PHAST - Hardware Accelerated Shortest path Trees

Daniel Delling       Andrew V. Goldberg
Andreas Nowatzyk    Renato F. Werneck

Microsoft Research Silicon Valley

March 25, 2011
Single-Source Shortest Paths

request:

- given a (positively) weighted directed graph \( G = (V, E, w) \) and a source node \( s \)
- compute distances from \( s \) to all other nodes in the graph
- applications: compute many trees for map services (sometimes even all-pairs shortest paths)
Single-Source Shortest Paths

request:
- given a (positively) weighted directed graph \( G = (V, E, w) \) and a source node \( s \)
- compute distances from \( s \) to all other nodes in the graph
- applications: compute many trees for map services (sometimes even all-pairs shortest paths)

solution:
- Dijkstra \([Dij59]\)
Single-Source Shortest Paths

request:
- given a (positively) weighted directed graph $G = (V, E, w)$ and a source node $s$
- compute distances from $s$ to all other nodes in the graph
- applications: compute many trees for map services (sometimes even all-pairs shortest paths)

solution:
- Dijkstra [Dij59]

some facts:
- $O(m + n \log n)$ with Fibonacci Heaps [FT87]
- linear (with a small constant) in practice [Gol01]
- exploiting modern hardware architecture is complicated
Modern CPU architecture

some facts:
- multiple cores
- more cores than memory channels
- hyperthreading
- multi-socket systems
- steep memory hierarchy
- cache coherency
- no register coherency
Modern CPU architecture

some facts:

- multiple cores
- more cores than memory channels
- hyperthreading
- multi-socket systems
- steep memory hierarchy
- cache coherency
- no register coherency

⇒ algorithms need to be tailored
⇒ speedups of 100x possible
GPU Architecture

some facts:

- many cores (up to 512)
- high memory bandwidth (5x faster than CPU)
- but main → GPU memory transfer slow ($\approx 20x$)
- no cache coherency
- Single Instruction Multiple Threads model (thread groups follow same instruction flow)
- barrel processing used to hide DRAM latency
  ⇒ need to keep thousands of independent (!) threads busy
- access of a thread group to memory only efficient for certain patterns
some facts:

- many cores (up to 512)
- high memory bandwidth (5x faster than CPU)
- but main → GPU memory transfer slow ($\approx$ 20x)
- no cache coherency
- Single Instruction Multiple Threads model (thread groups follow same instruction flow)
- barrel processing used to hide DRAM latency ⇒ need to keep thousands of independent (!) threads busy
- access of a thread group to memory only efficient for certain patterns

⇒ algorithms need to be tailored
Parallelizing Dijkstra’s Algorithm

**multiple trees:**
- multi-core by source
- instruction-level parallelism exploitable [Yan10]
- approach **not** applicable for a GPU implementation
  - not enough memory on GPU
  - transfer main → GPU memory too slow
Parallelizing Dijkstra’s Algorithm

**multiple trees:**
- multi-core by source
- instruction-level parallelism exploitable [Yan10]
- approach not applicable for a GPU implementation
  - not enough memory on GPU
  - transfer main → GPU memory too slow

**single tree computation:**
- speculation
- Δ-stepping [MS03],[MBBC09]
- more operations than Dijkstra
- no big speedups on sparse networks
Parallelizing Dijkstra’s Algorithm

**multiple trees:**
- multi-core by source
- instruction-level parallelism exploitable [Yan10]
- approach **not** applicable for a GPU implementation
  - not enough **memory** on GPU
  - transfer main $\rightarrow$ GPU memory too slow

**single tree computation:**
- speculation
- $\Delta$-stepping [MS03],[MBBC09]
- more operations than Dijkstra
- **no** big speedups on sparse networks

**other problem:**
- data locality
  $\Rightarrow$ memory bandwidth bound
experiments:

- input: Western European road network
- 18M nodes, 23M road segments
  - Dijkstra: $\approx 3.0$ s $\implies$ not real-time
  - BFS: $\approx 2.0$ s

numbers refer to a Core-i7 workstation (2.66 GHz)
experiments:

- input: Western European road network
- 18M nodes, 23M road segments
  - Dijkstra: \( \approx 3.0\) s \( \Rightarrow \) not real-time
  - BFS: \( \approx 2.0\) s
  - \( n + m\) clock cycles: \( \approx 15\) ms \( \Rightarrow \) big gap
- gap does not stem from data structures

numbers refer to a Core-i7 workstation (2.66 GHz)
PHAST

experiments:

- input: Western European road network
- 18M nodes, 23M road segments
  - Dijkstra: \(\approx 3.0 \text{ s}\) \(\Rightarrow\) not real-time
  - BFS: \(\approx 2.0 \text{ s}\)
    - \(n + m\) clock cycles: \(\approx 15 \text{ ms}\) \(\Rightarrow\) big gap
- gap does not stem from data structures

numbers refer to a Core-i7 workstation (2.66 GHz)

a new 2-phase algorithm for computing shortest path trees: [DGNW11]

- preprocessing:
  - a few minutes
  - works well in graphs with low highway dimension, e.g., road networks
- faster shortest path tree computation:
  - without optimization as fast as BFS
  - allows to exploit hardware architecture on all levels
    \(\Rightarrow\) up to 3 orders of magnitude faster than Dijkstra
Outline

1. Introduction
2. Contraction Hierarchies
3. PHAST
4. Parallelization
5. GPU Implementation
6. Conclusion
Contraction Hierarchies: A 2-phase algorithm for exact route planning

preprocessing:

- order nodes by importance (heuristic)
- process in order
- add shortcuts to preserve distances between more important nodes
- assign levels (ca. 150 in road networks)

≈ 5 minutes, 75% increase in number of edges

heavily relies on the metric (assumes a strong hierarchy)
Contraction Hierarchies: A 2-phase algorithm for exact route planning

preprocessing:
- order nodes by importance (heuristic)
Contraction Hierarchies: A 2-phase algorithm for exact route planning

preprocessing:
- order nodes by importance (heuristic)
- process in order
- add shortcuts to preserve distances between more important nodes
**Contraction Hierarchies:** A 2-phase algorithm for exact route planning

**preprocessing:**
- order nodes by importance (heuristic)
- process in order
- add shortcuts to preserve distances between more important nodes
Contraction Hierarchies: A 2-phase algorithm for exact route planning

preprocessing:
- order nodes by importance (heuristic)
- process in order
- add shortcuts to preserve distances between more important nodes
preprocessing:
- order nodes by importance (heuristic)
- process in order
- add *shortcuts* to preserve distances between more important nodes
Contraction Hierarchies: A 2-phase algorithm for exact route planning

preprocessing:
- order nodes by importance (heuristic)
- process in order
- add shortcuts to preserve distances between more important nodes
Contraction Hierarchies: A 2-phase algorithm for exact route planning

preprocessing:
- order nodes by importance (heuristic)
- process in order
- add shortcuts to preserve distances between more important nodes
Contraction Hierarchies: A 2-phase algorithm for exact route planning

preprocessing:
- order nodes by importance (heuristic)
- process in order
- add shortcuts to preserve distances between more important nodes
Contraction Hierarchies: A 2-phase algorithm for exact route planning

preprocessing:
- order nodes by importance (heuristic)
- process in order
- add shortcuts to preserve distances between more important nodes
- assign levels (ca. 150 in road networks)

preprocessing: order nodes by importance (heuristic)
process in order
add shortcuts to preserve distances between more important nodes
assign levels (ca. 150 in road networks)
Contraction Hierarchies: A 2-phase algorithm for exact route planning

preprocessing:
- order nodes by importance (heuristic)
- process in order
- add shortcuts to preserve distances between more important nodes
- assign levels (ca. 150 in road networks)
- \( \approx 5 \) minutes, 75% increase in number of edges
- heavily relies on the metric (assumes a strong hierarchy)
point-to-point query

- modified bidirectional Dijkstra
- only follow edges to more important nodes
point-to-point query
- modified bidirectional Dijkstra
- only follow edges to more important nodes
Contraction Hierarchies: A 2-phase algorithm for exact route planning

**point-to-point query**
- modified *bidirectional* Dijkstra
- only follow edges to *more important* nodes
point-to-point query

- modified bidirectional Dijkstra
- only follow edges to more important nodes
**point-to-point query**
- modified *bidirectional* Dijkstra
- only follow edges to *more important* nodes
Contraction Hierarchies: A 2-phase algorithm for exact route planning

point-to-point query
- modified bidirectional Dijkstra
- only follow edges to more important nodes

good performance on road networks:
- each upward search scans about 500 nodes
- 10000x faster than bidirectional Dijkstra (point-to-point)
Outline

1. Introduction
2. Contraction Hierarchies
3. PHAST
4. Parallelization
5. GPU Implementation
6. Conclusion
one-to-all search from source $s$: 

```
run CH forward search from $s$ (≈ 0.05 ms)
set distance labels $d$ of reached nodes
process all nodes $u$ in reverse level order:
  ▶ check incoming arcs $(v, u)$ with $\text{lev}(v) > \text{lev}(u)$
  ▶ set $d(u) = \min\{d(u), d(v) + w(v, u)\}$
top-down processing without priority queue (ca. 2.0 s)
```
one-to-all search from source s:

- run CH forward search from s (≈ 0.05 ms)
Replacing Dijkstra

**one-to-all search from source \( s \):**
- run CH forward search from \( s \) (≈ 0.05 ms)
- set distance labels \( d \) of reached nodes
one-to-all search from source $s$:
- run CH forward search from $s$ ($\approx 0.05$ ms)
- set distance labels $d$ of reached nodes
- process all nodes $u$ in reverse level order:
  - check incoming arcs $(v, u)$ with $\text{lev}(v) > \text{lev}(u)$
  - set $d(u) = \min\{d(u), d(v) + w(v, u)\}$
one-to-all search from source \( s \):  
- run CH \textit{forward} search from \( s \) (\( \approx 0.05 \) ms)  
- set distance labels \( d \) of reached nodes  
- process all nodes \( u \) in \textit{reverse} level order:  
  - check incoming arcs \((v, u)\) with \( \text{lev}(v) > \text{lev}(u) \)  
  - set \( d(u) = \min\{d(u), d(v) + w(v, u)\} \)
Replacing Dijkstra

one-to-all search from source $s$:
- run CH forward search from $s$ ($\approx 0.05$ ms)
- set distance labels $d$ of reached nodes
- process all nodes $u$ in reverse level order:
  - check incoming arcs $(v, u)$ with $\text{lev}(v) > \text{lev}(u)$
  - set $d(u) = \min\{d(u), d(v) + w(v, u)\}$
one-to-all search from source $s$:

- run CH forward search from $s$ ($\approx 0.05$ ms)
- set distance labels $d$ of reached nodes
- process all nodes $u$ in reverse level order:
  - check incoming arcs $(v, u)$ with $\text{lev}(v) > \text{lev}(u)$
  - set $d(u) = \min\{d(u), d(v) + w(v, u)\}$
Replacing Dijkstra

one-to-all search from source $s$:

- run CH forward search from $s$ ($\approx 0.05$ ms)
- set distance labels $d$ of reached nodes
- process all nodes $u$ in reverse level order:
  - check incoming arcs $(v, u)$ with $\text{lev}(v) > \text{lev}(u)$
  - set $d(u) = \min\{d(u), d(v) + w(v, u)\}$
Replacing Dijkstra

one-to-all search from source $s$:

- run CH forward search from $s$ ($\approx 0.05$ ms)
- set distance labels $d$ of reached nodes
- process all nodes $u$ in reverse level order:
  - check incoming arcs $(v, u)$ with $\text{lev}(v) > \text{lev}(u)$
  - set $d(u) = \min\{d(u), d(v) + w(v, u)\}$

![Graph diagram showing level and connections between nodes with distances and level labels.](attachment://graph_diagram.png)
one-to-all search from source $s$:

- run CH forward search from $s$ (≈ 0.05 ms)
- set distance labels $d$ of reached nodes
- process all nodes $u$ in reverse level order:
  - check incoming arcs $(v, u)$ with $\text{lev}(v) > \text{lev}(u)$
  - set $d(u) = \min\{d(u), d(v) + w(v, u)\}$
one-to-all search from source $s$:

- run CH forward search from $s$ ($\approx 0.05$ ms)
- set distance labels $d$ of reached nodes
- process all nodes $u$ in reverse level order:
  - check incoming arcs $(v, u)$ with $lev(v) > lev(u)$
  - set $d(u) = \min\{d(u), d(v) + w(v, u)\}$
- top-down processing without priority queue (ca. 2.0 s)
Analysis

observation:

- top-down process is the bottleneck
Analysis

observation:

- top-down process is the **bottleneck**
- access to the data is **inefficient**

dist(u):
observation:

- top-down process is the **bottleneck**
- access to the data is **inefficient**

![Diagram showing levels and access points](image-url)
observation:

- top-down process is the bottleneck
- access to the data is inefficient
observation:
- top-down process is the bottleneck
- access to the data is inefficient
analysis:
- top-down process is the bottleneck
- access to the data is inefficient
observation:
- top-down process is the bottleneck
- access to the data is inefficient
observation:
- top-down process is the bottleneck
- access to the data is inefficient
observation:

- top-down process is the bottleneck
- access to the data is inefficient
- processing order is independent of the source node
Analysis

observation:
- top-down process is the bottleneck
- access to the data is inefficient
- processing order is independent of the source node

idea:
- reorder nodes, arcs, distance labels by level
  \[\Rightarrow\] reading arcs and writing distances become a sequential sweep

\(\text{dist}(u):\)
observation:
- top-down process is the bottleneck
- access to the data is inefficient
- processing order is independent of the source node

idea:
- reorder nodes, arcs, distance labels by level
  reading arcs and writing distances become a sequential sweep
Analysis

observation:
- top-down process is the bottleneck
- access to the data is inefficient
- processing order is independent of the source node

idea:
- reorder nodes, arcs, distance labels by level

⇒ reading arcs and writing distances become a sequential sweep
observation:
- top-down process is the bottleneck
- access to the data is inefficient
- processing order is independent of the source node

idea:
- reorder nodes, arcs, distance labels by level

⇒ reading arcs and writing distances become a sequential sweep
Analysis

**observation:**
- top-down process is the **bottleneck**
- access to the data is **inefficient**
- processing order is **independent** of the source node

**idea:**
- **reorder** nodes, arcs, distance labels by level
  ⇒ reading arcs and writing distances become a sequential sweep
**Analysis**

**observation:**
- top-down process is the **bottleneck**
- access to the data is **inefficient**
- processing order is **independent** of the source node

**idea:**
- **reorder** nodes, arcs, distance labels by level
  - reading arcs and writing distances become a sequential sweep

\[ \text{dist}(u) : \]
analysis:

- top-down process is the bottleneck
- access to the data is inefficient
- processing order is independent of the source node

idea:

- reorder nodes, arcs, distance labels by level

⇒ reading arcs and writing distances become a sequential sweep
observation:

- top-down process is the bottleneck
- access to the data is inefficient
- processing order is independent of the source node

idea:

- reorder nodes, arcs, distance labels by level
  ⇒ reading arcs and writing distances become a sequential sweep
  ⇒ 172 ms per tree

dist(u):
Analysis

observation:
- top-down process is the bottleneck
- access to the data is inefficient
- processing order is independent of the source node

idea:
- reorder nodes, arcs, distance labels by level
  ⇒ reading arcs and writing distances become a sequential sweep
  ⇒ 172 ms per tree
- but reading distances still inefficient
Scenario: Multiple Sources

run k forward searches one sweep (update all k values)

align distance labels per node

96.8 ms per tree (k = 16)

SSE: 128-bit registers

basic operations (min, add) on four 32-bit integers in parallel

scan 4 sources at once

37.1 ms per tree (k = 16)
Scenario: Multiple Sources

idea:

![Diagram showing shortest path trees with multiple sources](image-url)
Scenario: Multiple Sources

idea:

- run $k$ forward searches

![Diagram](image_url)
idea:
- run \( k \) forward searches
idea:
- run $k$ forward searches
idea:

- run $k$ forward searches
- one sweep (update all $k$ values)
- align distance labels per node
Scenario: Multiple Sources

idea:

- run $k$ forward searches
- one sweep (update all $k$ values)
- align distance labels per node
idea:

- run $k$ forward searches
- one sweep (update all $k$ values)
- align distance labels per node

---

Diagram with labels:

Levels:
- Level 0: 5
- Level 1: 10
- Level 2: 3
- Level 3: 7

Nodes:
- Node 1
- Node 2
- Node 3
- Node 4
- Node 5
- Node 6
- Node 7

Edges:
- Edge 5
- Edge 3
- Edge 2
- Edge 4
- Edge 4
- Edge 5
idea:

- run $k$ forward searches
- one sweep (update all $k$ values)
- align distance labels per node
idea:

- run $k$ forward searches
- one sweep (update all $k$ values)
- align distance labels per node
idea:
- run $k$ forward searches
- one sweep (update all $k$ values)
- align distance labels per node
Scenario: Multiple Sources

idea:
- run $k$ forward searches
- one sweep (update all $k$ values)
- align distance labels per node
idea:
- run $k$ forward searches
- one sweep (update all $k$ values)
- align distance labels per node
- 96.8 ms per tree ($k = 16$)
Scenario: Multiple Sources

idea:
- run $k$ forward searches
- one sweep (update all $k$ values)
- align distance labels per node
- 96.8 ms per tree ($k = 16$)

SSE:
- 128-bit registers
- basic operations (min, add) on four 32-bit integers in parallel
- scan 4 sources at once
Scenario: Multiple Sources

idea:
- run $k$ forward searches
- one sweep (update all $k$ values)
- align distance labels per node
- 96.8 ms per tree ($k = 16$)

SSE:
- 128-bit registers
- basic operations (min, add) on four 32-bit integers in parallel
- scan 4 sources at once
- 37.1 ms per tree ($k = 16$)
Outline

1. Introduction

2. Contraction Hierarchies

3. PHAST

4. Parallelization

5. GPU Implementation

6. Conclusion
Parallelization

obvious way of parallelization

- by sources

\[ \text{dist}(s_1,u): \]

\[ \text{dist}(s_2,u): \]

\[ \text{dist}(s_3,u): \]

\[ \text{dist}(s_4,u): \]

results:
16 sources per sweep (updating via SSE)

multi-core by source nodes
\[ \Rightarrow \]
64 sources in parallel (4 cores)

why no perfect speedup?
lower bound tests indicate that we are close to memory bandwidth barrier
can a GPU help?
Parallelization

obvious way of parallelization

- by sources

\[
\text{dist}(s_1,u) : \quad \text{dist}(s_2,u) : \quad \text{dist}(s_3,u) : \quad \text{dist}(s_4,u):
\]

results:

- 16 sources per sweep (updating via SSE)
- multi-core by source nodes
  \Rightarrow 64 sources in parallel (4 cores)
- 18.8 ms per tree on average
Parallelization

obvious way of parallelization

- by sources

results:

- 16 sources per sweep (updating via SSE)
- multi-core by source nodes
  $\Rightarrow$ 64 sources in parallel (4 cores)
- 18.8 ms per tree on average
- why no perfect speedup?
obvious way of parallelization

- by sources

results:

- 16 sources per sweep (updating via SSE)
- multi-core by source nodes

⇒ 64 sources in parallel (4 cores)
- 18.8 ms per tree on average
- why no perfect speedup?
- lower bound tests indicate that we are close to memory bandwidth barrier
Parallelization

obvious way of parallelization

- by sources

results:

- 16 sources per sweep (updating via SSE)
- multi-core by source nodes

⇒ 64 sources in parallel (4 cores)
- 18.8 ms per tree on average
- why no perfect speedup?
- lower bound tests indicate that we are close to memory bandwidth barrier
- can a GPU help?
1 Introduction

2 Contraction Hierarchies

3 PHAST

4 Parallelization

5 GPU Implementation

6 Conclusion
Intel Xeon X5680:
- 3.33 GHz
- 32 GB/s memory bandwidth
- 6 cores

NVIDIA GTX 580:
- 772 MHz, 1.5 GB RAM
- 192 GB/s memory bandwidth
- 16 cores, 32 parallel threads (a warp) per core ⇒ 512 threads in parallel
observation:
- upward search is fast
- bottleneck is the linear sweep
- limited by memory bandwidth
observation:
- upward search is fast
- bottleneck is the linear sweep
- limited by memory bandwidth

idea:
- run upward search on the CPU
- copy search space to GPU (less than 2 kB)
- do linear sweep on the GPU
GPHAST - Basic Ideas

observation:
- upward search is fast
- bottleneck is the linear sweep
- limited by memory bandwidth

idea:
- run upward search on the CPU
- copy search space to GPU (less than 2 kB)
- do linear sweep on the GPU

problem:
- not enough memory on GPU to compute thousands of trees in parallel
- we need to parallelize a single tree computation
Parallel Linear Sweep

observation:

- when scanning level $i$:
  - only incoming arcs from level $> i$ are relevant
  - writing distance labels in level $i$, read from level $> i$
  - distance labels for level $> i$ are correct

- scanning a level-$i$ node is independent from other level-$i$ nodes

\[
\text{dist}(u): \quad \text{W}
\]
Parallel Linear Sweep

**observation:**

- when scanning level $i$:
  - only incoming arcs from level $> i$ are relevant
  - writing distance labels in level $i$, read from level $> i$
  - distance labels for level $> i$ are correct

- scanning a level-$i$ node is *independent* from other level-$i$ nodes

**results:**

5.5 ms on an NVIDIA GTX 480
511 speedup over Dijkstra (multiple trees: 2.2 ms)
Parallel Linear Sweep

observation:

- when scanning level $i$:
  - only incoming arcs from level $> i$ are relevant
  - writing distance labels in level $i$, read from level $> i$
  - distance labels for level $> i$ are correct

- scanning a level-$i$ node is independent from other level-$i$ nodes
Parallel Linear Sweep

observation:

- when scanning level $i$:
  - only incoming arcs from level $> i$ are relevant
  - writing distance labels in level $i$, read from level $> i$
  - distance labels for level $> i$ are correct

- scanning a level-$i$ node is independent from other level-$i$ nodes
observation:

- when scanning level $i$:
  - only incoming arcs from level $> i$ are relevant
  - writing distance labels in level $i$, read from level $> i$
  - distance labels for level $> i$ are correct
- scanning a level-$i$ node is independent from other level-$i$ nodes

idea:

- scan all nodes on level $i$ in parallel
- synchronization after each level
- one thread per node

\[
\text{dist}(u): r_4 r_1 r_2 r_3 \quad W_1 \quad W_2 \quad W_3 \quad W_4
\]
observation:
- when scanning level \(i\):  
  - only incoming arcs from level \(i > i\) are relevant  
  - writing distance labels in level \(i\), read from level \(i > i\)  
  - distance labels for level \(i > i\) are correct
- scanning a level-\(i\) node is independent from other level-\(i\) nodes

idea:
- scan all nodes on level \(i\) in parallel  
- synchronization after each level  
- one thread per node

\[
\text{dist}(u): \overline{\overline{\overline{r_3}} \overline{r_1} \overline{r_4} \overline{r_2}} \overline{W_1} \overline{W_2} \overline{W_3} \overline{W_4}
\]
observation:

- when scanning level $i$:
  - only incoming arcs from level $i > i$ are relevant
  - writing distance labels in level $i$, read from level $i > i$
  - distance labels for level $i > i$ are correct
- scanning a level-$i$ node is independent from other level-$i$ nodes

idea:

- scan all nodes on level $i$ in parallel
- synchronization after each level
- one thread per node

results:

- 5.5 ms on an NVIDIA GTX 480
- 511 speedup over Dijkstra

$$\text{dist}(u): r_1 \rightarrow r_4 \rightarrow r_3 \rightarrow r_2 \rightarrow w_1 \rightarrow w_2 \rightarrow w_3 \rightarrow w_4$$
observation:
- when scanning level $i$:
  - only incoming arcs from level $i > i$ are relevant
  - writing distance labels in level $i$, read from level $i > i$
  - distance labels for level $i > i$ are correct
- scanning a level-$i$ node is independent from other level-$i$ nodes

idea:
- scan all nodes on level $i$ in parallel
- synchronization after each level
- one thread per node

results:
- 5.5 ms on an NVIDIA GTX 480
- 511 speedup over Dijkstra
- (multiple trees: 2.2 ms)
### All-Pairs Shortest Paths

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Device</th>
<th>Time</th>
<th>Energy [MJ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dijkstra</td>
<td>4-core workstation</td>
<td>197d</td>
<td></td>
</tr>
<tr>
<td></td>
<td>12-core server</td>
<td>60d</td>
<td></td>
</tr>
<tr>
<td></td>
<td>48-core server</td>
<td>35d</td>
<td></td>
</tr>
<tr>
<td>PHAST</td>
<td>4-core workstation</td>
<td>94h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>12-core server</td>
<td>36h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>48-core server</td>
<td>20h</td>
<td></td>
</tr>
<tr>
<td>GPHAST</td>
<td>GTX 580</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## All-Pairs Shortest Paths

<table>
<thead>
<tr>
<th>algorithm</th>
<th>device</th>
<th>time</th>
<th>energy [MJ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dijkstra</td>
<td>4-core workstation</td>
<td>197d</td>
<td></td>
</tr>
<tr>
<td></td>
<td>12-core server</td>
<td>60d</td>
<td></td>
</tr>
<tr>
<td></td>
<td>48-core server</td>
<td>35d</td>
<td></td>
</tr>
<tr>
<td>PHAST</td>
<td>4-core workstation</td>
<td>94h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>12-core server</td>
<td>36h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>48-core server</td>
<td>20h</td>
<td></td>
</tr>
<tr>
<td>GPHAST</td>
<td>GTX 580</td>
<td>11h</td>
<td></td>
</tr>
</tbody>
</table>
### All-Pairs Shortest Paths

<table>
<thead>
<tr>
<th>algorithm</th>
<th>device</th>
<th>time</th>
<th>energy [MJ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dijkstra</td>
<td>4-core workstation</td>
<td>197d</td>
<td></td>
</tr>
<tr>
<td></td>
<td>12-core server</td>
<td>60d</td>
<td></td>
</tr>
<tr>
<td></td>
<td>48-core server</td>
<td>35d</td>
<td></td>
</tr>
<tr>
<td>PHAST</td>
<td>4-core workstation</td>
<td>94h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>12-core server</td>
<td>36h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>48-core server</td>
<td>20h</td>
<td></td>
</tr>
<tr>
<td>GPHAST</td>
<td>GTX 580</td>
<td>11h</td>
<td></td>
</tr>
</tbody>
</table>

4-core workstation without GPU: 163 watts
4-core workstation with GPU: 375 watts
12-core server: 332 watts
48-core server: 747 watts
## All-Pairs Shortest Paths

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Device</th>
<th>Time</th>
<th>Energy [MJ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dijkstra</td>
<td>4-core workstation</td>
<td>197d</td>
<td>2780.6</td>
</tr>
<tr>
<td></td>
<td>12-core server</td>
<td>60d</td>
<td>1725.9</td>
</tr>
<tr>
<td></td>
<td>48-core server</td>
<td>35d</td>
<td>2265.5</td>
</tr>
<tr>
<td>PHAST</td>
<td>4-core workstation</td>
<td>94h</td>
<td>55.2</td>
</tr>
<tr>
<td></td>
<td>12-core server</td>
<td>36h</td>
<td>43.0</td>
</tr>
<tr>
<td></td>
<td>48-core server</td>
<td>20h</td>
<td>54.2</td>
</tr>
<tr>
<td>GPHAST</td>
<td>GTX 580</td>
<td>11h</td>
<td>14.9</td>
</tr>
</tbody>
</table>

- 4-core workstation without GPU: 163 watts
- 4-core workstation with GPU: 375 watts
- 12-core server: 332 watts
- 48-core server: 747 watts
Outline

1 Introduction
2 Contraction Hierarchies
3 PHAST
4 Parallelization
5 GPU Implementation
6 Conclusion
Conclusion

Summary:

- One tree on a GPU: 5.5 ms (about 0.31 ns per entry)
- Real-time computation of shortest path trees
- 16 trees on a GPU at once: 2.2 ms per tree (about 0.13 ns per entry)
- APSP in 11 hours (on a workstation with one GPU), instead of half a year (on 4 cores)
- APSP-based computation becomes practical
- 150 times more energy-efficient than Dijkstra’s algorithm

Thank you for your attention!
Conclusion

**summary:**
- one tree on a GPU: **5.5 ms** (about **0.31 ns per entry**)
- real-time computation of shortest path trees
- 16 trees on a GPU at once: **2.2 ms per tree** (about **0.13 ns per entry**)
- APSP in **11 hours** (on a workstation with one GPU), instead of half a year (on 4 cores)
- APSP-based computation becomes **practical**
- **150** times more energy-efficient than Dijkstra’s algorithm

**other recent results:**
- point-to-point shortest paths with a few memory accesses
- refinement of highway dimension
- graph partitioning
- fully realistic driving directions

Thank you for your attention!
Conclusion

summary:
- one tree on a GPU: 5.5 ms (about 0.31 ns per entry)
- real-time computation of shortest path trees
- 16 trees on a GPU at once: 2.2 ms per tree (about 0.13 ns per entry)
- APSP in 11 hours (on a workstation with one GPU), instead of half a year (on 4 cores)
- APSP-based computation becomes practical
- 150 times more energy-efficient than Dijkstra’s algorithm

other recent results:
- point-to-point shortest paths with a few memory accesses
- refinement of highway dimension
- graph partitioning
- fully realistic driving directions

Thank you for your attention!
Appendix
1. natural cut detection
   - pick a random center
   - use BFS to define a core and a ring
   - find minimum cut between them
   - repeat multiple times
1. natural cut detection
   - pick a random center
   - use BFS to define a core and a ring
   - find minimum cut between them
   - repeat multiple times

2. contraction
   - keep only edges that appeared in some cut
   - contract the rest into fragments
   - reduces graph by several orders of magnitude
   - preserves natural cuts between dense regions
     (e.g., bridges, national borders, mountain passes...)

[DGWR11]
Graph Partitioning II: Assembly

1. run greedy algorithm
   - join well-connected fragments
   - find maximal solution

2. run local search
   - reoptimize pairs of adjacent cells
   - fragments can move to neighboring cells

3. enhanced optimizations (optional)
   - multistart, recombination, branch-and-bound

⇒ yields best known solutions for road networks
Case Study: Point-to-Point Shortest Paths

two phase approach:
- preprocess network to compute auxillary data
- use data to speed up queries
- three-criteria optimization (preprocessing time, space, query times)
Case Study: Point-to-Point Shortest Paths

two phase approach:
- preprocess network to compute auxiliary data
- use data to speed up queries
- three-criteria optimization (preprocessing time, space, query times)

<table>
<thead>
<tr>
<th>method</th>
<th>preprocessing time [h:m]</th>
<th>preprocessing space [GB]</th>
<th>query time [μs]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reach</td>
<td>0:15</td>
<td>1.5</td>
<td>1253.5</td>
</tr>
<tr>
<td>CH</td>
<td>0:05</td>
<td>0.4</td>
<td>93.5</td>
</tr>
<tr>
<td>TNR</td>
<td>1:52</td>
<td>3.7</td>
<td>1.8</td>
</tr>
<tr>
<td>Table Lookup</td>
<td>&gt; 11:03</td>
<td>1 208 358.7</td>
<td>0.056</td>
</tr>
</tbody>
</table>

observation:
- excellent performance in practice
- used in production
- prime example for algorithm engineering
- but for a long time: no theoretical justification
Case Study: Point-to-Point Shortest Paths

two phase approach:

- preprocess network to compute auxiliary data
- use data to speed up queries
- three-criteria optimization (preprocessing time, space, query times)

<table>
<thead>
<tr>
<th>method</th>
<th>preprocessing time [h:m]</th>
<th>space [GB]</th>
<th>query time [µs]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reach</td>
<td>0:15</td>
<td>1.5</td>
<td>1253.5</td>
</tr>
<tr>
<td>CH</td>
<td>0:05</td>
<td>0.4</td>
<td>93.5</td>
</tr>
<tr>
<td>TNR</td>
<td>1:52</td>
<td>3.7</td>
<td>1.8</td>
</tr>
<tr>
<td>Table Lookup</td>
<td>&gt; 11:03</td>
<td>1 208 358.7</td>
<td>0.056</td>
</tr>
</tbody>
</table>

observation:

- excellent performance in practice
- used in production
- prime example for algorithm engineering
- but for a long time: no theoretical justification
A Theoretical Justification: Highway Dimension

$(r, k)$ shortest path cover

- all shortest paths with length between $r$ and $2r$ are hit
A Theoretical Justification: Highway Dimension

\((r, k)\) shortest path cover

- all shortest paths with length between \(r\) and \(2r\) are hit
(r, k) shortest path cover

- all shortest paths with length between r and 2r are hit

- locally sparse
  \( \leq k \) vertices in any ball of radius \( O(r) \)
A Theoretical Justification: Highway Dimension

\((r, k)\) shortest path cover

- all shortest paths with length between \(r\) and \(2r\) are hit
- locally sparse
  \((\leq k\) vertices in any ball of radius \(O(r))\)
(r, k) shortest path cover

- all shortest paths with length between r and 2r are hit
- locally sparse

(≤ k vertices in any ball of radius O(r))
A Theoretical Justification: Highway Dimension

(r, k) shortest path cover
- all shortest paths with length between $r$ and $2r$ are hit
- locally sparse
  ($\leq k$ vertices in any ball of radius $O(r)$)

Highway Dimension

A graph with highway dimension $h$ has an $(r, h)$-SPC for all $r$. 

[AFGW10]
A Theoretical Justification: Highway Dimension

\((r, k)\) shortest path cover
- all shortest paths with length between \(r\) and \(2r\) are hit
- locally sparse
  \((\leq k\) vertices in any ball of radius \(O(r)\))

Highway Dimension
A graph with highway dimension \(h\) has an \((r, h)\)-SPC for all \(r\).

results:
- sublinear query bounds for many algorithms
- best query bound: a labeling algorithm
- has not been considered in practical implementations
A Labeling Algorithm

preprocessing:

- compute a label $L(v)$ for each vertex $v$
- compute $\text{dist}(v, w)$ for each vertex $w \in L(v)$
- obey the label property:
  for all $s, t$ a shortest $s$–$t$ path intersects $L(s) \cap L(t)$

observation:
practical if labels are small

how to compute labels efficiently?

SPC algorithms currently are too slow
(maybe PHAST can help)
A Labeling Algorithm

preprocessing:

- compute a label $L(v)$ for each vertex $v$
- compute $\text{dist}(v, w)$ for each vertex $w \in L(v)$
- obey the label property:
  for all $s, t$ a shortest $s$–$t$ path intersects $L(s) \cap L(t)$

observation:
practical if labels are small

how to compute labels efficiently?
SPC algorithms currently are too slow (maybe PHAST can help)
A Labeling Algorithm

preprocessing:

- compute a label $L(v)$ for each vertex $v$
- compute $\text{dist}(v, w)$ for each vertex $w \in L(v)$
- obey the label property:
  - for all $s, t$ a shortest $s$–$t$ path intersects $L(s) \cap L(t)$

...queries:

find vertex $w \in L(s) \cap L(t)$ that minimizes $\text{dist}(s, v) + \text{dist}(v, t)$

observation:

practical if labels are small

how to compute labels efficiently?

SPC algorithms currently are too slow (maybe PHAST can help)
A Labeling Algorithm

preprocessing:

- compute a label $L(v)$ for each vertex $v$
- compute $\text{dist}(v, w)$ for each vertex $w \in L(v)$
- obey the label property:
  for all $s, t$ a shortest $s$–$t$ path intersects $L(s) \cap L(t)$

$s$–$t$ queries:

- find vertex $w \in L(s) \cap L(t)$ ...
A Labeling Algorithm

preprocessing:

- compute a label $L(v)$ for each vertex $v$
- compute $\text{dist}(v, w)$ for each vertex $w \in L(v)$
- obey the label property:
  for all $s, t$ a shortest $s\rightarrow t$ path intersects $L(s) \cap L(t)$

$s\rightarrow t$ queries:

- find vertex $w \in L(s) \cap L(t)$ ...
A Labeling Algorithm

preprocessing:
- compute a label $L(v)$ for each vertex $v$
- compute $\text{dist}(v, w)$ for each vertex $w \in L(v)$
- obey the label property:
  for all $s, t$ a shortest $s$–$t$ path intersects $L(s) \cap L(t)$

$s$–$t$ queries:
- find vertex $w \in L(s) \cap L(t)$ …
- … that minimizes $\text{dist}(s, v) + \text{dist}(v, t)$
A Labeling Algorithm

preprocessing:

- compute a label $L(v)$ for each vertex $v$
- compute $\text{dist}(v, w)$ for each vertex $w \in L(v)$
- obey the label property:
  for all $s, t$ a shortest $s$–$t$ path intersects $L(s) \cap L(t)$

$s$–$t$ queries:

- find vertex $w \in L(s) \cap L(t)$ . . .
- . . . that minimizes $\text{dist}(s, v) + \text{dist}(v, t)$

observation:

- practical if labels are small
- how to compute labels efficiently?
- SPC algorithms currently are too slow
  (maybe PHAST can help)
idea:

- Search spaces of contraction hierarchies form valid labels
- Run upward (forward and backward) search from each vertex, store label
- Sort label entries by node id

Query:
- Process like merge sort
- Update whenever the ids match

Very cache-efficient

Problem:
- Average label sizes of around 500
- \( \Rightarrow 150 \) GB of data
idea:

- search spaces of contraction hierarchies form valid labels
- run upward (forward and backward) search from each vertex, store label
- sort label entries by node id

\[ L(s) = \begin{bmatrix} 1,0 & 4,1 & 5,2 & 7,3 \end{bmatrix} \]
idea:

- search spaces of contraction hierarchies form valid labels
- run upward (forward and backward) search from each vertex, store label
- sort label entries by node id

\[
\begin{align*}
L(s) &= 1,0, 4, 1, 5, 2, 7, 3 \\
L(t) &= 2, 0, 6, 1, 7, 4
\end{align*}
\]
idea:

- search spaces of contraction hierarchies form valid labels
- run upward (forward and backward) search from each vertex, store label
- sort label entries by node id

\[ L(s) \begin{array}{cccc}
1,0 & 4,1 & 5,2 & 7,3 \\
\end{array} \]

\[ L(t) \begin{array}{cccc}
2,0 & 6,1 & 7,4 \\
\end{array} \]
idea:
- search spaces of contraction hierarchies form valid labels
- run upward (forward and backward) search from each vertex, store label
- sort label entries by node id

query:
- process like merge sort
idea:

- search spaces of contraction hierarchies form valid labels
- run upward (forward and backward) search from each vertex, store label
- sort label entries by node id

query:

- process like merge sort

\[
L(s) = [1, 0, 4, 1, 5, 2, 7, 3]
\]

\[
L(t) = [2, 0, 6, 1, 7, 4]
\]
idea:
- search spaces of contraction hierarchies form valid labels
- run upward (forward and backward)
  search from each vertex, store label
- sort label entries by node id

query:
- process like merge sort
idea:

- search spaces of contraction hierarchies form valid labels
- run upward (forward and backward) search from each vertex, store label
- sort label entries by node id

query:

- process like merge sort

\begin{align*}
L(s) &= 1, 0, 4, 1, 5, 2, 7, 3 \\
L(t) &= 2, 0, 6, 1, 7, 4
\end{align*}
idea:

- search spaces of contraction hierarchies form valid labels
- run upward (forward and backward) search from each vertex, store label
- sort label entries by node id

query:

- process like merge sort
idea:

- search spaces of contraction hierarchies form valid labels
- run upward (forward and backward) search from each vertex, store label
- sort label entries by node id

query:

- process like merge sort
- update whenever the ids match
idea:
- search spaces of contraction hierarchies form valid labels
- run upward (forward and backward)
- search from each vertex, store label
- sort label entries by node id

query:
- process like merge sort
- update whenever the ids match
- very cache-efficient

problem:
- average label sizes of around 500 ⇒ 150 GB of data
Optimizations

**label sizes:**

- 80% of the nodes in search spaces unnecessary
- prune by *bootstrapping*
- SPC algorithms on small important subgraph

$\Rightarrow$ average label size shrinks to 85 ($\rightarrow$ 24 GB)
Optimizations

label sizes:
- 80% of the nodes in search spaces unnecessary
- prune by bootstrapping
- SPC algorithms on small important subgraph
  \[ \Rightarrow \text{average label size shrinks to 85 (} \rightarrow 24 \text{ GB)} \]

reduce number of cache lines read:
- use compression (\( \rightarrow 6 \text{ GB) } \)
- define partition oracle to accelerate long-range queries
- many algorithmic low-level optimizations
  \[ \Rightarrow \text{we fetch only a few cache lines from memory} \]
### Results

<table>
<thead>
<tr>
<th>Method</th>
<th>Preprocessing time [h:m]</th>
<th>Space [GB]</th>
<th>Query time [μs]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reach</td>
<td>0:15</td>
<td>1.5</td>
<td>1253.5</td>
</tr>
<tr>
<td>CH</td>
<td>0:05</td>
<td>0.4</td>
<td>93.5</td>
</tr>
<tr>
<td>TNR</td>
<td>1:52</td>
<td>3.7</td>
<td>1.8</td>
</tr>
<tr>
<td>Table Lookup</td>
<td>&gt; 14:01</td>
<td>1208358.7</td>
<td>0.056</td>
</tr>
</tbody>
</table>
## Results

<table>
<thead>
<tr>
<th>method</th>
<th>time [h:m]</th>
<th>space [GB]</th>
<th>time [μs]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reach</td>
<td>0:15</td>
<td>1.5</td>
<td>1253.5</td>
</tr>
<tr>
<td>CH</td>
<td>0:05</td>
<td>0.4</td>
<td>93.5</td>
</tr>
<tr>
<td>TNR</td>
<td>1:52</td>
<td>3.7</td>
<td>1.8</td>
</tr>
<tr>
<td>HL</td>
<td>2:14</td>
<td>21.3</td>
<td>0.276</td>
</tr>
<tr>
<td>Table Lookup</td>
<td>&gt; 14:01</td>
<td>1 208 358.7</td>
<td>0.056</td>
</tr>
</tbody>
</table>

The labeling algorithm is the fastest engineered implementation guided by theory. The scientific method at work: practical algorithms are empirically fast. The highway dimension provides sublinear query bounds.
### Results

<table>
<thead>
<tr>
<th>method</th>
<th>preprocessing time [h:m]</th>
<th>space [GB]</th>
<th>query time [µs]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reach</td>
<td>0:15</td>
<td>1.5</td>
<td>1253.5</td>
</tr>
<tr>
<td>CH</td>
<td>0:05</td>
<td>0.4</td>
<td>93.5</td>
</tr>
<tr>
<td>TNR</td>
<td>1:52</td>
<td>3.7</td>
<td>1.8</td>
</tr>
<tr>
<td>HL</td>
<td>2:14</td>
<td>21.3</td>
<td>0.276</td>
</tr>
<tr>
<td>HL compressed</td>
<td>2:45</td>
<td>5.7</td>
<td>0.527</td>
</tr>
<tr>
<td>Table Lookup</td>
<td>&gt; 14:01</td>
<td>1208 358.7</td>
<td>0.056</td>
</tr>
</tbody>
</table>

Scientific method at work: practical algorithms are empirically fast. Highway dimension and sublinear query bounds. The labeling algorithm is the fastest. Engineered implementation guided by theory. New running time record.
Results

<table>
<thead>
<tr>
<th>method</th>
<th>preprocessing time [h:m]</th>
<th>space [GB]</th>
<th>query time [µs]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reach</td>
<td>0:15</td>
<td>1.5</td>
<td>1253.5</td>
</tr>
<tr>
<td>CH</td>
<td>0:05</td>
<td>0.4</td>
<td>93.5</td>
</tr>
<tr>
<td>TNR</td>
<td>1:52</td>
<td>3.7</td>
<td>1.8</td>
</tr>
<tr>
<td>HL</td>
<td>2:14</td>
<td>21.3</td>
<td>0.276</td>
</tr>
<tr>
<td>HL compressed</td>
<td>2:45</td>
<td>5.7</td>
<td>0.527</td>
</tr>
<tr>
<td>Table Lookup</td>
<td>&gt; 14:01</td>
<td>1 208 358.7</td>
<td>0.056</td>
</tr>
</tbody>
</table>

Scientific method at work:
- Observation: practical algorithms are empirically fast
- Theory: highway dimension and sublinear query bounds
- Prediction: the labeling algorithm is the fastest
- Verification: engineered implementation guided by theory
Results

<table>
<thead>
<tr>
<th>method</th>
<th>preprocessing time [h:m]</th>
<th>space [GB]</th>
<th>query time [µs]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reach</td>
<td>0:15</td>
<td>1.5</td>
<td>1253.5</td>
</tr>
<tr>
<td>CH</td>
<td>0:05</td>
<td>0.4</td>
<td>93.5</td>
</tr>
<tr>
<td>TNR</td>
<td>1:52</td>
<td>3.7</td>
<td>1.8</td>
</tr>
<tr>
<td>HL</td>
<td>2:14</td>
<td>21.3</td>
<td>0.276</td>
</tr>
<tr>
<td>HL compressed</td>
<td>2:45</td>
<td>5.7</td>
<td>0.527</td>
</tr>
<tr>
<td>Table Lookup</td>
<td>&gt; 14:01</td>
<td>1208 358.7</td>
<td>0.056</td>
</tr>
</tbody>
</table>

**scientific method at work:**

- **observation:** practical algorithms are empirically fast
- **theory:** highway dimension and sublinear query bounds
- **prediction:** the labeling algorithm is the fastest
- **verification:** engineered implementation guided by theory

⇒ new running time record