High Performance CUDA Accelerated Local Optimization in Traveling Salesman Problem

Kamil Rocki, PhD
Department of Computer Science
Graduate School of Information Science and Technology
The University of Tokyo
Why TSP?

- It’s hard (NP-hard to be exact)
  - Especially to parallelize
- But easy to understand
  - A basic problem in CS
- TSPLIB - a set of standard problems
  - Allows comparisons of algorithms
- It’s simple to adjust the problem size

Platform for the study of general methods that can be applied to a wide range of discrete optimization problems
Not only salesmen’s headache

• Computer wiring
• Vehicle routing
• Crystallography
• Robot control

plaq5900.tsp

It took 15 years to solve this problem (solved in 2006)
Local/Global Optimization

- Usually the space is huge
- There are many valleys and local minima
- Local search might terminate in one of the local minima
- Global search should find the best solution
Iterated Global Search - Global Optimization (Approximate algorithm)

- Local search is followed by some sort of tour randomization and it’s repeated (until there’s time left)

- Most of the time is spent in the ‘Local Search’ part

Iterated Global Search - Global Optimization (Approximate algorithm)

1: **procedure** ITERATED LOCAL SEARCH
2: $s_0 := \text{GenerateInitialSolution()}$
3: $s^* := 2\text{optLocalSearch}(s_0)$  
   
4: while (termination condition not met)
5: $s' := \text{Perturbation}(s^*)$
6: $s'^* := 2\text{optLocalSearch}(s')$  
   
7: $s^* := \text{AcceptanceCriterion}(s^*, s'^*)$
8: end while
9: end procedure
TSP - Big valley theory (Why does ILS work?)

• Local minima form clusters

• Global minimum is somewhere in the middle

• The better the solution, the closer it is to the global optimum

• ILS worsens the tour only slightly

• Local Search is repeated close to the previous minimum

2-opt local optimization

- Exchange a pair of edges to obtain a better solution if the following condition is true

\[
\text{distance}(B, F) + \text{distance}(G, D) > \text{distance}(B, D) + \text{distance}(G, F)
\]
2-opt local optimization

• In order to find such a pair need to perform:
  \[ \binom{n-2}{2} = (n - 2) \times (n - 3)/2 \text{ checks} \]

• There are some pruning techniques in more complex algorithms

• I analyze all possible pairs
  
• 10000 city problem -> 49.9 Million pairs
  
• 100000 city problem -> 5 Billion pairs

for (int i = 1; i < points - 2; i++)
  for (int j = i + 1; j < points - 1; j++)
  {
    if (distance(i, i-1) + distance(j+1, j) >
         distance(i, j+1) + distance(i-1, j))
      update best i and j;
  }

remove edges (best i, best i-1) and (best j+1, best j)
add edges (best i,best j+1) and (best i-1, best j)
Obtaining a distance

- ‘**Naive**’ approach - calculate it each time it’s needed

- ‘**Smart**’ way - reuse the results

  - For a 100-city problem it is approximately 5 times faster (CPU)

<table>
<thead>
<tr>
<th>Problem (TSPLIB)</th>
<th>Number of cities (points)</th>
<th>Memory needed for LUT (MB)</th>
<th>Memory needed for coordinates (kB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>kroE100</td>
<td>100</td>
<td>0.038</td>
<td>0.78</td>
</tr>
<tr>
<td>ch130</td>
<td>130</td>
<td>0.065</td>
<td>1.02</td>
</tr>
<tr>
<td>ch150</td>
<td>150</td>
<td>0.086</td>
<td>1.17</td>
</tr>
<tr>
<td>kroA200</td>
<td>200</td>
<td>0.15</td>
<td>1.56</td>
</tr>
<tr>
<td>ts225</td>
<td>225</td>
<td>0.19</td>
<td>1.75</td>
</tr>
<tr>
<td>pr299</td>
<td>299</td>
<td>0.34</td>
<td>2.34</td>
</tr>
<tr>
<td>pr439</td>
<td>439</td>
<td>0.74</td>
<td>3.43</td>
</tr>
<tr>
<td>rat783</td>
<td>783</td>
<td>2.34</td>
<td>6.12</td>
</tr>
<tr>
<td>vm1084</td>
<td>1084</td>
<td>4.48</td>
<td>8.47</td>
</tr>
<tr>
<td>pr2392</td>
<td>2392</td>
<td>21.8</td>
<td>18.69</td>
</tr>
<tr>
<td>pcb3038</td>
<td>3038</td>
<td>35.21</td>
<td>23.73</td>
</tr>
<tr>
<td>fl3795</td>
<td>3795</td>
<td>54.9</td>
<td>29.65</td>
</tr>
<tr>
<td>fnl4461</td>
<td>4461</td>
<td>75.9</td>
<td>34.85</td>
</tr>
</tbody>
</table>

\[ n^2 \quad n \]
Obtaining a distance

- **Naive** approach - calculate it each time it’s needed
- **Smart** way - reuse the results
  - For a 100-city problem it is approximately 5 times faster (CPU)
  - For a 100000-city problem it can be slower
- Cache

<table>
<thead>
<tr>
<th>Problem (TSPLIB)</th>
<th>Number of cities (points)</th>
<th>Memory needed for LUT (MB)</th>
<th>Memory needed for coordinates (kB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>kroE100</td>
<td>100</td>
<td>0.038</td>
<td>0.78</td>
</tr>
<tr>
<td>ch130</td>
<td>130</td>
<td>0.065</td>
<td>1.02</td>
</tr>
<tr>
<td>ch150</td>
<td>150</td>
<td>0.086</td>
<td>1.17</td>
</tr>
<tr>
<td>kroA200</td>
<td>200</td>
<td>0.15</td>
<td>1.56</td>
</tr>
<tr>
<td>ts225</td>
<td>225</td>
<td>0.19</td>
<td>1.75</td>
</tr>
<tr>
<td>pr299</td>
<td>299</td>
<td>0.34</td>
<td>2.34</td>
</tr>
<tr>
<td>pr439</td>
<td>439</td>
<td>0.74</td>
<td>3.43</td>
</tr>
<tr>
<td>rat783</td>
<td>783</td>
<td>2.34</td>
<td>6.12</td>
</tr>
<tr>
<td>vm1084</td>
<td>1084</td>
<td>4.48</td>
<td>8.47</td>
</tr>
<tr>
<td>pr2392</td>
<td>2392</td>
<td>21.8</td>
<td>18.69</td>
</tr>
<tr>
<td>pcb3038</td>
<td>3038</td>
<td>35.21</td>
<td>23.73</td>
</tr>
<tr>
<td>fl3795</td>
<td>3795</td>
<td>54.9</td>
<td>29.65</td>
</tr>
<tr>
<td>fnl4461</td>
<td>4461</td>
<td>75.9</td>
<td>34.85</td>
</tr>
</tbody>
</table>

\[ n^2 \quad n \]
Obtaining a distance

- **‘Naive’** approach - calculate it each time it’s needed

- **‘Smart’** way - reuse the results
  - For a 100-city problem it is approximately 5 times faster (CPU)
  - For a 100000-city problem it can be slower
    - Cache
    - With multiple cores, it becomes slower
      - Cache/memory

<table>
<thead>
<tr>
<th>Problem (TSPLIB)</th>
<th>Number of cities (points)</th>
<th>Memory needed for LUT (MB)</th>
<th>Memory needed for coordinates (kB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>kroE100</td>
<td>100</td>
<td>0.038</td>
<td>0.78</td>
</tr>
<tr>
<td>ch130</td>
<td>130</td>
<td>0.065</td>
<td>1.02</td>
</tr>
<tr>
<td>ch150</td>
<td>150</td>
<td>0.086</td>
<td>1.17</td>
</tr>
<tr>
<td>kroA200</td>
<td>200</td>
<td>0.15</td>
<td>1.56</td>
</tr>
<tr>
<td>ts225</td>
<td>225</td>
<td>0.19</td>
<td>1.75</td>
</tr>
<tr>
<td>pr299</td>
<td>299</td>
<td>0.34</td>
<td>2.34</td>
</tr>
<tr>
<td>pr439</td>
<td>439</td>
<td>0.74</td>
<td>3.43</td>
</tr>
<tr>
<td>rat783</td>
<td>783</td>
<td>2.34</td>
<td>6.12</td>
</tr>
<tr>
<td>vm1084</td>
<td>1084</td>
<td>4.48</td>
<td>8.47</td>
</tr>
<tr>
<td>pr2392</td>
<td>2392</td>
<td>21.8</td>
<td>18.69</td>
</tr>
<tr>
<td>pcb3038</td>
<td>3038</td>
<td>35.21</td>
<td>23.73</td>
</tr>
<tr>
<td>fl3795</td>
<td>3795</td>
<td>54.9</td>
<td>29.65</td>
</tr>
<tr>
<td>fnl4461</td>
<td>4461</td>
<td>75.9</td>
<td>34.85</td>
</tr>
</tbody>
</table>

\[ n^2 \quad n \]
LUT (Look Up Table) vs recalculation

- Clever algorithm
- 'Not-so-clever' algorithm

Time vs Number of cores
A change in algorithm design - sequential

• Memory is free, reuse computed results

• Use sophisticated algorithms, prune the search space
A change in algorithm design - parallel

• Memory is free, reuse computed results

• Limited memory (size, throughput), computing is free (almost)

• Use sophisticated algorithms, prune the search space

• Sometimes brute-force search might be faster (especially on GPUs)
  • Avoiding divergence
  • Same amount of time to process 10, 100 even 10000 elements in parallel
  • But -> empty flops
GPU implementation

GPU uses this simple function for EUC_2D problems

```c
__device__ int calculateDistance2D(unsigned int i, unsigned int j, city_coords* coords) { //registers + shared memory
    register float dx, dy;
    dx = coords[i].x - coords[j].x;
    dy = coords[i].y - coords[j].y;
    return (int)(sqrtf(dx * dx + dy * dy) + 0.5f);
} //6 FLOPs + sqrt
```
GPU implementation

Matrix of city pairs to be checked for a possible swap, each pair corresponds to one GPU job

Each thread has to check:

if (distance(i, i-1) + distance(j+1, j) > distance(i, j+1) + distance(i-1, j))
update best i and j;

Example for 32 threads

Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Iteration 6
Iteration 7
Optimizations

- Preorder the coordinates in the route’s order on CPU
  - The reads from the global memory are no longer scattered - much faster
  - 8B needed instead of 10B per city
  - No unnecessary address calculations
  - The problem can be split if it’s too big for the shared memory’s size
GPU implementation

An overview of the algorithm:

1. Copy the **ordered coordinates** to the GPU global memory (CPU)
2. Execute the **kernel**
3. Copy the **ordered coordinates** to the shared memory
4. **Calculate swap** effect of one pair
5. **Find the best value** and store it in the global memory
6. **Read the result** (CPU)
What if coordinates’ size exceeds the size of shared memory?

Let’s assume a big problem
N too big for the shared memory

This part requires $2^m$ float2 elements to be calculated

coordinates range (0,m)
coordinates range (m+1,2m)

coordinates range (N-m+1,N)
coordinates range (N-2m+1,N-m)
What if coordinates’ size exceeds the size of shared memory?

Problem of size N

Assuming local memory of limited size m coordinates

Needed data: two arrays

coordinates range \((m/2+1,m)\)

coordinates range \((N-m/2+1,N)\)

Total size:

\[2 \times 2 \times n \times \text{sizeof(coordinate data type)}\]
What if coordinates’ size exceeds the size of shared memory?

- Arbitrarily big problems can be solved
- Work can be split and run on multiple devices

2 sets of coordinates needed

```c
__device__ int calculateDistance2D(
    unsigned int i,
    unsigned int j,
    city_coords* coordsA,
    city_coords* coordsB) {

    register float dx, dy;

    dx = coordsA[i].x - coordsB[j].x;
    dy = coordsA[i].y - coordsB[j].y;

    return (int)(sqrtf(dx * dx + dy * dy) + 0.5f);
}
```
Results

2-opt step processing time

 Approximately 22 billion comparisons/s

For a 200,000-city problem, 20 \times 10^9 pair need to be checked.
Results

- Time needed for a single local optimization run
- GTX 680
- Compared to Intel i7-3960X CPU
- Up to 300x speedup (single core, no SSE)
- Up to 40x speedup (Intel OpenCL)

<table>
<thead>
<tr>
<th>Problem (TSPLIB)</th>
<th>GPU kernel time full 2-opt</th>
<th>Host to device copy time</th>
<th>Device to host copy time</th>
<th>GPU total time 2-opt</th>
<th>2-opt checks/s (Millions)</th>
<th>Time to first minimum from MF</th>
<th>Initial Length</th>
<th>Optimized Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>berlin52</td>
<td>20 μs</td>
<td>50 μs</td>
<td>11 μs</td>
<td>81 μs</td>
<td>76.56</td>
<td>687 μs</td>
<td>9951</td>
<td>8930</td>
</tr>
<tr>
<td>kroA200</td>
<td>24 μs</td>
<td>50 μs</td>
<td>11 μs</td>
<td>85 μs</td>
<td>1013.78</td>
<td>2.547 ms</td>
<td>34601</td>
<td>31685</td>
</tr>
<tr>
<td>ts225</td>
<td>24 μs</td>
<td>50 μs</td>
<td>11 μs</td>
<td>85 μs</td>
<td>1145.97</td>
<td>1.090 ms</td>
<td>135193</td>
<td>128513</td>
</tr>
<tr>
<td>pr299</td>
<td>26 μs</td>
<td>50 μs</td>
<td>11 μs</td>
<td>87 μs</td>
<td>1878.46</td>
<td>2.555 ms</td>
<td>61493</td>
<td>54895</td>
</tr>
<tr>
<td>pr439</td>
<td>32 μs</td>
<td>50 μs</td>
<td>11 μs</td>
<td>93 μs</td>
<td>3307.85</td>
<td>3.067 ms</td>
<td>124127</td>
<td>115490</td>
</tr>
<tr>
<td>rat783</td>
<td>53 μs</td>
<td>51 μs</td>
<td>11 μs</td>
<td>115 μs</td>
<td>6385.53</td>
<td>8.827 ms</td>
<td>10734</td>
<td>9658</td>
</tr>
<tr>
<td>vm1084</td>
<td>80 μs</td>
<td>51 μs</td>
<td>11 μs</td>
<td>142 μs</td>
<td>8122.51</td>
<td>11.862 ms</td>
<td>287710</td>
<td>267210</td>
</tr>
<tr>
<td>pr2392</td>
<td>299 μs</td>
<td>53 μs</td>
<td>11 μs</td>
<td>363 μs</td>
<td>11065.33</td>
<td>89.577 ms</td>
<td>454068</td>
<td>412085</td>
</tr>
<tr>
<td>pcb3038</td>
<td>471 μs</td>
<td>55 μs</td>
<td>11 μs</td>
<td>547 μs</td>
<td>10689.4</td>
<td>168.02 ms</td>
<td>165688</td>
<td>147690</td>
</tr>
<tr>
<td>ft3795</td>
<td>723 μs</td>
<td>54 μs</td>
<td>11 μs</td>
<td>788 μs</td>
<td>10529.32</td>
<td>256.19 ms</td>
<td>34843</td>
<td>31312</td>
</tr>
<tr>
<td>fnl4461</td>
<td>746 μs</td>
<td>58 μs</td>
<td>11 μs</td>
<td>815 μs</td>
<td>14098.03</td>
<td>327.1 ms</td>
<td>215085</td>
<td>194746</td>
</tr>
<tr>
<td>rl9934</td>
<td>1009 μs</td>
<td>59 μs</td>
<td>11 μs</td>
<td>1079 μs</td>
<td>18172.88</td>
<td>291.09 ms</td>
<td>627417</td>
<td>582958</td>
</tr>
<tr>
<td>pla397</td>
<td>1547 μs</td>
<td>58 μs</td>
<td>11 μs</td>
<td>1616 μs</td>
<td>18135.36</td>
<td>870.2 ms</td>
<td>26954302</td>
<td>24734292</td>
</tr>
<tr>
<td>usa13509</td>
<td>4728 μs</td>
<td>66 μs</td>
<td>11 μs</td>
<td>4805 μs</td>
<td>19481.58</td>
<td>6.5251 s</td>
<td>23271693</td>
<td>20984503</td>
</tr>
<tr>
<td>dl15112</td>
<td>5963 μs</td>
<td>69 μs</td>
<td>11 μs</td>
<td>6043 μs</td>
<td>19272.07</td>
<td>8.6757 s</td>
<td>1810355</td>
<td>1652806</td>
</tr>
<tr>
<td>dl18512</td>
<td>8928 μs</td>
<td>75 μs</td>
<td>11 μs</td>
<td>9014 μs</td>
<td>19268.93</td>
<td>14.975 s</td>
<td>745930</td>
<td>675638</td>
</tr>
<tr>
<td>sw24978</td>
<td>14.99 ms</td>
<td>85 μs</td>
<td>11 μs</td>
<td>15.08 ms</td>
<td>20863.46</td>
<td>37.284 s</td>
<td>1002187</td>
<td>908598</td>
</tr>
<tr>
<td>pla33810</td>
<td>25.81 ms</td>
<td>96 μs</td>
<td>11 μs</td>
<td>26.92 ms</td>
<td>21340.36</td>
<td>68.67 s</td>
<td>76680735</td>
<td>69763154</td>
</tr>
<tr>
<td>pla85900</td>
<td>168.4 ms</td>
<td>205 μs</td>
<td>11 μs</td>
<td>168.6 ms</td>
<td>21915.11</td>
<td>1109.2 s</td>
<td>163831861</td>
<td>149708033</td>
</tr>
<tr>
<td>sra104815</td>
<td>249.5 ms</td>
<td>249 μs</td>
<td>11 μs</td>
<td>249.8 ms</td>
<td>22017.38</td>
<td>2093.4 s</td>
<td>291403</td>
<td>264889</td>
</tr>
<tr>
<td>usa115475</td>
<td>302.3 ms</td>
<td>287 μs</td>
<td>11 μs</td>
<td>302.6 ms</td>
<td>22054.81</td>
<td>3337.8 s</td>
<td>7143419</td>
<td>6492848</td>
</tr>
<tr>
<td>ara238025</td>
<td>1.294 s</td>
<td>740 μs</td>
<td>11 μs</td>
<td>1.2941 s</td>
<td>21902.18</td>
<td>26497 ms</td>
<td>674838</td>
<td>610795</td>
</tr>
<tr>
<td>lra498378</td>
<td>5.639 s</td>
<td>1774 μs</td>
<td>11 μs</td>
<td>5.6388 s</td>
<td>22031.65</td>
<td>3078 ms</td>
<td>2467227</td>
<td>2264810</td>
</tr>
<tr>
<td>lrb744710</td>
<td>12.634 s</td>
<td>2833 μs</td>
<td>11 μs</td>
<td>12.638 s</td>
<td>21947.17</td>
<td>203.8 h</td>
<td>1867266</td>
<td>1692308</td>
</tr>
</tbody>
</table>
Results

1 Kepler GPUs
2 Kepler GPUs
4 Kepler GPUs
8 Kepler GPUs
I have implemented and published LOGO TSP Solver

**LOGO - LOcal GPU Optimization**

CUDA + OpenCL

http://olab.is.s.u-tokyo.ac.jp/~kamil.rocki/projects.html

Linux/Mac: Source code, Windows: Binary with GUI

GPL License

+ More resources about LOGO (slides, papers)
Summary

• Big TSP problems can be solved very quickly on GPU! (The fastest GPU TSP Solver?)

• Sometimes ‘naive’ algorithms might run faster overall in parallel

• Large and very large problems typically can be decomposed into smaller ones and fit fast memory

• Avoid memory accesses (true for both CPUs and GPUs now)

  • Memory is a scarce resource, FLOPs are abundant

  • Exploit locality by manually reordering data (additional 0.1s of preprocessing may save 1s somewhere else)