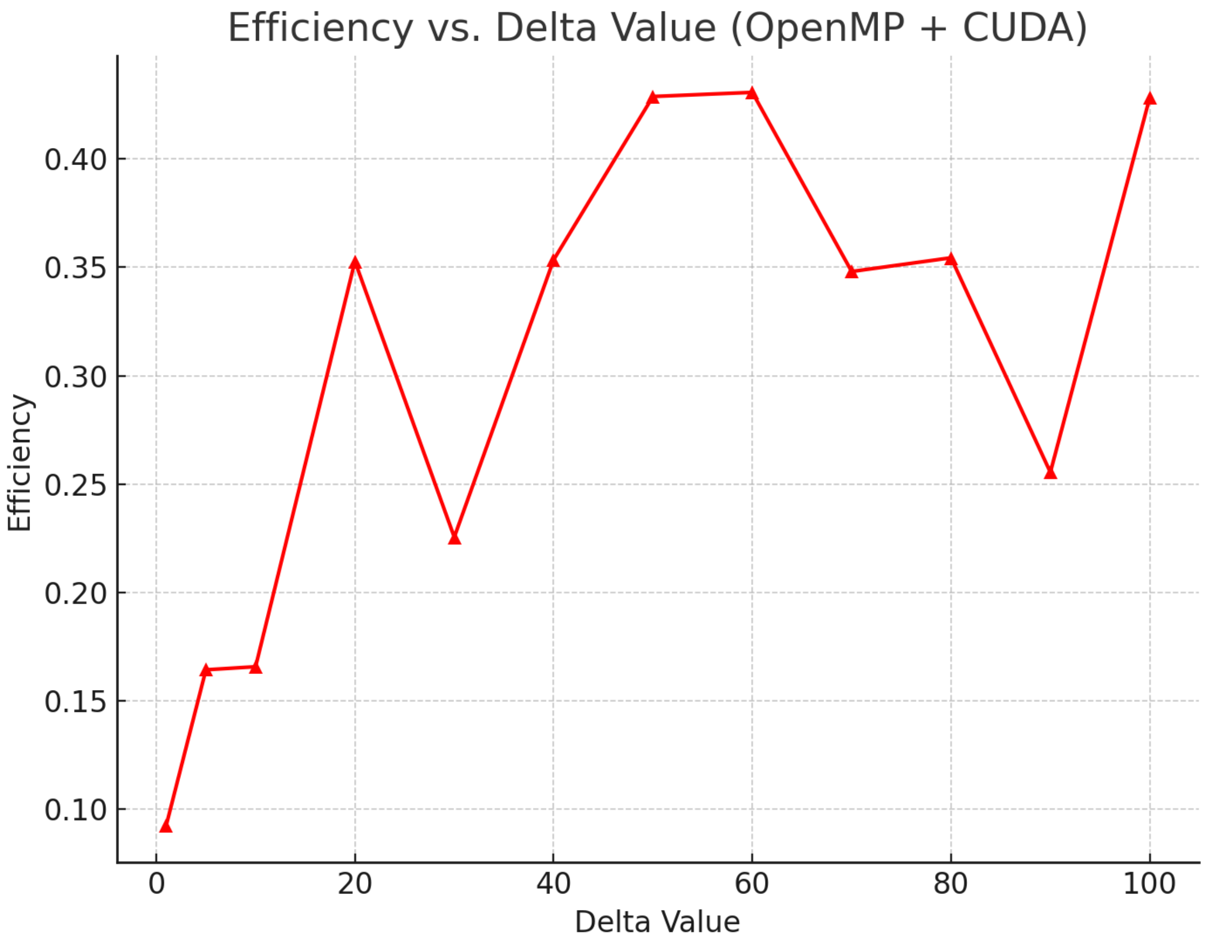

Figure 10: Efficiency vs. Delta Value showing how effectively system resources are utilized