Parallel Local search for the CVRP on the GPU

Christian Schulz, Geir Hasle, Oddvar Kloster, Atle Riise and Morten Smedsrud

SINTEF ICT

28. October 2010

Outline

1. Motivation
2. CVRP & REFs
3. Three-opt on GPU
4. Summary

Motivation

Vehicle Routing Problem

- Still gap between requirements and performance
- Variants of large neighborhood search, variable neighborhood search, iterated local search proven effective

Why parallelize local search

- Local search is an essential part of more advanced strategies such as metaheuristics
- Embarrassingly parallel: Moves independent from each other

$\Rightarrow$ Potential for significant speed up

Why GPU

- High computational power and memory bandwidth
- Cheap

Model

CVRP

- Given: depot & customer nodes, travelling costs, vehicle capacity, customer demands
- Wanted: Feasible route(s) with minimal length

Model

- Based on paper "A Unified Modeling and Solution Framework for Vehicle Routing and Local Search-based Metaheuristics" by Stefan Irnich, INFORMS JOURNAL ON COMPUTING, Vol. 20, No. 2, Spring 2008, pp. 270-287
- Solution represented as a giant tour
- Use of classical resource extension functions to model capacity constraint ⇒ Constant time move evaluation

Classical Resource extension function

- Resource vector $\mathbf{T} \in \mathbb{R}^n$
- Each node has a associated resource interval $[\mathbf{a}_i, \mathbf{b}_i]$
- A classical REF models change in resource from i to j:
  $$\mathbf{f}_{ij}(\mathbf{T}) = \mathbf{T} + \mathbf{t}_{ij} \quad \text{or} \quad \mathbf{f}_{ij}(\mathbf{T}) = \max(\mathbf{a}_j, \mathbf{T} + \mathbf{t}_{ij})$$
- A path is feasible if for each node i there exists a resource vector $\mathbf{T}_i \in [\mathbf{a}_i, \mathbf{b}_i]$ s.th.
  $$\mathbf{f}_{i,i+1}(\mathbf{T}_i) \leq \mathbf{T}_{i+1}$$

Classical Resource extension function

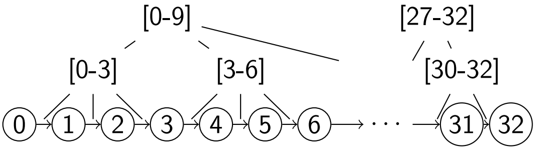
- Resource vector $\mathbf{T} \in \mathbb{R}^n$
- Each node has a associated resource interval $[\mathbf{a}_i, \mathbf{b}_i]$
- A classical REF models change in resource from i to j:
  $$\mathbf{f}_{ij}(\mathbf{T}) = \mathbf{T} + \mathbf{t}_{ij} \quad \text{or} \quad \mathbf{f}_{ij}(\mathbf{T}) = \max(\mathbf{a}_j, \mathbf{T} + \mathbf{t}_{ij})$$
- A path is feasible if for each node i there exists a resource vector $\mathbf{T}_i \in [\mathbf{a}_i, \mathbf{b}_i]$ s.th.
  $$\mathbf{f}_{i,i+1}(\mathbf{T}_i) \leq \mathbf{T}_{i+1}$$

Segment hierarchy $\Rightarrow$ Constant time move evaluation

Aggregation:

[3-6]: $3 \rightarrow 5$, $3 \rightarrow 6$, $4 \rightarrow 6$, inverse

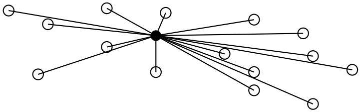
[0-9]: $0 \rightarrow 6$, $0 \rightarrow 9$, $3 \rightarrow 9$, inverse



Method

Initial solution

- Star solution: A single route to each customer



Simple method: Local search with 3-opt move on giant tour

- Remove 3 connections/edges $\Rightarrow$ 4 parts
- Reconnect parts in all possible (new) ways $\Rightarrow$ 7 possibilities

$\Rightarrow$ $(7/6)(n-1)(n-2)(n-3)$ moves (n: #nodes in solution)

What we do on the GPU

- Once
  - Create neighborhood
- Each iteration
  - Create hierarchy
  - Evaluation of capacity constraint and length objective for each move
  - Choosing best move

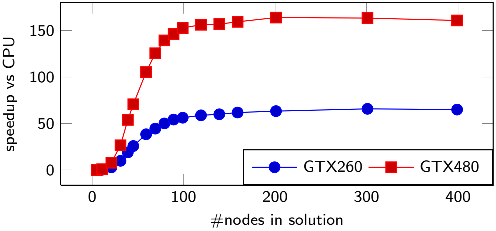
$\Rightarrow$ Neighborhood and hierarchy live whole time on GPU

What we do on the GPU

- Once
  - Create neighborhood
- Each iteration
  - Create hierarchy
  - Evaluation of capacity constraint and length objective for each move
  - Choosing best move

$\Rightarrow$ Neighborhood and hierarchy live whole time on GPU

Both codes not optimized!

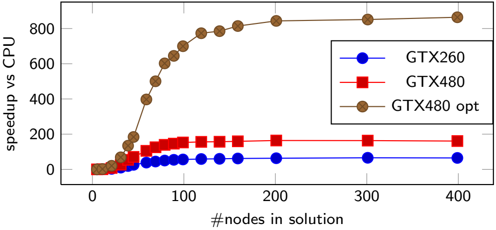


What we do on the GPU

- Once
  - Create neighborhood
- Each iteration
  - Create hierarchy
  - Evaluation of capacity constraint and length objective for each move
  - Choosing best move

⇒ Neighborhood and hierarchy live whole time on GPU

Unfair comparison!
GPU is fast is known
Real task: Efficient usage of GPU hardware



GPU analysis

Look at data for largest available solution (399 nodes)

	Time (ms)	Time % (%)	Bandwidth (Gbyte/sec)	L1 hit (%)	Ipc ≤ 2
First try	1069	42.5	12.2	75.4	0.73
	1410	56.1	33.5	80.8	0.68

GPU analysis

Average number of instructions per cycle on a multiprocessor (Fermi can execute 2 instructions on each multiprocessor)

Implementational approach for criterion/objective evaluation

% of neighborhood evaluation

Hit on L1-cache for global reads

Runtime on GPU

$\leq$ 177.4 Gbyte/sec

	Time (ms)	Time % (%)	Bandwidth (Gbyte/sec)	L1 hit (%)	Ipc ≤ 2
First try	1069	42.5	12.2	75.4	0.73
	1410	56.1	33.5	80.8	0.68

Data for evaluation of objective (tour length)

Data for evaluation of criterion (demands)

GPU analysis

	Time (ms)	Time % (%)	Bandwidth (Gbyte/sec)	L1 hit (%)	Ipc ≤ 2
First try	1069	42.5	12.2	75.4	0.73
	1410	56.1	33.5	80.8	0.68

- Number of registers per thread limited to 32 as compile option
  $\Rightarrow$ Set to 64
- Only 32 threads per block, increase
- Default 16k Cache, change to 48k

GPU analysis

	Time (ms)	Time % (%)	Bandwidth (Gbyte/sec)	L1 hit (%)	Ipc ≤ 2
First try	1069	42.5	12.2	75.4	0.73
	1410	56.1	33.5	80.8	0.68
Max 64 registers,	475	40.8	68.8	86.2	1.64
128 threads, 48k Cache	657	56.3	119.6	93.3	1.39

- Currently use of array for 4 parts in 3-opt
  - $\Rightarrow$ In local memory (slow)
  - $\Rightarrow$ Store in registers

    (Registers per thread: before: 32/39, after: 32/37)

GPU analysis

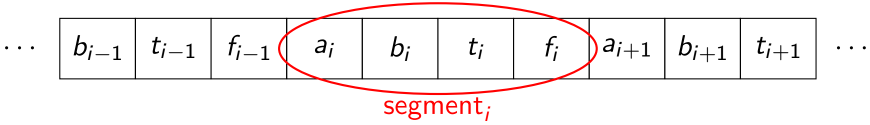
	Time (ms)	Time % (%)	Bandwidth (Gbyte/sec)	L1 hit (%)	Ipc ≤ 2
First try	1069	42.5	12.2	75.4	0.73
	1410	56.1	33.5	80.8	0.68
Max 64 registers,	475	40.8	68.8	86.2	1.64
128 threads, 48k Cache	657	56.3	119.6	93.3	1.39
Parts in registers	479	45.3	24.6	89.2	1.60
	544	51.1	49.6	95.5	1.60

Array of Structures or Structure of Arrays

A hierarchy segment has 4 entries:

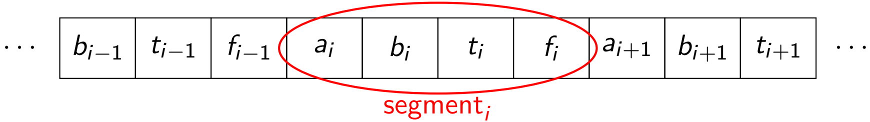
- Interval $[a, b]$
- Cost t
- Feasible information f

Array of Structures



Array of Structures or Structure of Arrays

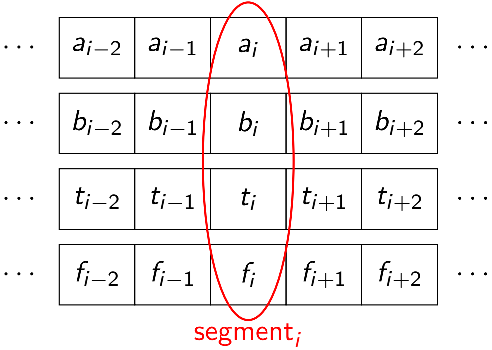
Array of Structures



Structure of Arrays

Normally:

- Neighboring threads access neighboring entries
- Better coalescing
- Fewer transactions
- Faster



GPU analysis

	Time (ms)	Time % (%)	Bandwidth (Gbyte/sec)	L1 hit (%)	Ipc ≤ 2
First try	1069	42.5	12.2	75.4	0.73
	1410	56.1	33.5	80.8	0.68
Max 64 registers,	475	40.8	68.8	86.2	1.64
128 threads, 48k Cache	657	56.3	119.6	93.3	1.39
Parts in registers	479	45.3	24.6	89.2	1.60
	544	51.1	49.6	95.5	1.60
Structure of arrays	479	43.6	24.6	89.2	1.60
	584	53.3	46.7	94.1	1.62

- Most accessed hierarchy segments identical
- All data from a segment needed to compute part
- Array of structure: Data cached!

GPU analysis

	Time (ms)	Time % (%)	Bandwidth (Gbyte/sec)	L1 hit (%)	Ipc ≤ 2
First try	1069	42.5	12.2	75.4	0.73
	1410	56.1	33.5	80.8	0.68
Max 64 registers,	475	40.8	68.8	86.2	1.64
128 threads, 48k Cache	657	56.3	119.6	93.3	1.39
Parts in registers	479	45.3	24.6	89.2	1.60
	544	51.1	49.6	95.5	1.60

- So far: Complicated order to ensure access of neighboring structures (most of the times)
- But: Most accessed hierarchy segments identical, reduced coalescing due to array of structures

$\Rightarrow$ Use simpler order

GPU analysis

	Time (ms)	Time % (%)	Bandwidth (Gbyte/sec)	L1 hit (%)	Ipc ≤ 2
First try	1069	42.5	12.2	75.4	0.73
	1410	56.1	33.5	80.8	0.68
Max 64 registers,	475	40.8	68.8	86.2	1.64
128 threads, 48k Cache	657	56.3	119.6	93.3	1.39
Parts in registers	479	45.3	24.6	89.2	1.60
	544	51.1	49.6	95.5	1.60
Simpler order	295	42.3	38.4	86.6	1.59
(array of structures)	369	53.0	86.2	92.7	1.54

- Modulo operations expensive
- Integer division expensive
- Both can be replaced by bitwise operations for powers of 2

GPU analysis

	Time (ms)	Time % (%)	Bandwidth (Gbyte/sec)	L1 hit (%)	Ipc ≤ 2
First try	1069	42.5	12.2	75.4	0.73
	1410	56.1	33.5	80.8	0.68
Max 64 registers,	475	40.8	68.8	86.2	1.64
128 threads, 48k Cache	657	56.3	119.6	93.3	1.39
Parts in registers	479	45.3	24.6	89.2	1.60
	544	51.1	49.6	95.5	1.60
Simpler order	295	42.3	38.4	86.6	1.59
(array of structures)	369	53.0	86.2	92.7	1.54
Modulo computations	213	39.4	52.8	87.8	1.60
switched to base 2 op.	295	54.5	104.1	93.1	1.52

- So far single precision
- What about double precision

GPU analysis

	Time (ms)	Time % (%)	Bandwidth (Gbyte/sec)	L1 hit (%)	Ipc ≤ 2
First try	1069	42.5	12.2	75.4	0.73
	1410	56.1	33.5	80.8	0.68
Max 64 registers,	475	40.8	68.8	86.2	1.64
128 threads, 48k Cache	657	56.3	119.6	93.3	1.39
Parts in registers	479	45.3	24.6	89.2	1.60
	544	51.1	49.6	95.5	1.60
Simpler order	295	42.3	38.4	86.6	1.59
(array of structures)	369	53.0	86.2	92.7	1.54
Modulo computations	213	39.4	52.8	87.8	1.60
switched to base 2 op.	295	54.5	104.1	93.1	1.52
Double precision	215	38.9	69.3	87.8	1.60
for tour length	295	53.2	104.1	93.1	1.52

Summary & Future Work

Summary

- Local search suited for data parallelism
- Use of GPU can lead to significant speed ups
- Challenge to get full performance of GPU

Future Work

- Larger solutions: memory limit
- More advanced strategies such as metaheuristics
- Keep CPU and GPU busy
- Richer problems

Thank you for your attention!