Rethinking the Parallelization of Random-Restart Hill Climbing
A Case Study in Optimizing a 2-Opt TSP Solver for GPU Execution

Molly A. O’Neil and Martin Burtscher
Department of Computer Science

TEXAS STATE UNIVERSITY
The rising STAR of Texas

Efficient Computing Laboratory
ECL
Overview

- TSP and 2-opt heuristic
- Previous GPU approaches
  - Assign a climber per thread
- Our new approach
  - Assigns a climber per thread block, parallelizes the 2-opt evaluations between threads in a block
  - Several other optimizations
  - Outperforms previous implementations
- Experimental comparison
Traveling Salesman Problem (TSP)

- Combinatorial optimization problem
  - Find minimum-distance Hamiltonian tour in complete, undirected, weighted graph
  - Finding optimal solution is NP-hard
  - Test bed for heuristic approximation approaches

- Application areas
  - Logistics
  - Wire routing
  - Genome analysis
Random-Restart Hill Climbing

- Iterative hill climbing (IHC) local search
  - Generate initial candidate solution
  - Iteratively improve solution via move to neighbor
  - Unlikely to reach global optimum

- Random restart
  - Repeatedly perform IHC from random initial solutions
  - Can require 1,000s to 1,000,000s+ of restarts
  - Each restart (climber) is independent; evaluation of possible moves within each climber also independent
2-Opt Move Evaluation

- Random-restart TSP
  - Generate $k$ random initial tours (city orderings)
  - Iteratively improve tours until local minimum reached
- Tour improvement via application of 2-opt move
  - Remove edges $(i,i+1)$ and $(j,j+1)$ of the tour, reconnect the resulting subtours in the other order by adding edges $(i,j)$ and $(i+1,j+1)$
  - In each IHC step, evaluate all moves and apply best
// city[i] is i^{th} city in tour order
#define dist(a,b) dmat[city[a]][city[b]]
do{
    minchange = 0
    for (i = 0; i < cities-2; i++) {
        for (j = i+2; j < cities; j++) {
            change = dist(i,j) + dist(i+1,j+1) - dist(i,i+1) - dist(j,j+1)
            if (minchange > change) {
                minchange = change
                mini = i, minj = j
            }
        }
    }
    // apply best 2-opt move (mini/minj)
} while (minchange < 0)
2-opt Pseudo Code

// city[i] is i\text{th} city in tour order
#define dist(a,b) dmat[city[a]][city[b]]
do {
    minchange = 0
    for (i = 0; i < cities-2; i++) {
        minchange += dist(i,i+1)
        for (j = i+2; j < cities; j++) {
            change = dist(i,j) + dist(i+1,j+1) - dist(j,j+1)
            if (minchange > change) {
                minchange = change
                mini = i, minj = j
            }
        }
        minchange -= dist(i,i+1)
    }
    // apply best 2-opt move (mini/minj)
} while (minchange < 0)
Experimental Methodology

- **Metric**
  - Throughput in billions of 2-opt moves evaluated per second (Gigamoves/second)

- **System**
  - K40 (Kepler) GPU with 15 SMs and 2880 PEs
  - TACC Maverick node (2x Xeons with 10 cores each)

- **Inputs**
  - First $n$ points of ‘d18512.tsp’ from TSPLIB
  - Select climber count $k$ to fully load SMs
1. Distance Matrix (*matr_s*)

- Our original implementation (2011)
  - Assign a climber (initial random tour) per thread
  - Pre-compute distance matrix in shared memory
  - Each climber needs a tour order array (local memory)

✔ Distance lookups all to shared memory

✖ $O(n^2)$ shared memory requirement (48kB max) limits problem size to 110 cities

✖ Lots of bank conflicts from random matrix accesses
Rethinking the Parallelization of Random-Restart Hill Climbing

Throughput: \textit{matr}_s

- ~26 Gmoves/sec
- Runs out of shared memory
- Limited to problems $\leq$ 110 cities
- Need a more scalable solution
2. Distance Matrix—Global \((matr_g)\)

- Naïve way to remove the shared mem limit...
  - Pre-compute distance matrix in global memory
    - ✔ No more shared memory limit on problem size
    - ✖ Random accesses to large global memory matrix are uncoalesced and uncached in the L1
Rethinking the Parallelization of Random-Restart Hill Climbing
2. Distance Matrix—Global (*matr_g_ro*)

- Naïve way to remove the shared mem limit...
  - Pre-compute distance matrix in global memory
    - ✔ No more shared memory limit on problem size
    - ✖ Random accesses to large global memory matrix are uncoalesced and uncached in the L1

- OK, but distance matrix is read-only...
  - Use *__ldg(*) to force read onto read-only data cache path
    - ✔ High hit rate in the cache at smaller problem sizes
    - ✖ Still random access pattern to \(O(n^2)\) storage
Throughput: $matr_g_ro$

Nearly as good as $matr_s$

Until we exceed RO cache space

And L2 space

Need a \textit{faster} and more scalable solution
3. Distance Re-Calculation (_calc_)

- Published by K. Rocki and R. Suda (2012, 2013)
  - Re-compute distances as needed rather than look up
  - Allows direct permutation of coordinates in tour order (no need for separate array)

✔ _O(n)_ storage allows larger problem sizes (~4000)
✔ Coalesced memory accesses
✖ Limited by local memory size
✖ Large _k_ (≥30720) needed to fully utilize K40 GPU
Pseudo Code Update

// city[i] is i^{th} city in tour order
#define dist(a,b) dmat[city[a]][city[b]]
do {
    minchange = 0
    for (i = 0; i < cities-2; i++) {
        minchange += dist(i,i+1)
        for (j = i+2; j < cities; j++) {
            change = dist(i,j) + dist(i+1,j+1) - dist(j,j+1)
            if (minchange > change) {
                minchange = change
                mini = i, minj = j
            }
        }
    }
    minchange -= dist(i,i+1)
}
// apply best 2-opt move (mini/minj)
}while (minchange < 0)
Pseudo Code Update

// x[i], y[i] are coordinates of i\textsuperscript{th} city in tour order
#define dist(a,b) sqrtf( (x[a]-x[b])^2 + (y[a]-y[b])^2 )
do {
    minchange = 0
    for (i = 0; i < cities-2; i++) {
        minchange += dist(i,i+1)
        for (j = i+2; j < cities; j++) {
            change = dist(i,j) + dist(i+1,j+1) - dist(j,j+1)
            if (minchange > change) {
                minchange = change
                mini = i, minj = j
            }
        }
    }
    minchange -= dist(i,i+1)
}
// apply best 2-opt move (mini/minj)
} while (minchange < 0)

Re-calculate distance rather than index into matrix
Rethinking the Parallelization of Random-Restart Hill Climbing

Throughput: \( \text{calc} \)

- Still need a more scalable solution
- > 20 Gmoves/s
- Local memory limit (\(\leq\)4000 cities)
4. Intra-Parallelization (\textit{intra})

- Hierarchical parallelization of the 2-opt evals
  - Assign a tour per thread block instead of per thread
  - Parallelize 2-opt computation across threads in block
    - Distribute outer i-loop across threads in block (fully parallelized if cities < 1024); inner j-loop sequential
    - Requires reduction + sync to identify best 2-opt move

- ✔ Storage requirement per block reduced
  - Single set of coordinates in tour order
- ✗ Complexity of implementation increases
Pseudo Code Update—Intra

```c
#define dist(a,b) sqrtf((x[a]-x[b])^2 + (y[a]-y[b])^2)

do {
    minchange = 0
    for (i = 0; i < cities-2; i++) {
        minchange += dist(i,i+1)
        for (j = i+2; j < cities; j++) {
            change = dist(i,j) + dist(i+1,j+1) - dist(j,j+1)
            if (minchange > change) {
                minchange = change
                mini = i, minj = j
            }
        }
        minchange -= dist(i,i+1)
    }
    // apply best 2-opt move (mini/minj)
} while (minchange < 0)
```

Rethinking the Parallelization of Random-Restart Hill Climbing
Pseudo Code Update—Intra

```c
#define dist(a,b) sqrtf( (x[a]-x[b])^2 + (y[a]-y[b])^2 )

do {
    minchange = 0
    for (i = threadIdx; i < cities-2; i += blockDim) {
        minchange += dist(i,i+1)
        for (j = i+2; j < cities; j++) {
            change = dist(i,j) + dist(i+1,j+1) - dist(j,j+1)
            if (minchange > change) {
                minchange = change
                mini = i, minj = j
            }
        }
        minchange -= dist(i,i+1)
    }
    __syncthreads()
    // reduction to identify + apply best 2-opt move
} while (minchange < 0)
```

Distribute outer loop to threads in block

Each thread tracks its best move; reduction required to find overall best.
Pseudo Code Update—Intra

```c
#define dist(a,b) sqrtf((x[a]-x[b])^2 + (y[a]-y[b])^2)

do {
    for (i = threadID; i < cities; i += blockDim)
        buf[i] = -dist(i,i+1)
    __syncthreads()

    minchange = 0
    for (i = threadID; i < cities-2; i += blockDim) {
        minchange -= buf[i]
        for (j = i+2; j < cities; j++) {
            change = dist(i,j) + dist(i+1,j+1) + buf[j]
            if (minchange > change) {
                minchange = change
                mini = i, minj = j
            }
        }
    }
    minchange += buf[i]
} __syncthreads()

// reduction to identify + apply best 2-opt move
while (minchange < 0)
```

Pre-compute tour segment lengths

Segment distances read from global memory buffer
Rethinking the Parallelization of Random-Restart Hill Climbing

Throughput: *intra*

On par with *calc*, but not quite there...

No practical limit on input size!

Can we recover the performance loss?
Blocks share ordered tour and buffer space
- Shared mem is small, don’t want to limit problem size

Strip mine the inner j-loop
- Break iterations into chunks s.t. each chunk’s working set fits in shared memory and preload each tile
- But… each thread’s j-loop begins at a different index!
  - Solution: run inner j-loop backwards

✔ Most accesses go to shared memory
✔ No bank conflicts, full coalescing
✖ Implementation complexity increases further
for (j = i+2; j < cities; j++) {
    change = dist(i, j) + dist(i+1, j+1) + buf[j]
    if (minchange > change) {
        minchange = change
        mini = i, minj = j
    }
}
for (j = jj; j >= tileLowerBound; j--) {
    change = dist(i,j) + dist(i+1,j+1)
            + buf[j]
    if (minchange > change) {
        minchange = change
        mini = i, minj = j
    }
}
Pseudo Code Update—Tile

```
parallel_load_tile(x_shmem[], x[])
parallel_load_tile(y_shmem[], y[])
parallel_load_tile(buf_shmem[], buf[])
__syncthreads()

for (j = jj; j >= tileLowerBound; j--) {
    change = shmem_dist(i, j) + shmem_dist(i+1, j+1)
    + shmem_buf[j]
    if (minchange > change) {
        minchange = change
        mini = i, minj = j
    }
}
```
Pseudo Code Update—Tile

for (jj = cities-1; jj >= i+2; jj -= tileSize) {
    parallel_load_tile(x_shmem[], x[])
    parallel_load_tile(y_shmem[], y[])
    parallel_load_tile(buf_shmem[], buf[])
    __syncthreads()

    for (j = jj; j >= tileLowerBound; j--) {
        change = shmem_dist(i,j) + shmem_dist(i+1,j+1)
        + shmem_buf[j]
        if (minchange > change) {
            minchange = change
            mini = i, minj = j
        }
    }
}

J-loop broken into chunks, each pre-loads tile into shared memory
Pseudo Code Update—Tile

for (jj = cities-1; jj >= i+2; jj -= tileSize) {
    parallel_load_tile(x_shmem[], x[])
    parallel_load_tile(y_shmem[], y[])
    parallel_load_tile(buf_shmem[], buf[])
    __syncthreads()

    for (j = jj; j >= tileLowerBound; j--) {
        change = shmem_dist(i,j) + shmem_dist(i+1,j+1)
        + shmem_buf[j]
        if (minchange > change) {
            minchange = change
            mini = i, minj = j
        }
    }
    __syncthreads()
}

Additional synchronization
Throughput: \textit{tile}

- Static launch configuration
- No bank conflicts
- Can we do even better?

Rethinking the Parallelization of Random-Restart Hill Climbing
6. Intra + Tiling + Tuned Launch (*tuned*)

- Tune thread count per block
  - Based on # of cities, shared memory usage, max threads per block and SM, max blocks for SM, and registers per SM
- Launch kernel with computed thread count

✔ Maximizes hardware usage

✖ None (except small CPU code block)
Throughput: tuned

Up to 3X *calc* throughput, 60 Gmoves/s

~50% boost from tuning
Rethinking the Parallelization of Random-Restart Hill Climbing

Throughput: GPU vs. CPU

At peak CPU perf, GPU code 3X faster

GPU code 8X faster on largest inputs

OpenMP on 2 10-core Xeons

Throughput (Gigamoves/s)

Number of Cities

matr_s matr_g matr_g_ro calc intra tile tuned cpu
Conclusions

- CUDA 2-opt TSP solver based on hierarchical parallelization of climbers and move evaluation
  - Uses shared memory without limiting problem size
  - Faster time to first solution
  - Outperforms prior GPU implementations by up to 3X
  - Outperforms OpenMP version on 20 cores by up to 8X

- Another reminder to *rethink* parallelization strategy and optimize code for GPU hardware
Questions?

Acknowledgments

- NSF Graduate Research Fellowship grant 1144466
- NSF grants 1141022, 1217231, 1406304, and 1438963
- REP grant from Texas State University
- Texas Advanced Computing Center (TACC) HPC resources
- Grants and gifts from NVIDIA Corporation