Atlanta Logistics Hub Optimization

This project determines the optimal location for a package pickup facility in Atlanta, Georgia, targeting the top 10 most populated neighborhoods. We compare two distinct methodologies:

1. **Theoretical Optimal** (Haversine Distance)
2. **Real-world Optimal** (Driving Distance using Google Maps API)

Initial Setting

In [ ]:
```
# Install necessary libraries
!pip install googlemaps

import time
import googlemaps
import pandas as pd
import numpy as np
from geopy.geocoders import GoogleV3

# API Configuration
## Enter your own API key
API_KEY = 'your_api_key'
geolocator = GoogleV3(api_key=API_KEY)
gmaps = googlemaps.Client(key=API_KEY)
```

```
Requirement already satisfied: googlemaps in /opt/anaconda3/lib/python3.13/site-packages (4.10.0)
Requirement already satisfied: requests<3.0,>=2.20.0 in /opt/anaconda3/lib/python3.13/site-packages (from googlemaps) (2.32.3)
Requirement already satisfied: charset-normalizer<4,>=2 in /opt/anaconda3/lib/python3.13/site-packages (from requests<3.0,>=2.20.0->googlemaps) (3.3.
2)
Requirement already satisfied: idna<4,>=2.5 in /opt/anaconda3/lib/python3.13/site-packages (from requests<3.0,>=2.20.0->googlemaps) (3.7)
Requirement already satisfied: urllib3<3,>=1.21.1 in /opt/anaconda3/lib/python3.13/site-packages (from requests<3.0,>=2.20.0->googlemaps) (1.26.20)
Requirement already satisfied: certifi>=2017.4.17 in /opt/anaconda3/lib/python3.13/site-packages (from requests<3.0,>=2.20.0->googlemaps) (2025.4.26)
```

In [2]:
```
# Define neighborhoods dataset
data = {
    "area": [
        "Midtown",
        "Downtown",
        "Old Fourth Ward",
        "North Buckhead",
        "Pine Hills",
        "Morningside/Lenox Park",
        "Virginia-Highland",
        "Grant Park",
        "Georgia Tech",
        "Kirkwood"
    ],
    "population": [
        16569,
        13411,
        10505,
        8270,
        8033,
        8030,
        7800,
        6771,
        6607,
        5897
    ]
}

df = pd.DataFrame(data)
df
```

Out[2]:

	area	population
0	Midtown	16569
1	Downtown	13411
2	Old Fourth Ward	10505
3	North Buckhead	8270
4	Pine Hills	8033
5	Morningside/Lenox Park	8030
6	Virginia-Highland	7800
7	Grant Park	6771
8	Georgia Tech	6607
9	Kirkwood	5897

Data Preparation and Initial Mapping

In [3]:
```
# Create a refined search query for better API accuracy
def geocode_one(query):
    try:
        res = gmaps.geocode(query)
        time.sleep(0.15)

        if not res:
            return pd.Series([None, None, None])

        loc = res[0]["geometry"]["location"]
        formatted = res[0].get("formatted_address")

        return pd.Series([
            round(loc["lat"], 6),
```

```
            round(loc["lng"], 6),
            formatted
        ])
    except Exception as e:
        print("Error for:", query, e)
        return pd.Series([None, None, None])

df["query"] = df["area"] + ", Atlanta, GA"
df[["lat", "lng", "formatted_address"]] = df["query"].apply(geocode_one)

# Save results to CSV
df.to_csv("atl_neighborhoods_geocoded.csv", index=False)
df_final = df.drop(columns=["query"])
df_final
```

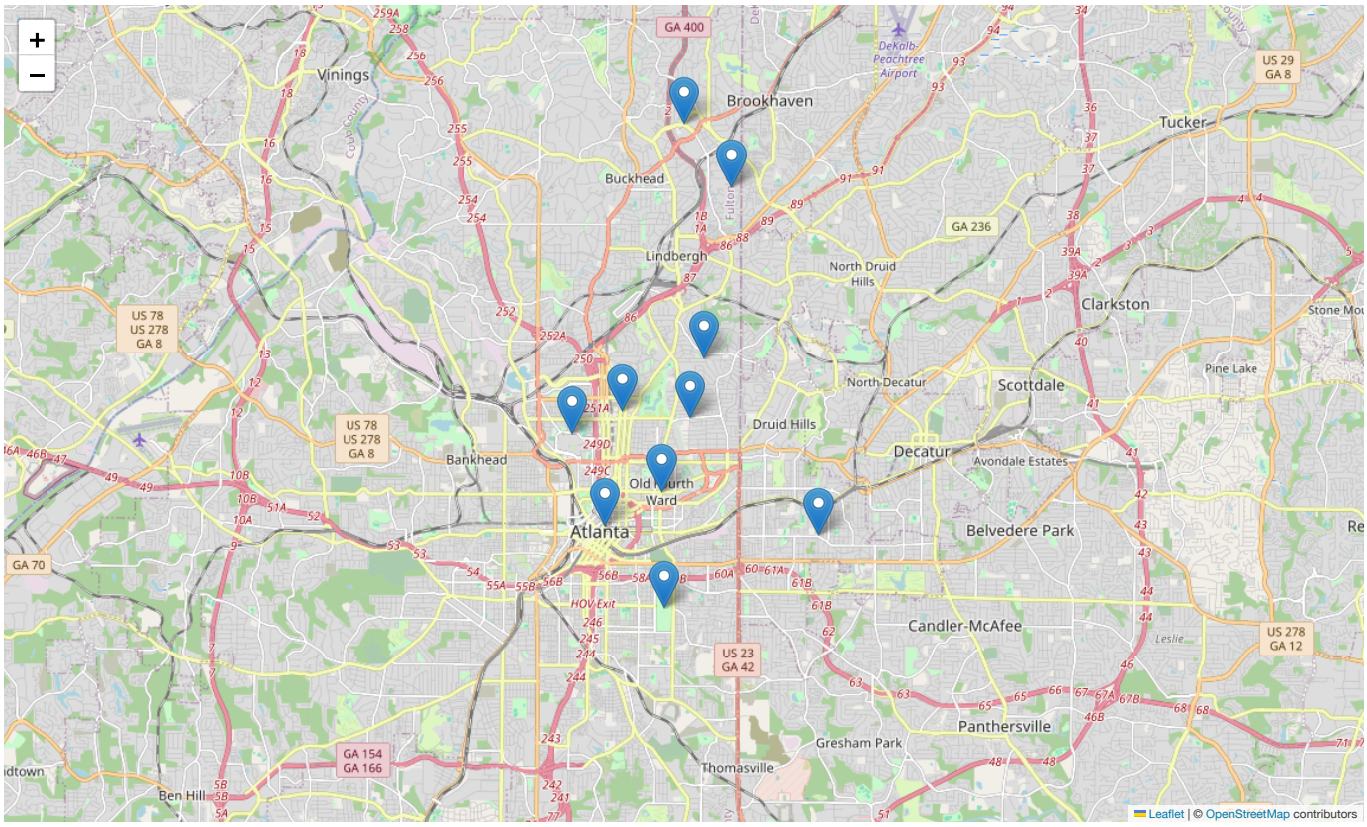
Out[3]:

	area	population	lat	lng	formatted_address
0	Midtown	16569	33.783315	-84.383117	Midtown Atlanta, Atlanta, GA, USA
1	Downtown	13411	33.755711	-84.388372	Downtown Atlanta, Atlanta, GA, USA
2	Old Fourth Ward	10505	33.763959	-84.371973	Old Fourth Ward, Atlanta, GA, USA
3	North Buckhead	8270	33.852656	-84.365375	North Buckhead, Atlanta, GA, USA
4	Pine Hills	8033	33.837536	-84.351576	Pine Hills, Atlanta, GA 30324, USA
5	Morningside/Lenox Park	8030	33.796156	-84.359463	Morningside-Lenox Park, Atlanta, GA, USA
6	Virginia-Highland	7800	33.781734	-84.363513	Virginia-Highland, Atlanta, GA 30306, USA
7	Grant Park	6771	33.735704	-84.371164	Grant Park, 840 Cherokee Ave SE, Atlanta, GA 3...
8	Georgia Tech	6607	33.777979	-84.397964	Atlanta, GA 30332, USA
9	Kirkwood	5897	33.753340	-84.326218	Kirkwood, Atlanta, GA, USA

In [4]:

```
# Map Initialization
!pip -q install folium
import folium

# Calculate the geographic center of all neighborhoods to center the map
center_lat = df["lat"].mean()
center_lng = df["lng"].mean()

# Create an interactive Folium map
m = folium.Map(
    location=[center_lat, center_lng],
    zoom_start=12,
    tiles="OpenStreetMap"
)

# Adding Markers
for _, r in df.iterrows():
    folium.Marker(
        location=[r["lat"], r["lng"]],
        popup=f"""
        <b>{r['area']}</b><br>
        Population: {r['population']:,}<br>
        {r['formatted_address']}
        """,
        tooltip=r["area"]
    ).add_to(m)

# Display the map
m
```

Out[4]:

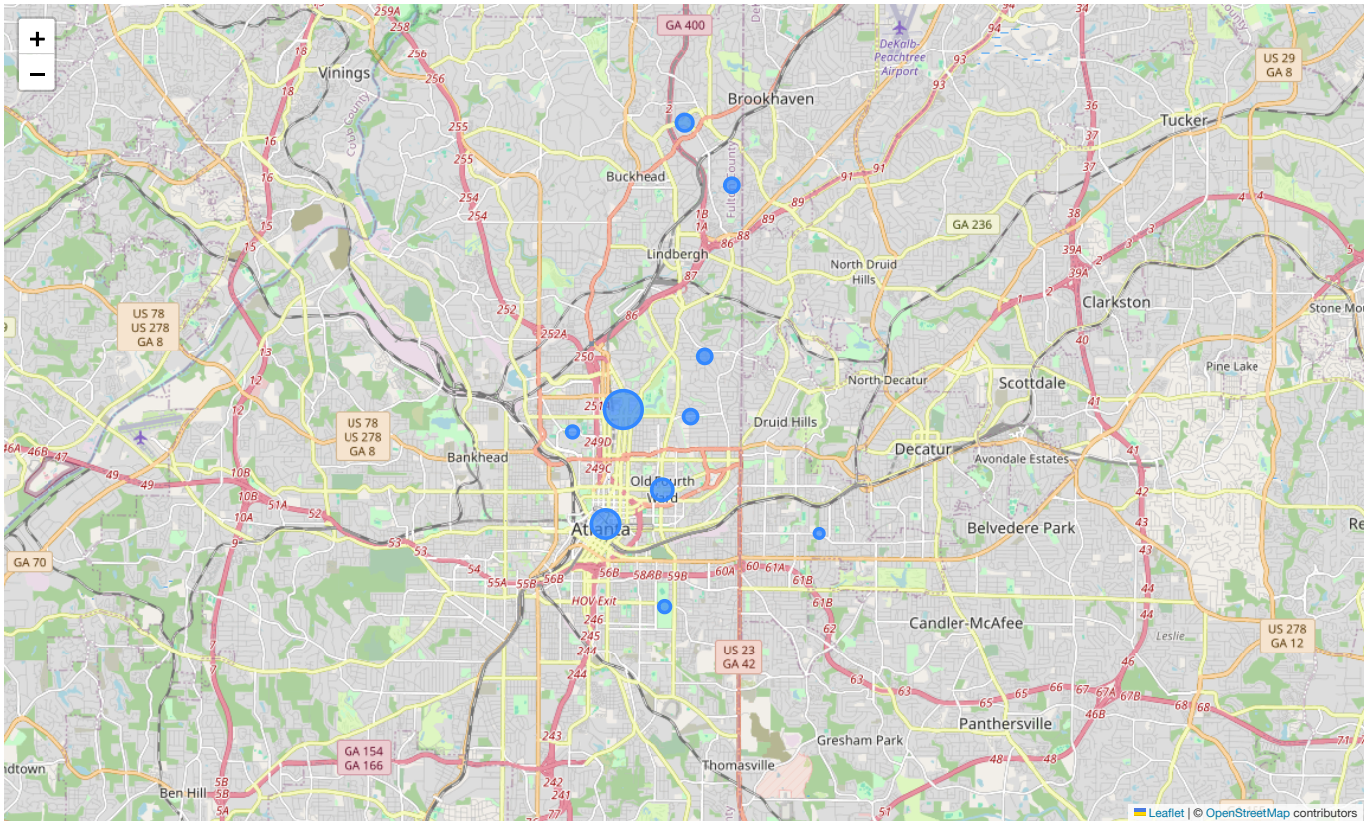


Population-Weighted Visualization

To better understand the demographic influence of each neighborhood, we visualize them using **CircleMarkers** where the radius is proportionally scaled to the population. This "Bubble Map" approach highlights which areas have the most weight in our subsequent location optimization models.

In [5]:
```
# Population Scaling Logic
pop_min, pop_max = df["population"].min(), df["population"].max()

def scale_radius(pop, rmin=4, rmax=16):
    if pop_max == pop_min:
        return rmin # Avoid division by zero
    return rmin + (pop - pop_min) * (rmax - rmin) / (pop_max - pop_min)

# Map Initialization
m_weighted = folium.Map(
    location=[center_lat, center_lng],
    zoom_start=12,
    tiles="OpenStreetMap"
)

# Adding Weighted CircleMarkers
for _, r in df.iterrows():
    folium.CircleMarker(
        location=[r["lat"], r["lng"]],
        radius=scale_radius(r["population"]),
        popup=f"""
        <b>{r['area']}</b><br>
        Population: {r['population']:,}
        """,
        tooltip=r["area"],
        fill=True,
        fill_opacity=0.7
    ).add_to(m_weighted)

# Display the map
m_weighted
```

Out[5]:



Haversine Distance Optmization

In [6]:
```
# Data Cleaning & Weight Normalization
df2 = df.dropna(subset=["lat","lng","population"]).copy()

lats = df2["lat"].to_numpy()
lngs = df2["lng"].to_numpy()
w = df2["population"].to_numpy().astype(float)

# normalize weights
w = w / w.sum()

def weighted_geometric_median(points, weights, eps=1e-7, max_iter=1000):
    """
    points: (n,2) array in some planar coords (x,y)
    weights: (n,) nonnegative weights sum to 1 (or not; scale doesn't matter)
    """
    x = np.average(points, axis=0, weights=weights)  # Start at weighted centroid

    for _ in range(max_iter):
        diff = points - x
        dist = np.linalg.norm(diff, axis=1)

        # Avoid divide by zero
```

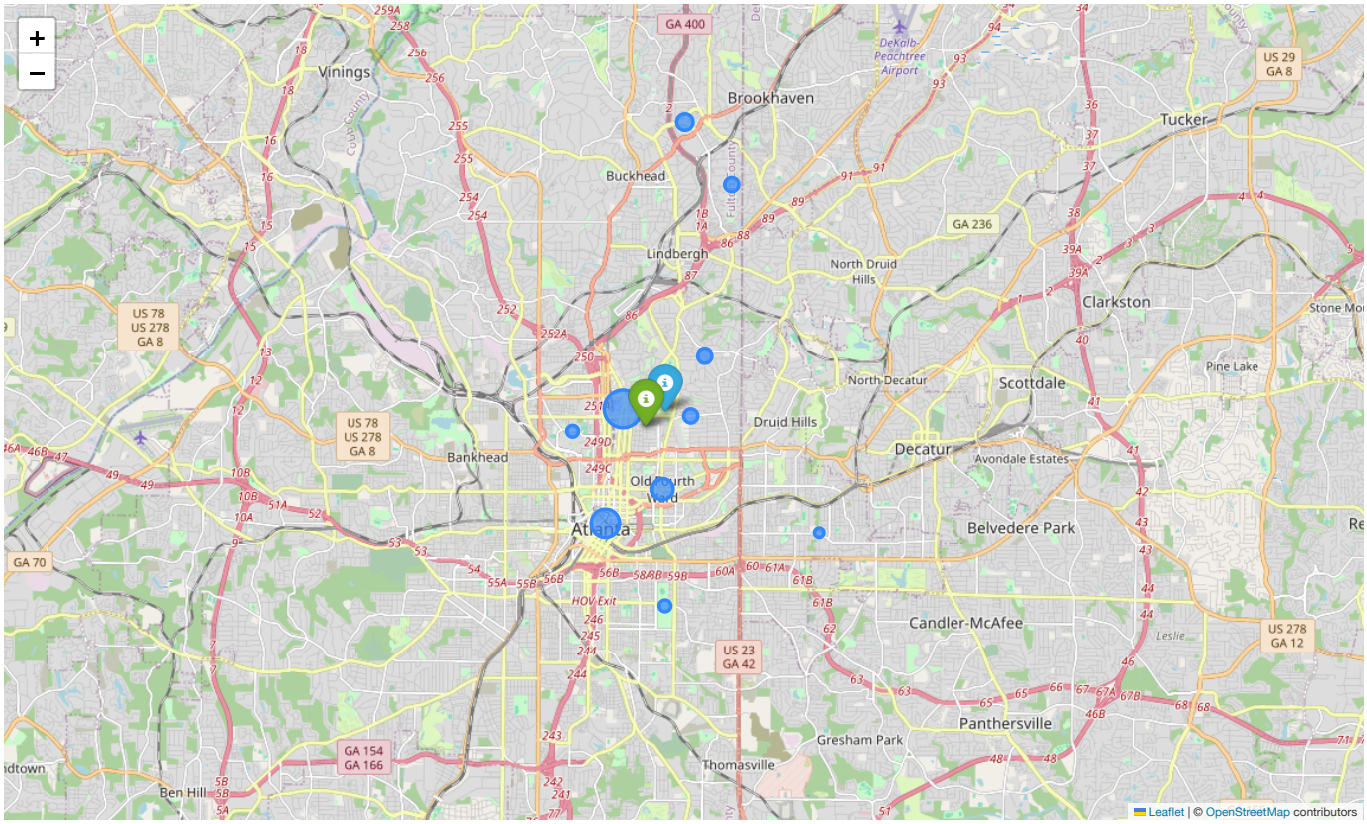
```
        mask = dist > eps
        if not np.any(mask):
            return x

        inv = np.zeros_like(dist)
        inv[mask] = weights[mask] / dist[mask]

        x_new = (points * inv[:, None]).sum(axis=0) / inv.sum()

        if np.linalg.norm(x_new - x) < eps:
            return x_new
        x = x_new

    return x

# Project lat/lng to local planar coords (approx): x = lon*cos(lat0), y = lat
lat0 = np.deg2rad(lats.mean())
x = lngs * np.cos(lat0)
y = lats
pts = np.column_stack([x, y])

# Optimization Execution
gm_xy = weighted_geometric_median(pts, w)
gm_lat = float(gm_xy[1])
gm_lng = float(gm_xy[0] / np.cos(lat0))

# Calculate Total Weighted Haversine Distance
def haversine_km(lat1, lon1, lat2, lon2):
    R = 6371.0
    lat1, lon1, lat2, lon2 = map(np.deg2rad, [lat1, lon1, lat2, lon2])
    dlat, dlon = lat2 - lat1, lon2 - lon1
    a = np.sin(dlat/2)**2 + np.cos(lat1)*np.cos(lat2)*np.sin(dlon/2)**2
    return 2 * R * np.arcsin(np.sqrt(a))

# Results
distances = haversine_km(lats, lngs, gm_lat, gm_lng)
weighted_sum_km = float((distances * df2["population"].to_numpy()).sum())
print(f"Optimal Coordinates (Lat, Lng): ({gm_lat:.6f}, {gm_lng:.6f})")
print(f"Total Weighted Distance Cost:  {weighted_sum_km:,.2f} person-km")
```

```
Optimal Coordinates (Lat, Lng): (33.779529, -84.376797)
Total Weighted Distance Cost:  299,968.98 person-km
```

In [7]:

```
# Comparison of Facility Location Models:
'''
here is the original map, with both IT'S centroid, population weighted, as well as
the theoretical min haversine distance facility location, which will be different
but not unreasonably so. Nor should we EXPECT it to land on a road or nexus, since
its just using great circle distance, but it should be relatively close, since there
are few compounding variables to make the math considerably different.
'''

# Calculate the Population-Weighted Centroid (Simple Weighted Mean)
w_lat = (df["lat"] * df["population"]).sum() / df["population"].sum()
w_lng = (df["lng"] * df["population"]).sum() / df["population"].sum()

# Add Centroid Marker (Blue)
folium.Marker(
    location=[w_lat, w_lng],
    tooltip="Population-weighted centroid",
    popup=f"w_lat={w_lat:.6f}, w_lng={w_lng:.6f}",
    icon=folium.Icon(color="blue")
).add_to(m_weighted)

# Add Haversine Optimal Marker (Green)
folium.Marker(
    location=[gm_lat, gm_lng],
    tooltip="Haversine facility (weighted geometric median)",
    popup=f"gm_lat={gm_lat:.6f}, gm_lng={gm_lng:.6f}",
    icon=folium.Icon(color="green")
).add_to(m_weighted)

# Display the map
m_weighted
```

Out[7]:



Driving Distance and Time Optimization

In [8]:
```
# Data Preparation
df2 = df.dropna(subset=["lat","lng","population"]).copy()
destinations = list(zip(df2["lat"], df2["lng"]))                    # 10 points
weights_raw = df2["population"].to_numpy(dtype=float)
weights = weights_raw / weights_raw.sum()   # normalized weights (sum to 1)

# Grid Generation
def make_grid(center_lat, center_lng, lat_span=0.10, lng_span=0.10, n=15):
    lats = np.linspace(center_lat - lat_span, center_lat + lat_span, n)
    lngs = np.linspace(center_lng - lng_span, center_lng + lng_span, n)
    return [(la, lo) for la in lats for lo in lngs]

candidates = make_grid(gm_lat, gm_lng, lat_span=0.10, lng_span=0.10, n=11) # 225 points reduced to 121 for "coarse pass" t0 not trigger API block
len(candidates)
```

Out[8]: 121

In [9]:
```
# Data Preparation
df2 = df.dropna(subset=["lat", "lng", "population"]).copy()
destinations = list(zip(df2["lat"], df2["lng"]))
weights_raw = df2["population"].to_numpy(dtype=float)
weights = weights_raw / weights_raw.sum()

# Grid Generation (Reduced to 11x11 to optimize API usage)
def make_grid(center_lat, center_lng, lat_span=0.10, lng_span=0.10, n=11):
    lats = np.linspace(center_lat - lat_span, center_lat + lat_span, n)
    lngs = np.linspace(center_lng - lng_span, center_lng + lng_span, n)
    return [(la, lo) for la in lats for lo in lngs]

candidates = make_grid(gm_lat, gm_lng, lat_span=0.10, lng_span=0.10, n=11)

# Fetches both Duration (Time) and Distance (Road length)
def weighted_drive_metrics(origin_latlng, destinations_latlng, weights, mode="driving", pause_s=0.4):
    origin = f"{origin_latlng[0]},{origin_latlng[1]}"
    dest_strs = [f"{d[0]},{d[1]}" for d in destinations_latlng]
    resp = gmaps.distance_matrix(origins=[origin], destinations=dest_strs, mode=mode, units="metric")
    time.sleep(pause_s)

    elems = resp["rows"][0]["elements"]
    secs, meters = [], []
    for el in elems:
        if el.get("status") != "OK":
            return float("inf"), float("inf"), None
        secs.append(el["duration"]["value"])
        meters.append(el["distance"]["value"])

    time_score = float((np.array(secs) * weights).sum())
    dist_score = float((np.array(meters) * weights).sum())
    return time_score, dist_score, np.array(secs)

# Grid Search Execution
best = {"origin": None, "time_score": float("inf"), "dist_score": None}

for i, cand in enumerate(candidates, start=1):
    try:
        # Unpack all 3 return values (use _ for the one we don't need right now)
        t_score, d_score, _ = weighted_drive_metrics(cand, destinations, weights)

        # Use consistent dictionary keys ("time_score" instead of "time")
        if t_score < best["time_score"]:
            best = {"origin": cand, "time_score": t_score, "dist_score": d_score}
            print(f"[{i:3d}/{len(candidates)}] New Best! Coords: ({cand[0]:.4f}, {cand[1]:.4f}) | "
```

```
                f"Time: {t_score/60:5.2f} min | "
                f"Road Dist: {d_score/1000:5.2f} km")
    except Exception as e:
        print(f"Error at point {i}: {e}")
        continue

    if i % 10 == 0:
        time.sleep(2.0)

# Final Results Check
if best["origin"] is not None:
    drive_lat, drive_lng = best["origin"]

    score_t_drive, score_d_drive, _ = weighted_drive_metrics((drive_lat, drive_lng), destinations, weights)
    score_t_hav, score_d_hav, _     = weighted_drive_metrics((gm_lat, gm_lng), destinations, weights)

    print(f"\nMethod 1 (Haversine-Point):")
    print(f"  - Weighted Avg Time:     {score_t_hav/60:.2f} minutes")
    print(f"  - Weighted Avg Distance: {score_d_hav/1000:.2f} km (Road Distance)")

    print(f"\nMethod 2 (Drive-Optimal Best):")
    print(f"  - Coordinates:           {best['origin']}")
    print(f"  - Weighted Avg Time:     {score_t_drive/60:.2f} minutes")
    print(f"  - Weighted Avg Distance: {score_d_drive/1000:.2f} km")
else:
    print("Optimization failed to find a valid origin.")
```

```
[  1/121] New Best! Coords: (33.6795, -84.4768) | Time: 23.12 min | Road Dist: 23.22 km
[  2/121] New Best! Coords: (33.6795, -84.4568) | Time: 21.04 min | Road Dist: 19.53 km
[  3/121] New Best! Coords: (33.6795, -84.4368) | Time: 19.83 min | Road Dist: 17.21 km
[  4/121] New Best! Coords: (33.6795, -84.4168) | Time: 17.36 min | Road Dist: 15.48 km
[  5/121] New Best! Coords: (33.6795, -84.3968) | Time: 15.73 min | Road Dist: 14.57 km
[ 16/121] New Best! Coords: (33.6995, -84.3968) | Time: 14.63 min | Road Dist: 13.65 km
[ 31/121] New Best! Coords: (33.7195, -84.3168) | Time: 13.50 min | Road Dist: 13.77 km
[ 37/121] New Best! Coords: (33.7395, -84.4168) | Time: 12.62 min | Road Dist:  9.86 km
[ 38/121] New Best! Coords: (33.7395, -84.3968) | Time: 11.79 min | Road Dist:  7.84 km
[ 49/121] New Best! Coords: (33.7595, -84.3968) | Time: 11.69 min | Road Dist:  6.24 km
[ 50/121] New Best! Coords: (33.7595, -84.3768) | Time:  9.54 min | Road Dist:  6.01 km
[ 71/121] New Best! Coords: (33.7995, -84.3968) | Time:  8.99 min | Road Dist:  6.83 km

Method 1 (Haversine-Point):
  - Weighted Avg Time:     10.12 minutes
  - Weighted Avg Distance: 4.65 km (Road Distance)

Method 2 (Drive-Optimal Best):
  - Coordinates:           (np.float64(33.799529412768514), np.float64(-84.39679674439644))
  - Weighted Avg Time:     8.99 minutes
  - Weighted Avg Distance: 6.83 km
```

Final Decision Summary & Comparison Insights

For a **parcel pick up facility**, the most significant finding is that the shortest path is not necessarily the fastest.

- **The Haversine location** minimizes the physical road distance (4.65 km), but results in a weighted travel time that is approximately 12.5% slower than Method 2.
- **The Driving-Optimal location** increases the travel distance to 6.83 km but reduces the time to 8.99 minutes. This suggests that this location has superior access to major arteries or highway on-ramps (like I-75/85), allowing users to bypass local traffic lights and congestion in the Midtown core.

> In a logistics context, reducing travel time increases facility throughput and improves the overall user experience for the most populous neighborhoods. Therefore, `Method 2` is the superior choice because it saves an average of 1.13 minutes per trip.

Regarding **spatial shift and accessibility**, the two locations represent fundamentally different strategic implications.

- **The Haversine point** is located in the heart of Midtown (near 8th St NE), which is the mathematical **center of gravity** for the neighborhood coordinates.
- **The Driving point** shifts northwest toward the Loring Heights area. This shift indicates that in a city like Atlanta, the efficiency of the road network—rather than simple proximity—dictates the most accessible location for residents.

In [10]:
```
# Reverse Geocode Addresses
try:
    drive_addr = gmaps.reverse_geocode((drive_lat, drive_lng))[0]["formatted_address"]
    hav_addr   = gmaps.reverse_geocode((gm_lat, gm_lng))[0]["formatted_address"]
except:
    drive_addr, hav_addr = "N/A", "N/A"

# Comparison Table
summary = pd.DataFrame({
    "Method": ["1. Haversine (Midpoint)", "2. Driving (Optimal)"],
    "Address": [hav_addr, drive_addr],
    "Avg Time (min)": [score_t_hav/60, best['time_score']/60],
    "Avg Dist (km)": [score_d_hav/1000, best['dist_score']/1000]
}).round(2)

# 3. Final Print
print(summary.to_string(index=False))
```

```
                Method                                    Address  Avg Time (min)  Avg Dist (km)
1. Haversine (Midpoint)     320 8th St NE, Atlanta, GA 30309, USA           10.12           4.65
   2. Driving (Optimal) 1629 Loring Dr NW, Atlanta, GA 30309, USA            8.99           6.83
```

In [11]:
```
# Final Visualization: Comparison Map
# Add Drive-Time Optimal (Red)
folium.Marker(
    location=[drive_lat, drive_lng],
    tooltip="Method 2: Drive-Time Optimal",
    popup=folium.Popup(
        f"<b>Driving Optimal Best</b><br>"
        f"Address: {drive_addr}<br>"
        f"Avg Time: {best['time_score']/60:.2f} min<br>"
        f"Avg Road Dist: {best['dist_score']/1000:.2f} km",
        max_width=300
    ),
    icon=folium.Icon(color="red", icon="road")
).add_to(m_weighted)

# Add Haversine Point (Green)
folium.Marker(
```

```
    location=[gm_lat, gm_lng],
    tooltip="Method 1: Haversine Optimal",
    popup=folium.Popup(
        f"<b>Haversine Point</b><br>"
        f"Address: {hav_addr}<br>"
        f"Avg Time: {score_t_hav/60:.2f} min<br>"
        f"Avg Road Dist: {score_d_hav/1000:.2f} km",
        max_width=300
    ),
    icon=folium.Icon(color="green", icon="info-sign")
).add_to(m_weighted)

# Display Map
m_weighted
```

Out[11]: