GPU Road Network Graph Contraction and SSSP Query

Roozbeh Karimi
rkarim2@lsu.edu
Louisiana State University, ECE
Baton Rouge, LA, USA

David M. Koppelman
koppel@ece.lsu.edu
Louisiana State University, ECE
Baton Rouge, LA, USA

Chris J. Michael
chris.michael@nrlssc.navy.mil
Naval Research Laboratory
Stennis Space Center, MS, USA

ABSTRACT

PHAST is to date one of the fastest algorithms for performing *single source shortest path (SSSP)* queries on road-network graphs. PHAST operates on graphs produced in part using Geisberger's *contraction hierarchy (CH)* algorithm. Producing these graphs is time consuming, limiting PHAST's usefulness when graphs are not available in advance. CH iteratively assigns scores to nodes, contracts (removes) the highest-scoring node, and adds shortcut edges to preserve distances. Iteration stops when only one node remains. Scoring and contraction rely on a *witness path search (WPS)* of nearby nodes. Little work has been reported on parallel and especially GPU CH algorithms. This is perhaps due to issues such as the validity of simultaneous potentially overlapping searches, score staleness, and parallel graph updates.

A GPU contraction algorithm, *CU-CH*, is presented which overcomes these difficulties by partitioning the graph into levels composed of independent sets of nodes (non-adjacent nodes) with similar scores. This allows contracting multiple nodes simultaneously with little coordination between threads. A GPU-efficient WPS is presented in which a small neighborhood is kept in shared memory and a hash table is used to detect path overlap. Low-parallelism regions of contraction and query are avoided by halting contraction early and computing APSP on the remaining graph. A PHAST-like query computes SSSP using this contracted graph. Contraction of some DIMACS road network graphs on an Nvidia P100 GPU achieves a speedup of 20 to 37 over Geisberger's serial code on a Xeon E5-2640 v4. Query times on CU-CH- and CH-contracted graphs were comparable.

ACM Reference Format:
Roozbeh Karimi, David M. Koppelman, and Chris J. Michael. 2019. GPU Road Network Graph Contraction and SSSP Query. In *2019 International Conference on Supercomputing (ICS '19), June 26–28, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3330345.3330368

1 INTRODUCTION

An *SSSP* query for some $s \in V$ of weighted graph $G = (V, E)$ computes the distance from s to all nodes in V. Though SSSP is a well studied problem [5, 7–9, 11, 14–16, 18, 21, 22, 26, 29] one area that has received little attention is parallel and GPU implementations for road-network graphs. For example, Davidson *et al*'s otherwise well-performing GPU SSSP implementation yields slowdowns over CPU code on road-network graphs [8]. The problem is that the size of the workfront (set of nodes ready to be operated on) is often too small for GPUs' demanding parallelism requirements.

Delling *et al*, in *PHAST*, overcome this problem in part by adapting Geisberger *et al*'s *contraction hierarchies (CH)* [15] to SSSP queries [9]. For point-to-point queries a search on each of the *upward* and *downward* graphs produced by CH will identify a node on the shortest path while visiting far fewer nodes than would a bidirectional Dijkstra search on an ordinary graph [15]. Delling adapts this to parallel SSSP queries by using an upward graph search from a source to compute distances for a small subset of nodes followed by a fixed-order traversal of all nodes in the downward graph. The nodes are organized into *levels* such that in the downward pass updates to nodes within a level can be done in parallel. Delling reports very efficient implementations of this SSSP query, approaching data bandwidth saturation [9].

Because one-time preparation of the upward and downward graphs is time consuming PHAST and other uses of contracted graphs are less useful when the graph is not available in advance. For example, Geisberger reports that it can take about 10 minutes to compute the CH of a Western Europe graph with roughly 18 million nodes [15].

A fast GPU contraction and query algorithm, CU-CH, is described here. CU-CH computes the *parallel contraction (*PCH*)* of a graph, including structures needed for query. Parallelism is facilitated by computing PHAST-like levels in lieu of CH *ranks*. In a further departure from CH, the graph is not fully contracted. Instead the remnant is converted into a complete graph, avoiding what would be inefficient steps in both contraction and query. GPU-efficient local searches are used to find *witness paths*. Both the contraction and query are performed on a GPU.

The remainder of this paper is organized as follows: graph terminology, and the CH and PHAST techniques are described in Section 2. CU-CH is described in Section 3, followed by experimental methodology in Section 4 and results in Section 5. Those are followed by prior work, Section 6, and conclusions, Section 7.

2 BACKGROUND

2.1 Preliminaries

A positive weighted directed graph, $G = (V, E)$, consists of a set of nodes, V, and a set of weighted edges, $E \subset V \times V \times \mathbb{R}_{\geq 0}$. Both $e \in E$ and $(u, v) \in E$ denote edges, $\text{len}(e)$ and $\text{len}(u, v)$ denote their respective weights. For brevity *graph* will refer to such a graph, with the further restriction that there are no edges such as $(u, u) \in E$. A path is a sequence of nodes connected by edges, its length is the sum of the weights. The *distance* from $s \in V$ to $t \in V$ in G, $\text{Dist}_G(s, t)$, is the length of a shortest path from s to t. Triple $(u, w, v) \in E$ denotes two-hop path (u, w), (w, v) and $\text{len}(u, w, v)$ denotes its length. For $U \subseteq V$ define $U^2 = U \times U$, and $V \setminus U = \{ v \mid v \in V,\ v \notin U \}$, and

ICS '19, June 26–28, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6079-1/19/06…$15.00
https://doi.org/10.1145/3330345.3330368

$E \setminus U = \{ (u, v) \mid (u, v) \in E,\ u, v \notin U \}$. In places V and E are used for $|V|$ and $|E|$, the number of nodes and edges, d is used for the average degree, $|E|/|V|$, and D is used for the maximum degree, all for some understood graph.

A *road network graph* is a weighted graph used to model a transportation system in which edge weights are set to some cost of travel (*e.g.*, distance, time) between incident nodes. Such graphs have been observed to have low degree and small separators [4], factors likely to contribute to the stark difference in performance of SSSP-like algorithms on such graphs compared to other classes of graphs. Work to better define and characterize road network graphs is ongoing [2, 3, 12, 13].

2.2 Dijkstra's Algorithm

Dijkstra's algorithm [11] is the classic sequential algorithm for calculating SSSP. The distance of some $s \in V$ is initialized to zero, all others to infinity, and all nodes are placed in a priority queue. The node with the smallest distance is removed and the distances of its forward neighbors are updated. This step is repeated until the queue is empty. Dijkstra shows that the distance of a node removed from the queue can not change and is the correct distance from s [11]. Nodes that have been removed from the queue are said to be *settled*. *Relaxing* an edge refers to following it to update a distance. The execution time is $|E|$ relaxations plus $|V|$ queue removals.

The realized performance of Dijkstra depends on the method used for maintaining the priority queue. Using a Fibonacci heap the Dijkstra algorithm has a theoretical time complexity of $O(E + V \log V)$ [4] however in practice the Fibonacci heap suffers from large overhead, so most implementations use different priority queue implementations such as a binary heap which results in an overall complexity of $O(E \log V)$ [4]. This makes Dijkstra theoretically the most efficient sequential algorithm for SSSP.

The only obvious parallelism is in relaxing a node's neighbors, which is insufficient on most graphs and hardware. This makes standard Dijkstra unsuitable for modern parallel processors such as GPUs. The Bellman Ford algorithm [7, 14] sacrifices efficiency for parallelism by relaxing each edge $|V|$ times, or $O(EV)$ total work. GPU SSSP implementations blend these approaches, see the discussion in Section 6.

2.3 Contraction Hierarchy (CH)

The Contraction Hierarchy (CH) technique [15] was designed to speed up point-to-point shortest-path calculations on road network graphs. The algorithm constructs an *augmented graph* G_A for a given graph G by assigning a unique *rank* to each node and adding weighted *shortcut edges*. The shortcut edges are chosen so that for any two nodes $u, v \in V$ there exists a path of length $\text{Dist}_G(u, v)$ that omits nodes ranked lower than $\min\{r(u), r(v)\}$. Traversals limited to paths of ascending (or descending) rank order can touch far fewer nodes and are the basis of fast shortest path algorithms [15] and for techniques like PHAST [9] which avoid the limited workfront that frustrates ordinary parallel SSSP algorithms [8] when operating on road network graphs.

The *contraction hierarchy* of graph $G = (V, E)$, denoted $G_{CH} = (G, r, A)$, consists of a bijection $r\colon V \to [0, |V|)$, called the *node ranking*, and a valid set of weighted shortcut edges, $A \subseteq V^2$. Associated with G_{CH} are overlay graphs $G_0, G_1, \ldots G_{|V|-1}$, where $G_i = (V_i, E_i)$, $V_i = \{ v \mid r(v) \geq i \}$, and $E_i = (E \cup A) \cap V_i^2$. Shortcuts A are said to be valid if $\text{Dist}_{G_i}(s, t) = \text{Dist}_G(s, t)$ for all $s, t \in V_i$ and $i \in [0, |V|)$.

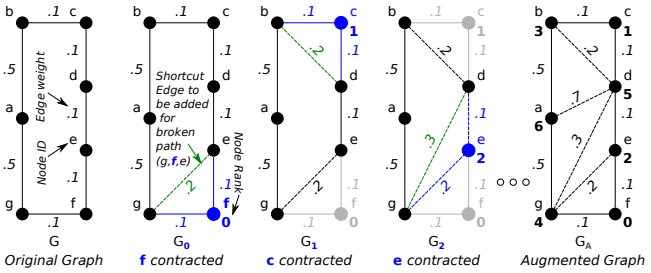


Figure 1: Contraction Example

Derived from G_{CH} are the *augmented graph*, $G_A = (V, E \cup A)$, and the *upward* and *downward* graphs, $G_\uparrow = (V, E_\uparrow)$ and $G_\downarrow = (V, E_\downarrow)$, where $E_\uparrow = \{ (u, v) \mid (u, v) \in E \cup A,\ r(u) < r(v) \}$ and $E_\downarrow = \{ (u, v) \mid (u, v) \in E \cup A,\ r(u) > r(v) \}$ [15]. Geisberger shows that a forward Dijkstra search from $s \in V$ in $G_\uparrow$ and a backward Dijkstra search from $t \in V$ in $G_\downarrow$ will both reach the highest rank node on a shortest path from s to t (if such a path exists) [15].

Typically a CH graph is constructed iteratively. At step i, operating on overlay graph G_i, a node $w \in V_i$ is chosen to receive rank i thus fixing $V_{i+1} = V_i \setminus w$. For each *broken path* $(u, w, v) \in E_i$ shortcut edge (u, v) of length $\text{len}(u, w, v)$ is appended to A if it is possible that $\text{Dist}_{(V_{i+1}, E_i \cup A \cap V_{i+1}^2)}(u, v) > \text{len}(u, w, v)$, where A is the set of shortcuts before considering (u, v). This determination is made by performing a *witness path search (WPS)* from u to v: a distance query which avoids w and which terminates early if the distance would be over $\text{len}(u, w, v)$. *Node contraction* refers to the process of finding shortcuts and constructing the next overlay.

Contraction is illustrated in Figure 1. The first node chosen for contraction is f. Removing f from G_0 breaks path (g, f, e) and so shortcut edge (g, e) of weight .2 is added to preserve the distance between g and e. (If a were contracted a shortcut would not be needed because the shortest path from b to g does not pass through a.) Graph G_1 is formed from G_0 by eliminating f and its incident edges, and adding the shortcut. (Eliminated edges are shown in light gray.) The next two graphs, G_1 and G_2 show the contraction of c and e. The augmented graph, G_A, was obtained by continuing with b, g, d, and a. Graph $G_\uparrow$ is obtained from G_A by directing each edge at the higher-ranked node, or retaining only such edges if the graph is directed. Dijkstra-like searches on $G_\uparrow$ usually visit a small fraction of V. For example, a search starting at b only visits d and a.

The character of the CH graph is determined by the ranking. For example, with an ill-chosen ranking the number of shortcuts will be unacceptably large. Ranking is typically performed by computing *scores* for unranked nodes and assigning the next rank to the highest-scoring node. Geisberger computes scores using a number of factors, chief among them is *edge difference*, the change in the number of edges if the node were contracted [15].

Calculating edge difference for $w \in V_i$ requires the results of a WPS on every broken path $(u, w, v) \in E_i$, a total of $|V_i|d^2$ searches.

On top of that, after each iteration the scores for the remaining nodes can change, so in order to maintain an accurate score, some of the scoring criteria will need to be recalculated for the overlay graph. Though for edge difference it only needs to be recalculated for the immediate neighbors of w rather than the entire graph. Even so, this can be very costly. To address this Geisberger introduces a *lazy update* scheme to selectively rescore nodes.

In Geisberger's implementation the WPS for broken path (u, w, v) is performed using a bidirectional Dijkstra search. It is necessary that the WPS be correct when it finds a witness path, otherwise an essential shortcut will be omitted. However, if the WPS does not find a witness path that does exist a superfluous shortcut will be added, hurting performance but not effecting correctness. This enables the use of faster inexact searches, such as those with hop limits.

2.4 PHAST: Parallel SSSP CH Queries

Delling *et al* introduce *PHAST* [9], a parallel SSSP query algorithm that starts with the hierarchy provided by a CH graph. *Levels* are assigned to nodes so that $(u, v) \in E_\uparrow \Rightarrow L(u) < L(v)$ where $L : V \rightarrow [0, l_{\max})$ indicates the level and $l_{\max}$ is the number of levels. Delling suggests determining levels during contraction as follows: Initially the level of each node is set to zero. When w in G_l is contracted set $L(x) = \max\{L(x), L(w) + 1\}$ for all $x : (w, x) \in E_l \vee (x, w) \in E_l$ [9].

Given such a level assignment an SSSP query for $s \in V$ proceeds as follows. In the *upward pass* use Dijkstra's algorithm to compute distances in $G_\uparrow$. The time needed for this step should be small. At this point nodes at level $l_{\max} - 1$ (the highest) reachable from s are settled (have final distances assigned). Next, in the *downward pass*, nodes at level $l_{\max} - 2$ update their distances following backward edges in $G_\downarrow$. These edges can only reach nodes in level $l_{\max} - 1$, which have already been settled. The process is repeated level by level until all nodes are settled. Since nodes in a level only access nodes in higher levels, node update within a level can easily be done in parallel [9].

By ordering edge lists by destination the distance update can be implemented using contiguous loads of edges within a level for high efficiency. The gather-style accesses needed to load higher-level nodes' distances are less efficient on certain systems but even that can be eliminated by performing multiple SSSP queries in parallel. The combination of these characteristics makes PHAST especially suitable for modern processor architectures with vector processing capabilities as well as GPUs [9].

2.5 GPU Background

GPUs achieve high floating-point and memory bandwidth by eliminating CPUs' extensive mechanisms for delivering operands to functional units (such as large caches and dynamic scheduling) and replacing them with more efficient mechanisms including an elaborate and exposed memory hierarchy and a large number of thread contexts (to hide latency without speculation or bypassing), and so these devices are dependent on well-tuned code. GPU organizations continue to evolve, the description below is of Nvidia *Pascal*-generation GPUs, however the older Maxwell and newer Volta/Turing generations are similar.

Code is prepared for Nvidia GPUs using the *CUDA* toolchain. Code is initiated in a *kernel*, which consists of *warps* of threads grouped into *blocks*. Blocks are dispatched to *SMs*, where they run to completion. An SM has up to 128 functional units (64 on the P100) and can actively schedule 64 warps. Four warps provide sufficient threads for the functional units, additional warps are used to hide latency. It is not unusual to require 16, 32, or 64 warps to maximize performance. An Nvidia P100 has 56 SMs.

Re-used data is kept in *shared memory* and registers, which have access times comparable with CPU L1 caches. The P100 has 64 kiB and 256 kiB of register storage per SM. CU-CH carefully uses these resources for its working sets, especially node neighborhoods. The P100 has 3 MiB of L2 cache, its small size and high latency make it useful primarily for reducing off-chip bandwidth. The P100 has an off-chip bandwidth of 550 GB/s. It is assumed that readers are familiar with issues and techniques associated with GPU data layout and access. [6, 24] provide a good review of the material.

3 CU-CH: CUDA CH CONSTRUCTION, QUERY

The *CU-CH contraction algorithm*, given some input graph, constructs a *parallel contraction hierarchy (PCH) graph* along with other graphs needed for query. The PHAST-like *CU-CH query algorithm* uses these to perform SSSP queries. Both are implemented in CUDA tuned for Nvidia Pascal-generation GPUs.

Parallel contraction is achieved by assigning nodes to levels and contracting nodes within a level in parallel. The challenge is to assign levels and to restrict witness path searches so that the parallel WPS results are valid despite traversing contracted nodes. These levels are compatible with PHAST's, but they are assigned in lieu of ranking rather than determined after ranking. For efficient parallel contraction and query there must be a sufficient number of nodes in a level. This is a problem in later levels where there are too few nodes. CU-CH avoids the problem by contracting an overlay graph only if the number of nodes is above a threshold, otherwise contraction stops and the overlay is converted into a complete graph. The computation of the complete graph and its use for queries are both efficient GPU operations.

3.1 CU-CH Graph Description

The PCH of $G = (V, E)$, denoted $G_{\text{PCH}} = (G, \mathcal{U}, \mathcal{A})$, consists of a valid partition of V, $\mathcal{U} = U_0, U_1, \ldots, U_{l_{\max}}$, and valid sets of weighted shortcuts $\mathcal{A} = A_0, A_1, \ldots, A_{l_{\max}-2}$. Define overlay graphs $G_i = (V_i, E_i)$ as follows: Set $V_0 = V$ and $E_0 = E$. Then $V_{i+1} = V_i \setminus U_i$ and $E_{i+1} = A_i \cup E_i \setminus U_i$ for $i \in [0, l_{\max} - 2]$. Finally set $V_{l_{\max}} = U_{l_{\max}}$, $E_{l_{\max}} = U^2_{l_{\max}}$, and $\text{len}(u, v) = \text{Dist}_G(u, v)$ for all $(u, v) \in E_{l_{\max}}$. Nodes in U_i are said to be in level i.

The partition $\mathcal{U}$ is valid if $E_i \cap U_i^2 = \emptyset$ for $i \in [0, l_{\max})$, that is, if U_i is an independent set in G_i. For $\mathcal{A}$ to be valid $\text{Dist}_{G_i}(s, t) = \text{Dist}_G(s, t)$ for all $s, t \in V_i$, $i \in [1, l_{\max} - 1]$. A PCH for which $|\mathcal{U}| = |V|$ is equivalent to a CH with level serving as rank.

3.2 CU-CH Algorithm Overview

Algorithm 1 constructs the PCH described above, plus structures needed for query. The input is the graph to be contracted, G, and tuning constants $K < |V|$, the *cutoff*, and $C \in (0, 1]$, the *selection fraction*. The output consists of the upward and downward graphs,

$G_\uparrow$, $G_\downarrow$, and the complete graph, G_K. The output graphs contain the shortcut *midpoints* and other information needed to reconstruct paths. The algorithm initializes overlay graph G_0 to the input graph and then enters the main loop, which iterates until the number of nodes in the current overlay graph is less than the cutoff, $|V_l| < K$. Each iteration has five steps: The *Score* step computes scores for all nodes. The *Select* step determines the set of nodes to contract, U_l, by finding an independent set among the $C|V_l|$ highest-scoring nodes. The *Shortcut* step finds candidate shortcut edges, A_{l+}, around the contracted nodes. The *Update* step appends contracted nodes and needed edges to the upward and downward graphs, and it uses A_{l+} to create intermediate overlay graph G_{l+}. Finally, the *Extract* step creates the next overlay graph, G_{l+1}, by removing contracted nodes and attached edges and by renumbering retained nodes. After the main loop exits the *APSP* step computes G_K. A rough complexity analysis appears below and measured times for the main loop steps operating on a mid-sized graph are plotted in Figure 9 for varying C, see Section 4.

By far the most computationally challenging operation performed is the witness path search, used by the Score and Shortcut steps. A conventional approach would perform a bidirectional distance-limited Dijkstra search for each of the d^2 broken paths in length order starting at the shortest-length broken path. Shortcuts found by earlier searches can be used in later searches. CU-CH foregoes this benefit for parallelism and efficiency: a combined approach is used for the d^2 searches per node, reducing the total work to $O(d^2)$ per node for the Score step and $O(d^3)$ work for the Shortcut step. The correctness of these searches is discussed in Section 3.4 and their implementation in Section 3.5.

The Select step is performed quickly: Scores are sorted to identify the $C|V_l|$ top-scoring nodes, U_{l+}, followed by two iterations of Luby's MIS algorithm [20] to find an independent set among these.

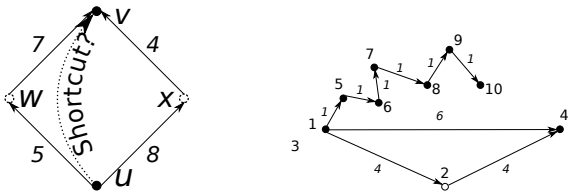
Careful design was required for the Update and Extract steps. The witness-path searches dominate execution time and their performance is sensitive to the layout of edge lists. If edges to contracted nodes are present, time will be lost checking for these edges or otherwise avoiding them and L2 cache space would be wasted. For these reasons Extract constructs a new overlay graph each iteration. This may sound wasteful, but Extract consumes relatively little time. See Section 3.7 for details of data layout.

The Shortcut and Update steps each operate just once on each node in V. In contrast the other steps each operate on nodes from 1 to $l_{\max}$ times. The average number of times each node is operated on can be estimated by assuming a constant independent set size. Call $\lambda_l = |U_l|/|V_l|$ the *contraction fraction* and call $\alpha_l = |U_l|/|U_{l+}|$ the *MIS fraction*. Assume, a bit unrealistically (see Figure 4), that $\alpha_0 = \alpha_1 = \cdots = \alpha_{l_{\max}-1} = \alpha$ and so $\lambda_l = \lambda = \alpha C$. Then $|V_l| = (1-\alpha C)^l |V|$ and $\sum_0^\infty |V_l| = \frac{1}{\alpha C}|V|$ and so steps such as Score operate on each node an average of $\frac{1}{\alpha C}$ times.

Based on this analysis setting C to larger values should reduce the number of times nodes are operated on, and this is borne out by measurement, see Figure 9. Because the score for a node is an estimate of changes if it were the only node contracted, increasing C weakens the predictive power of the scores. Among the measured implications are more shortcut edges and higher-degree overlay graphs and so more time needed for the $O(d^2)$ and $O(d^3)$ complexity

Algorithm 1: CU-CH

Input: Graph $G = (V, E)$
Constant: $K = 1025$, $C = .3$
Output: Graphs $G_\uparrow$, $G_\downarrow$, G_K
$G_\uparrow = G_\downarrow = (\emptyset, \emptyset)$; $G_0 = G$; $l = 0$;
while $|V_l| \geq K$ **do**
  // **Score:** Calculate scores of $v \in V_l$.
  SC = `Score(`G_l`)` // Hard Part: 1-hop witness-path search.
  // **Select:** Based on scores, find nodes to contract, U_l.
  U_{l+} = `Sort(SC)` // Find indices of $C|V_l|$ highest-scoring nodes.
  U_l = `Luby_Two(`G_l, U_{l+}`)` // Find $\subseteq U_{l+}$ that is indep. set in G_l.
  // **Shortcut:** Find shortcuts, A_{l+}, for contracted nodes, U_l.
  A_{l+} = `Shortcut(`G_l,U_l`)` // Hard Part: 2-hop witness-path srch.
  // **Update:** Add shcuts to G_l. Append U_l, etc. to $G_\uparrow$ and $G_\downarrow$.
  (G_{l+}, $G_\uparrow$, $G_\downarrow$) = `Update_Graph(`G_l, $G_\uparrow$, $G_\downarrow$, A_{l+}`)`;
  // **Extract:** Construct G_{l+1} using renumbered $V_l \setminus U_l$.
  G_{l+1} = `Extract_Overlay(`G_{l+}, U_l, *CU-CH*`)`;
  $l = l + 1$;
// **APSP:** Perform a blocked APSP on graph final overlay G_l.
G_K = `Blocked_APSP(`G_l`)`;



(a) Omit necessary shortcut (u, v) because WPS for w finds (u, x, v), likewise for x.

(b) 1-hop search finds a witness path where a 5-hop limited Dijkstra search does not

Figure 2: Witness-Path Search Examples

WPS used in the Score and Shortcut steps and longer query times. Figures 3 and 4 show the impact of C on d_l and α_l as tested on the CAL graph (see Section 4.1). Issues and measurements related to C are discussed further in Section 5.2.

3.3 Scoring

The scoring criteria used in CU-CH are based on those described by Geisberger *et al* [15]. The score is computed using edge difference, degree, maximum degree of forward and backward neighbors, normalized average weight of incoming and outgoing edges, and normalized maximum weight of incoming and outgoing edges. As with Geisberger's well-performing scoring methods, edge difference is given the largest weight when computing the score. In CU-CH less than half the execution time is spent scoring, so selective re-scoring methods were not considered.

3.4 Parallel Contraction and Shortcuts

While finding a set of shortcuts for one contracted node, $|U_l| = 1$, is straightforward, finding shortcuts for multiple nodes in parallel presents two potential problems. First, if two contracted nodes are neighbors, say $w, x \in U_l$ and $(w, x) \in E_l$, then $A_{l+} = \{(u, v) \mid$

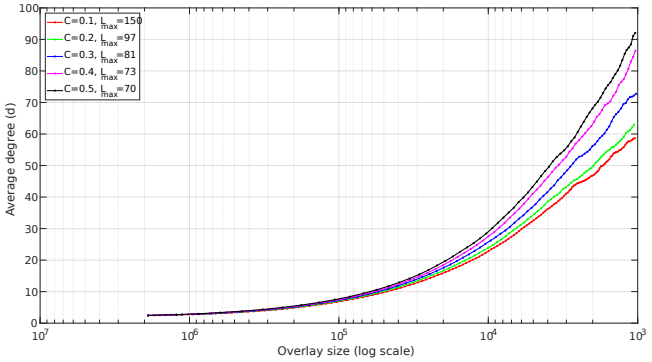


Figure 3: Overlay degree, d, v. size for CAL.

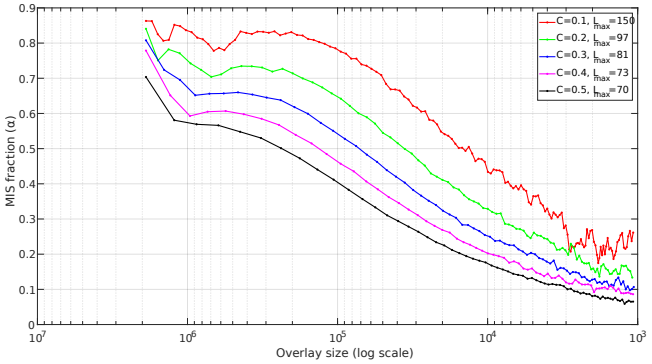


Figure 4: MIS fraction, α, v. overlay size for CAL.

$(u, w, v) \in E_l$ } will no longer serve as a set of candidate shortcuts because $(u, x) \in A_{l+}$ though x won't be in the next overlay. The solution used by CU-CH is to simply restrict U_l to be an independent set in G_l. This restriction has the added benefit of satisfying the requirements for PHAST-like queries, see Section 2.4.

The second problem is more subtle. Let $w, x \in U_l$ and suppose $(u, w, v) \in E_l$, $(u, x, v) \in E_l$, and $\text{len}(u, w, v) = \text{len}(u, x, v)$. In this case the witness path search for broken path (u, w, v) might find (u, x, v) and vice versa. Ordinarily if a witness path is found a shortcut is not added, but if that rule were followed here a necessary shortcut, (u, v), would be omitted. This is illustrated in Figure 2a. It would be simple enough to avoid this problem by checking whether traversed nodes are in U_l, but since the WPS dominates execution time the cost of such checks would be significant. Instead, CU-CH tightens the condition on witness paths of more than one hop: their length must be *strictly* less than the length of the broken path. With this condition (u, w, v) could not be a witness for broken path (u, x, v). Edge $(u, v) \in E_l$ with $\text{len}(u, v) = \text{len}(u, x, v)$ could still serve has a witness path for either broken path because it is only one hop.

The tightened WP condition will result in only a few more shortcut candidates than the original condition. For situations with equal-length broken paths (u, w, v) and (u, x, v) two identical shortcut candidates would be generated. (CU-CH will merge them into one.) If the equal-length witness path does not contain a contracted edge an unnecessary shortcut might be added. But equal-length paths should be rare for road network graphs in which lengths are based on physical distance or travel time.

3.5 Witness Path Search

A major challenge for CU-CH is performing witness path searches efficiently on the GPU. Anything like Dijkstra's algorithm is out of the question due to irregular memory access and limited parallelism. GPU SSSP algorithms like those of [8] were designed for a single SSSP computation on an entire graph, whereas the SSSP for a WPS need touch only a small part of the graph, and multiple such searches need to be performed in parallel. The CU-CH WPS implementation is specifically designed for such parallel local searches. The search efficiently visits all nodes within 1 or 2 hops of the contraction candidate. For the 1-hop search each visited node is loaded from global memory just once, and the entire search takes $O(d^2)$ work, which is the minimum complexity for checking d^2 broken paths. The 2-hop search loads each node from global memory $\lceil d/h \rceil$ times and takes $O(d^3)$ work, where h is determined by the number of available registers, $h = 32$ on the P100. Global memory access is ideal here too until the degree exceeds h, but at that point execution is likely computation bound. Some details of the algorithm are described below.

The WPS finds witness path candidates for broken path $(u, w, v) \in E_l$ by identifying nodes $Y = \{ y \mid (y, v) \in E_l,\ y \neq w \}$ and then checking whether nodes on forward paths from u are members. If $u \in Y$ a 1-hop path has been found; if $(u, x) \in E_l$ and $x \in Y$ a 2-hop path has been found. The found path's length is compared to the broken path's length to determine whether it is a witness path. Set membership is efficiently tested using a simple hash table that takes only $O(1)$ work per thread to load and lookup.

The WPS is summarized in Algorithm 2. For an overlay graph of maximum degree D_l, a group of $T_l = \max\{8, 2^{\lceil \lg D_l \rceil}\}$ threads finds the shortcuts needed for a contraction candidate. Each thread returns a bit vector identifying prudent shortcuts. Let $u_0, u_1, \ldots$ and $v_0, v_1, \ldots$ denote the backward and forward neighbors of w. Thread τ finds witness paths starting at u_τ. Shared memory is used for edges $(w, v_\star)$ throughout the execution. Within the two q-loops, thread τ loads (y_τ, v_q) into shared memory and writes τ to the hash table using key y_τ. The hash table is implemented with an array of bytes, the hash function is the lower bits of the node ID, and collisions result in lost elements. Thread τ uses the hash table to find a node in $y_\star$ which is the same as u_τ (for the 1-hop search) or some $x \mid (u_\tau, x) \in E_l$ (for the 2-hop search). Collisions are rare. In tests using the CAL graph they occur in less than 1% of the WPS's. To assess their impact fallback code is invoked on collisions to iteratively compare x or u_τ to each element of Y.

The 1-hop search only has a single loop, of d iterations, and so the total work is $O(d^2)$ assuming $d \approx T_l$ and loads $O(d^2)$ data. This suggests bandwidth-bound execution.

The 2-hop search is a greater challenge because for each q, $(y_\star, v_q)$ must be compared to every $X = \{ (u, x) \mid (u, w) \in E_l,\ (u, x) \in E_l,\ u \neq x \}$. Call X the *frontier*. CU-CH stores the frontier in GPU registers, the most abundant high-speed storage available. When there are not enough registers for the entire frontier something will need to be loaded multiple times. Let h denote the maximum frontier size a thread can accommodate. CU-CH loads the frontier

Algorithm 2: Witness Path Search

Input: Contraction candidate: $w \in V_l$.
Output: Shortcut vector: $S_{[q]} = 1$ if shortcut (u_τ, v_q) prudent.
Threads :Use T threads per node; IDs: $\tau \in [0, T)$.

```
// Thread τ checks broken paths (u_τ, w, v_⋆).
Regs   d ← w:D                                   // Degree of w.
Regs   u_τ ← w:B[τ]:NW                           // Edge (u_τ, w).
Shared v_[τ] ← w:F[τ]:NW                         // Edge (w, v_τ).
Regs   S_[q] ← u_τ ≠ v_[q] for q ∈ [0, d)        // Init shortcut vector.
// One-Hop Search
for q ∈ [0, d) do                    // Can (u_τ, v_q) witness (u_τ, w, v_q)?
    l ← len(u_τ, w) + len(w, v_[q])              // Broken path length.
    Shared y_[τ] ← {w:F[q]::B}[τ]:NW             // Edge (y_τ, v_q)
    Hash_Insert.Key( y_[τ]).Val ( τ )
    e ← Hash_Lookup( u_τ)                        // Look for e such that y_e = u_τ.
    if y_[e] = u_τ ∧ len(y_[e], v_[q]) ≤ l then S_[q] ← 0
// Two-Hop Search
for r ∈ {0, h, 2h, ..., d} do // Edges (u_τ, x_r), ... (u_τ, x_{r+h-1})
    Regs x_[j] ← {w:B[τ]::F}[r + j]:NW for j ∈ [0, h)
    for q ∈ [0, d) do // Can a (u_τ, x_⋆, v_q) witness (u_τ, w, v_q)?
        l ← len(u_τ, w) + len(w, v_[q])          // Broken path length.
        Shared y_[τ] ← {w:F[q]::B}[τ]:NW         // Edge (y_τ, v_q)
        Hash_Insert.Key( y_[τ]).Val ( τ )
        for j ∈ [0, h) do // Can (u_τ, x_j, v_q) witness (u_τ, w, v_q)?
            e ← Hash_Lookup( x_[j])              // Seek e such that y_e = x_j
            if x_[j] = y_[e] ∧ len(u_τ, x_[j]) + len(y_[e], v_q) < l
              then S_[q] ← 0
```

in h-node chunks per thread. This chunking does not change the number of hash lookups (inner j loop iterations in the code) but does increase the number of hash inserts from d to $d\lceil d/h \rceil \approx d^2/h$ and the amount of data loaded from $O(d^2)$ to $O(d^3/h)$. Regardless of h the total work is $O(d^3)$.

Fortunately the P100 can handle up to $h = 32$. CU-CH sets h so that re-use only occurs when the degree exceeds 32. At that point re-use is likely not slowing execution because each loaded y is compared against 32 x_j and so execution is either computation-bound or there are not enough contracted nodes to saturate any resource.

Algorithm 2 shows global access using shorthand in which `i:H` denotes an access to array `H` at index i, that is, `H[i]` when written in C. The C equivalent of chained accesses `w:B[τ]:N` is `N[B[w] + τ]`. Each `:` denotes an array access. Braces surround values cached in shared memory. For example, `{w:B[τ]::F}[r+j]:W` indicates that the value `F[N[B[w]+τ]]` had earlier been placed in shared memory, and that is being accessed, call it x. Then `W[x + r + j]` is accessed from global memory. `B` and `F` are node-indexed arrays returning edge-list indices. `N` and `W` are edge-indexed arrays returning node ID and edge weight. Many details are implicit, such as whether `N` is a forward or backward edge list.

3.6 Overlay APSP

As contraction progresses one would expect $\alpha_l = |U_l|/|U_{l+}|$ to decrease, in part due to increasing degree, and that has been observed on tested graphs such as CAL in Figure 4. This, combined with the decreasing size of V_l results in U_l becoming too small to efficiently utilize the GPU for both contraction and query. Delling observes something similar: In the Western Europe graph less than 0.01% of the nodes are spread over half the levels [9].

To avoid this underutilization contraction stops once $V_l < K$, and the final overlay, $G_{l_{\max}}$, is converted into complete graph G_K using an APSP query. For the systems tested $K = 1025$ works well, in part because GPU APSP can be efficiently coded at that size [17]. The APSP used requires $O(V^3_{l_{\max}})$ work and a measured time of 20 ms on a graph with 1024 nodes.

3.7 Data Structures

A brief overview of the data structures used in CU-CH is provided here. Graphs in CU-CH use a layout similar to the *Compressed Sparse Row (CSR)* representation[6]. Graphs consist of arrays for edge weights and edge destinations, a node-ID-indexed array storing the indices into these edge arrays, and a node-ID-indexed array storing node degrees. To ensure alignment and to accommodate the addition of shortcuts overlay graph edge arrays are padded. The amount of padding is chosen to maintain 32 B alignment and to balance edge list size against time lost when shortcut space is exhausted. When a shortcut is to be added to a node with no remaining padding that node's edges are moved to an overflow area and the node array is adjusted accordingly. Note that unlike standard CSR, the representation used in CU-CH does not guarantee ascending order in edge and weight arrays. While using an adjacency list or DCSR [19] would facilitate adding shortcuts, they would have other overheads associated with them.

In different steps of CU-CH either the incoming or the outgoing edges or both are accessed. To maintain efficient memory bandwidth utilization in each of these access types, edge lists are ordered by both source (in a *forward* edge list) and destination (in a *backward* edge list).

Similar to PHAST, nodes and edges in $G_\uparrow$ and $G_\downarrow$ are ordered based on level, however they are padded differently than edge lists in the overlay graphs. The edge and weight arrays of multiple nodes belonging to the same level are packed into groups of ≤ 1024 edges and groups are padded to maintain 4 kiB (1024 element) alignment. These modifications improve data localization and allow contiguous memory access patterns within each active block during queries while ensuring different blocks can work independently. This layout also acts as a simple load balancing scheme between different blocks working on the same level.

3.8 CU-CH Query

The CU-CH query is similar to PHAST [9]. Unlike PHAST, the upward pass (and everything else) is performed on the GPU. For the first 31 levels the upward pass visits only nodes reachable from the source. Bit vectors are used to identify reachable nodes and reachable levels. The level vector is scanned until the next reachable level is found, then the node vector is scanned to find reachable nodes. Those nodes update their distances following backward upward edges, which reach lower-level nodes. Next, they update the vectors for higher-level nodes (those at the end of forward upward edges). The vectors are not used past level 31, instead all nodes within a level update their distances. Unlike PHAST, the last

Table 1: Benchmark Graphs

Graph	Description	# Nodes	# Edges	d	D
LA	USA / Louisiana	413,574	988,458	2.39	6
NY	USA / New York State	716,215	1,773,794	2.48	8
NW	USA / North Western US	1,207,945	2,840,208	2.35	8
NE	USA / North Eastern US	1,524,453	3,897,636	2.55	8
CAL	USA / California and Nevada	1,890,815	4,630,444	2.45	7
TX	USA / Texas	2,073,870	5,116,492	2.47	8

level is a complete graph, so when the upward pass reaches $l_{\max}$, distances are updated using the complete set of edges. Like PHAST, the downward pass visits all nodes. Like PHAST, the amount of work and data visited is $O(E_{\downarrow})$.

To provide the shortest path from each node to a source the CU-CH query computes the *parent* of each node (a neighbor on a shortest path to the source). Parents are determined using edge *midpoints*. The *forward midpoint* of $(u, v) \in E$ (an original edge) is u. The forward midpoint of shortcut (u, v) added for broken path (u, w, v) is the forward midpoint of (w, v). *Backward midpoint* is defined similarly. Midpoints are collected during contraction, and are computed in the APSP query on $G_{l_{\max}}$. In a query the midpoints are used to reconstruct the path after the upward and downward passes. (PHAST finds parents in a third pass by finding the minimum-distance neighbor of each node.)

4 METHODOLOGY

4.1 Benchmarks

The workload used to evaluate CU-CH and comparison codes consists of graphs from the DIMACS 9 competition and from the TIGERS database [1]. These graphs describe road networks in parts of North America, key details are given in Table 1. The graphs were prepared by removing duplicate and self edges and where necessary (TX) scaling weights to fit 32 b integers. Distance was used for edge weight.

4.2 Code Preparation, Hardware, and Timing

Code for the experiments reported here were prepared on hosts running Red Hat Enterprise Linux 7.6 using CUDA Toolkit Version 9.2, gcc 4.8.5, and BOOST 1.53. GPU results were collected on an HPC-grade, Pascal-generation GPU, the Nvidia Tesla P100. CPU results were run on an Intel Xeon E5-2640 v4 system. The Xeon has ten cores, a 2.4 GHz clock, and 25 MB of cache.

The time reported for GPU contraction starts with the graph in CPU memory and ends with the contracted graph and other items needed for query in GPU memory. These are based on wall time, with the end time taken after the GPU has completed pending work. The execution time for individual kernels was collected using CUDA event timers.

CU-CH is compared with CH as implemented by Geisberger's Contraction Hierarchies distribution, Version 2008/06/24 Revision 447 [15]. This CH code can be run with many options for scoring. Two sets of options are provided and they are used here. The *economical variant*, abbreviated *Geis-Eco*, provides fast preprocessing and medium query times. The *aggressive variant*, abbreviated *Geis-Agr*, provides medium preprocessing time and low query times. The economic variant hop-limits WPS, among other differences. Problems were encountered preprocessing smaller graphs using Geisberger's code, and data for those graphs is omitted from the results.

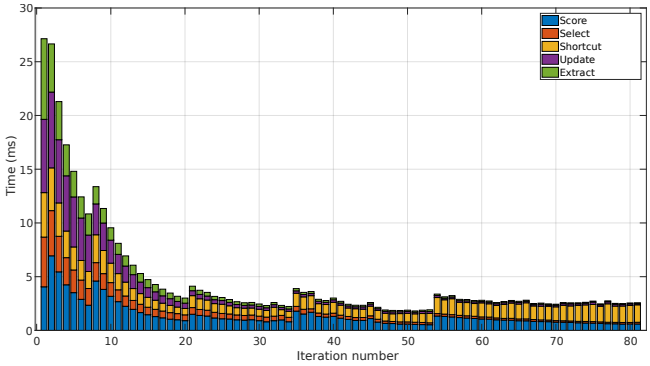


Figure 5: Time of each iteration for CAL with $C = .3$

5 RESULTS

5.1 Overall Performance

Table 2 shows the time needed for contraction of the benchmark graphs by CU-CH and Geis-Agr and Geis-Eco, and the speedup over Geis-Eco. For the Geisberger runs *contraction time* refers to the time for an ordinary CH, in which all but one node is contracted. For CU-CH contraction time includes both contraction until about 1024 nodes remain (at which contraction stops), and the time for an APSP query on them. CU-CH achieves 20× to 37× speedup over Geis-Eco, and even larger speedups over Geis-Agr. Even if Geisberger's code were perfectly parallelized for the 10-core CPU CU-CH would enjoy a substantial advantage.

The quality of the contracted graphs is shown in Table 2 by the number of levels and by average SSSP query time (averaged over 100 queries on randomly selected sources). For CAL graph, these stats are also reported for a hypothetical *CU-CH-true* implementation that uses an oracle WPS (further discussed in Section 5.3). The best results, meaning the fewest levels and smallest query times, are obtained with graphs contracted using Geis-Agr. Query times on the CU-CH graphs are between those of Geis-Agr and Geis-Eco on two out of three graphs for which data is available. CU-CH results in only a small drop in query performance in exchange for much faster contraction.

For comparison, the time needed for CPU SSSP queries using the BOOST library SSSP (Dijkstra) code is shown. An interesting question is the number of queries at which contraction plus query using CU-CH is faster than using BOOST to perform the queries. For NW and CAL 3 or more queries are faster using CU-CH, for TX even 2 queries are faster on CU-CH despite the time needed for contraction. SSSP queries were also tried using the Nvidia nvGRAPH library [23]. As with other GPU SSSP queries on road network graphs, the performance is poor and so comparisons are omitted.

Table 2: Benchmark results

Graph	Contraction Method	Contraction Time / ms	SpdUp	Num of Levels	Mean Query Time / ms
LA	CU-CH $C = .3$	242	-	40	2.496
	BOOST Dijkstra	-	-	-	93.459
NY	CU-CH $C = .3$	415	-	56	4.056
	BOOST Dijkstra	-	-	-	166.470
NW	CU-CH $C = .3$	589	20×	60	5.634
	Geis-Agr	34,969	.35×	47	4.790
	Geis-Eco	12,212	1×	172	6.230
	BOOST Dijkstra	-	-	-	278.180
NE	CU-CH $C = .3$	903	37×	92	7.659
	Geis-Agr	85,239	.40×	74	6.352
	Geis-Eco	33,775	1×	818	-
	BOOST Dijkstra	-	-	-	372.860
CAL	CU-CH $C = .3$	950	34×	82	8.541
	CU-CH-true $C = .3$			62	8.131
	Geis-Agr	81,347	.40×	63	7.089
	Geis-Eco	32,292	1×	292	9.773
	BOOST Dijkstra	-	-	-	477.830
TX	CU-CH $C = .3$	1,030	23×	85	9.319
	Geis-Agr	65,848	.36×	43	7.050
	Geis-Eco	23,965	1×	186	9.227
	BOOST Dijkstra	-	-	-	566.590

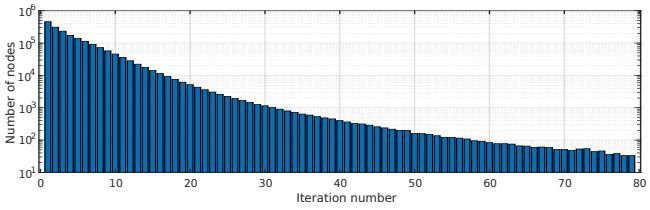


Figure 6: Contracted nodes per iteration for CAL with $C = .3$

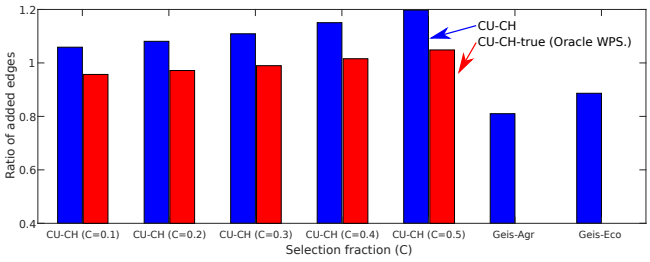


Figure 7: Edge growth, $(|E_{\uparrow}| + |E_{\downarrow}|)/|E|$, on CAL.

Figure 5 shows the time for each step per iteration and Figure 6 shows $|U_l|$ per iteration both for a contraction of CAL with $C = .3$. Ideally the time for iteration l would be $\propto |V|(1-\lambda)^l$, where $\lambda_l = |U_l|/|V_l|$ is the fraction of contracted nodes. That shape can be seen, though interrupted by jumps, for example, the jump after iteration 20. The jumps are due to increases in the number of threads assigned to a node, T_l, which is chosen based on the maximum degree. At an iteration l in which $T_{l-1} < T_l$ there will be many more idle threads in the WPS than there were in iteration $l-1$, because in a WPS $T_l - d_l$ threads are idle and most likely $d_{l-1}/T_{l-1} > d_l/T_l$. This drop in efficiency when T_l increases is painful but hard to avoid. For example, assigning fewer threads per node would reduce the amount of local storage available for WPS, which would increase the number of global memory accesses.

Apart from increases in T_l, three additional effects undermine the ideal execution time: an increase in the degree of the overlay graph, d_l, a reduction in λ_l, and a reduction in the absolute number of contracted nodes, $|U_l|$.

The increase in degree brings the time for the $O(|V_l|d_l^2)$ score and $O(|U_l|d_l^3)$ shortcut WPS to dominate, and that can clearly be seen. The WPS is discussed further in Section 5.3.

Decay in α_l with l, the size of the discovered independent set of U_{l+}, has two impacts. It reduces the number of contracted nodes, increasing the overlay size at some iteration l (over the case where α_l is unchanging) and so the modeled time for iteration l increases to $O(|V| \prod_{i<l}(1-\lambda_i))$, where $\lambda_l = \frac{1}{C\alpha_l}$.

A second impact occurs when the number of contracted nodes drops below the number that the device can compute in parallel. At that point the time for the Shortcut step is determined by the latency of the worst-performing thread, which increases with l due to growth of the maximum degree. In contrast the time for the Score step continues to decrease. At $T_l = 128$ the P100 can simultaneously execute 280 Shortcut operations. (Each node is handled by 128 threads, and due to shared memory constraints up to 6400 threads can be active on each of 56 SMs.) At iteration 50 there are $|U_{50}| = 160$ nodes to contract and so the device is underutilized. The P100 can handle up to 280 parallel Score steps, and with $|V_{50}| = 3271$ there is plenty of work and so Score time continues to drop. At $T_l = 256$ the P100 can execute Shortcut for 112 nodes in parallel and Score for 168 nodes in parallel, so Score fully utilizes the device to when the cut-off is reached. The cutoff, K, puts an end to the Shortcut underutilization. The rationale for $K = 1025$ includes a variety of factors, and will be revisited in the future.

5.2 Impact of Selection Fraction (C)

Selection fraction, C, is an important tuning parameter. For larger values more contraction candidates, U_{l+}, are selected, hopefully reducing $l_{\max}$ and the average number of times each node is scored, $\frac{1}{\alpha C}$ in the ideal analysis. But with larger U_{l+} the scores are less relevant, resulting in overlay graphs with more shortcuts than necessary. This in turn increases the time needed for a WPS and query, and reduces the size of the discovered independent sets, $\alpha_l = |U_l|/|U_{l+}|$, which increases $l_{\max}$.

The measured impact of C on the number of shortcuts in CAL is plotted in Figure 7, as well as the ratio of added shortcuts for the Geis-agr and Geis-eco runs. Even at the most patient setting, $C = .1$, Geisberger adds many fewer edges. The impacts on degree and α are shown in Figures 3 and 4, both figures show $l_{\max}$. Here it can be seen that for larger overlays the product $C\alpha$ increases with C, and so the number of levels, $l_{\max}$, drops with increasing C, a benefit, but one that can be undermined by the degree growth. Figures 8 and 9, which show contraction time, indicate that degree growth becomes a problem only when C reaches .5.

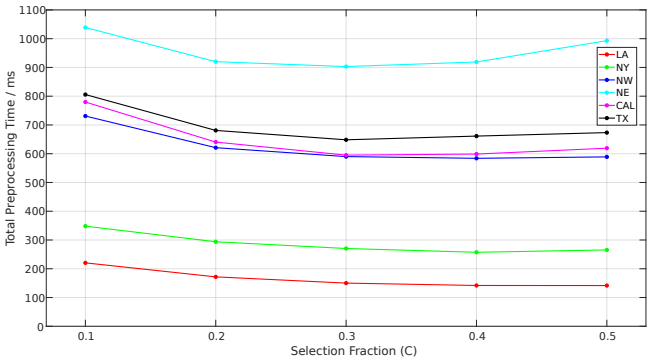


Figure 8: Total preprocessing time v. C.

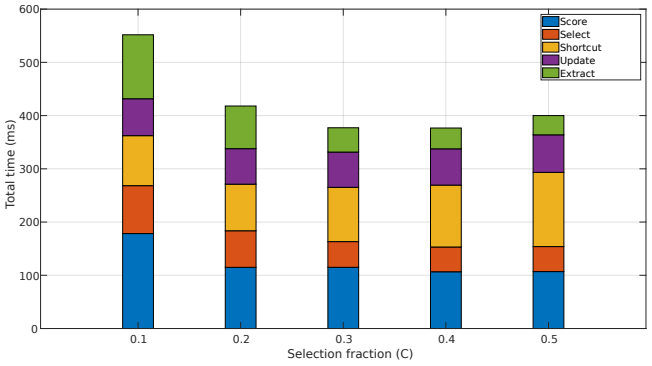


Figure 9: Time for each step v. C for CAL.

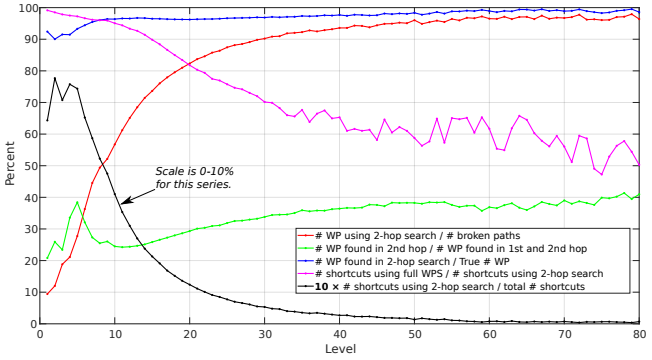


Figure 10: Ratio of # WP and shortcuts on CAL with $C = .3$.

Finally, Figure 11 shows the impact of C on query time. Intermediate values of C do best, though further tuning may improve the performance at larger values.

5.3 Effectiveness of Witness Path Search

The CU-CH WPS executes efficiently (and quickly) but stops at 2-hops. To determine the impact of this limitation CU-CH is compared to CU-CH-true, a version of CU-CH that uses an oracle (a full WPS performed on the CPU) in the Shortcut step. Figure 7 shows the number of edges in the output graphs generated by CU-CH-true, CU-CH, Geis-Agr, and Geis-Eco operating on CAL. For $C = .3$ there would be 11% fewer edges using the full WPS. The impact on query time for CAL is found in Table 2: using the full WPS provides only a small improvement in query time, from 8.5 ms to 8.1 ms.

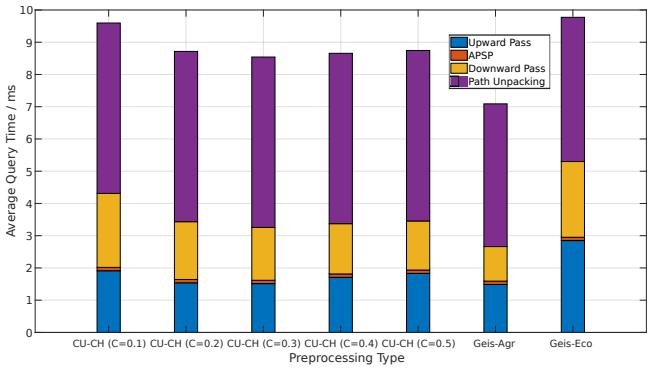


Figure 11: Query time of CAL contracted various ways.

Further details on the WPS effectiveness appear in Figure 10, which shows per-level comparisons for CAL. The blue series shows the ratio of the number of WP found by the 2-hop and full searches. By this measure the 2-hop search is performing well, at least 90%. However the number of broken paths increases quadratically with degree, and that results in an increase in the number of unneeded shortcuts, shown by the purple series which shows the ratio of the number of shortcuts added using the full search over the number added using the 2-hop search. This starts out at nearly 100% but sinks to 50%. Fortunately the absolute number of shortcuts added drops with level, shown in the black series (scaled by a factor of 10 to improve readability). The black series shows the number of shortcuts added in the level over the number of shortcuts in the completed contraction. In later levels that number drops to well below 1%, showing the small impact of the unneeded shortcuts revealed by the purple series. The green series shows the ratio of WP found in the 2nd hop to those found in both hops. Over 60% are found in the 1st hop, which is why Score uses a 1-hop search.

5.4 Query

Table 2 shows total query times for the benchmark graphs. To provide more detail, Figure 11 shows a breakdown of the kernel times on a query of CAL operating on graphs prepared by CU-CH using different values of C and graphs prepared using Geis-agr and Geis-eco. The time for updating the overlay results and the time for unpacking the shortcuts in the path are unaffected by the preprocessing method since they are only dependent on K and V respectively.

Figure 12 shows a breakdown of query times per level. Figure 13 shows the number of edge groups (Section 3.7) per each level. There is a trade-off between total number of edges, number of edge groups per level and the number of levels in query performance. On one hand, the query time is dependent on the total number of edges since in PHAST, each edge and each node is touched only once. On the other hand however, because parallelism in PHAST is limited to levels, having more levels can result in a larger kernel launch

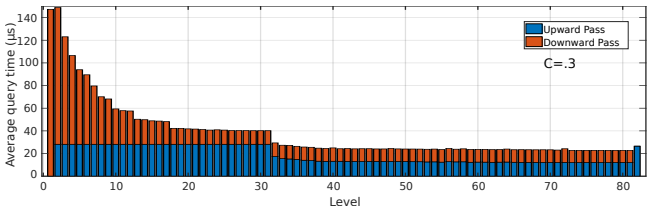


Figure 12: Query time by level of CAL.

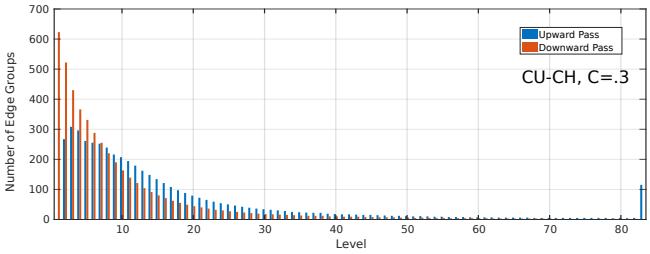


Figure 13: No. edge groups of < 1024 per level on CAL.

overhead which is more than 4 μs on our benchmark platform for a blank kernel.

On top of that, in levels where the total number of edge groups is fewer than the available SMs, there is reduced hardware utilization. Note that under-utilization on thread levels is already addressed by using edge groups. This second mechanism is the reason why the graph prepared using the economic Geisberger implementation has a larger query time despite having fewer added shortcuts.

This trade-off between $l_{\max}$ and $|E_{CU-CH}|$ is the reason why the query times decrease on the CAL graph when C is increased from .1 to .3 and why they see a slight increase when C is further increased to .5.

6 PRIOR WORK

Davidson *et al* [8] present well-performing GPU SSSP algorithms. A thread retrieves a node from a *current* work queue, with other threads updates the distances of its neighbors, and places any neighbors that change in one of several *future* work queues chosen based on distance. In the best of these schemes there are two future work queues, *near* and *far*. When the current queue is empty it is replenished from the near queue. If the near queue is empty the choice threshold is updated and used to redistribute nodes in the near and far queues. The technique works well for most graphs, achieving over 10× speedup over CPU code for four of eight graphs, but yielding a slowdown for the two road-network graphs tried. The poor performance of the road network graphs was attributed to the small workfront (occupancy of current queue) [8]. Pai *et al* reduce the small-workfront impact in part by moving some iteration to GPU code [25], improving execution time by about 6×.

There is a substantial amount of work on accelerating point-to-point queries in road network graphs by operating on a preprocessed graph, these are surveyed in [4]. They include Geisberger's CH [15] and Delling's adaptation to SSSP queries [9] discussed in Section 2.4. Work on reducing the time to compute CH has been done for cases where the topology remains fixed but the edge weights change. Dibbelt *et al* describe *Customizable Contraction Hierarchies (CCH)* in which the pre-processing is divided into a time-consuming weight-independent phase and a fast weight-dependent phase [10]. In contrast, CU-CH applies when no version of the graph is available in advance.

The *Highways on Disk (HoD)* scheme described by Zhu *et al* [29] has some interesting parallels with CU-CH and PHAST. HoD is intended for systems in which there is enough high-speed storage for the nodes, but not enough all edges. (The high-speed storage was cache-backed DRAM, with the alternative being a much slower magnetic disk.) Like CU-CH nodes are assigned to levels (rather than being ranked first) though using a simpler scoring than CH. Like PHAST and CU-CH nodes within a level cannot be neighbors, and like CU-CH level-assignment stops when a size threshold is reached, the remaining nodes form the *core*. Shortcut candidates and selected paths are sorted to find witness paths. While their work was an inspiration for CU-CH, it addresses the very different challenges arising when graphs cannot fit in high-speed storage.

There are a number of frameworks for implementing graph algorithms on GPUs which facilitate data layout and graph traversal [18, 27, 28]. One framework, CuSha [18], helps re-structure data to fit a GPUs memory hierarchy. For their SSSP implementation they report a run time of 384 ms on a GTX 780 for the roadnet-CA graph (road network of California with 1.97 million nodes and 5.53 million edges).

7 CONCLUSION

In this paper we presented CU-CH, a generalized parallel CH algorithm and CUDA implementation for road networks. Solutions for some of the challenges associated with efficient use of the GPU's memory and computation hierarchy to allow CU-CH were covered. It was shown that by applying an independent set constraint to the set of nodes contracted in parallel, the search for the necessary set of shortcuts to maintain the CH definition can be limited to the immediate neighborhood of each node being contracted. This was followed by describing a simple yet effective shallow bidirectional breadth-first search of a given node's neighborhood that could be placed in the shared memory. Thus performing witness path searches efficiently on the GPU. A modified implementation of PHAST was also briefly introduced to perform SSSP queries on graphs prepared by CU-CH. By performing an APSP on the highest ranked nodes, hardware underutilization in both the later iterations of CU-CH as well as the highest levels of PHAST is avoided, improving performance.

The simple witness path search method shows results within 20% to 30% of the graphs prepared by Geisberger's original implementation of CH in terms of number of added shortcuts. In terms of preprocessing times, CU-CH performs 20 to 94 times faster than Geisberger's serial CPU implementation. Finally, the SSSP query times using the modified implementation of PHAST, showed similar performance to those prepared by Geisberger's implementation ranging between 25% slower to 20% faster depending on benchmark conditions.

ACKNOWLEDGMENTS

This work has been funded by the U.S. Naval Research Laboratory Base Program under award N00173-16-2-C901. Equipment was funded in part by the National Science Foundation under award ACI-1265449 and the Louisiana Board of Regents under award LEQSF(2015-16)-ENH-TR-10.

REFERENCES

[1] 2006. 9th DIMACS implementation challenge - shortest paths. Retrieved September 27, 2018 from http://www.dis.uniroma1.it/challenge9/download.shtml

[2] Ittai Abraham, Daniel Delling, Amos Fiat, Andrew V Goldberg, and Renato F Werneck. 2011. VC-dimension and shortest path algorithms. In *International Colloquium on Automata, Languages, and Programming*. Springer, 690–699.

[3] Ittai Abraham, Daniel Delling, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. 2016. Highway Dimension and Provably Efficient Shortest Path Algorithms. *J. ACM* 63, 5, Article 41 (Dec. 2016), 26 pages. https://doi.org/10.1145/2985473

[4] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. 2016. *Route Planning in Transportation Networks*. Springer International Publishing, Cham, 19–80. https://doi.org/10.1007/978-3-319-49487-6_2

[5] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. 2015. Route Planning in Transportation Networks. *CoRR* abs/1504.05140 (2015). arXiv:1504.05140 http://arxiv.org/abs/1504.05140

[6] Nathan Bell and Michael Garland. 2008. *Efficient Sparse Matrix-Vector Multiplication on CUDA*. NVIDIA Technical Report NVR-2008-004. NVIDIA Corporation. http://www.nvidia.com/object/nvidia_research_pub_001.html

[7] Richard Bellman. 1958. On a routing problem. *Quarterly of applied mathematics* 16, 1 (1958), 87–90.

[8] A. Davidson, S. Baxter, M. Garland, and J.D. Owens. 2014. Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. 349–359. https://doi.org/10.1109/IPDPS.2014.45

[9] Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. 2011. PHAST: Hardware-Accelerated Shortest Path Trees. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS '11)*. IEEE Computer Society, Washington, DC, USA, 921–931. https://doi.org/10.1109/IPDPS.2011.89

[10] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. 2016. Customizable Contraction Hierarchies. *J. Exp. Algorithmics* 21, Article 1.5 (April 2016), 49 pages. https://doi.org/10.1145/2886843

[11] E. W. Dijkstra. 1959. A Note on Two Problems in Connexion with Graphs. *Numer. Math.* 1, 1 (Dec. 1959), 269–271. https://doi.org/10.1007/BF01386390

[12] David Eppstein and Michael T. Goodrich. 2008. Studying (Non-planar) Road Networks Through an Algorithmic Lens. In *Proceedings of the 16th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS '08)*. ACM, New York, NY, USA, Article 16, 10 pages. https://doi.org/10.1145/1463434.1463455

[13] David Eppstein and Siddharth Gupta. 2017. Crossing Patterns in Nonplanar Road Networks. In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL '17)*. ACM, New York, NY, USA, Article 40, 9 pages. https://doi.org/10.1145/3139958.3139999

[14] Lester R Ford Jr. 1956. *Network flow theory*. Technical Report. RAND CORP SANTA MONICA CA.

[15] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. 2008. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *Proceedings of the 7th International Conference on Experimental Algorithms (WEA'08)*. Springer-Verlag, Berlin, Heidelberg, 319–333. http://dl.acm.org/citation.cfm?id=1788888.1788912

[16] Y. Jia, V. Lu, J. Hoberock, M. Garland, and J. C. Hart. 2011. Edge v. Node Parallelism for Graph Centrality Metrics. In *GPU Computing Gems Jade Edition* (1st ed.), Wenmei W. Hwu (Ed.), Vol. 2. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 15–30.

[17] Gary J Katz and Joseph T Kider Jr. 2008. All-pairs shortest-paths for large graphs on the GPU. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*. Eurographics Association, 47–55.

[18] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. 2014. CuSha: Vertex-centric Graph Processing on GPUs. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing (HPDC '14)*. ACM, New York, NY, USA, 239–252. https://doi.org/10.1145/2600212.2600227

[19] James King, Thomas Gilray, Robert M Kirby, and Matthew Might. 2016. Dynamic sparse-matrix allocation on GPUs. In *International Conference on High Performance Computing*. Springer, 61–80.

[20] M Luby. 1985. A Simple Parallel Algorithm for the Maximal Independent Set Problem. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing (STOC '85)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/22145.22146

[21] Saeed Maleki, Donald Nguyen, Andrew Lenharth, María Garzarán, David Padua, and Keshav Pingali. 2016. DSMR: a shared and distributed memory algorithm for single-source shortest path problem. *ACM SIGPLAN Notices* 51, 8 (2016), 39.

[22] Ulrich Meyer and Peter Sanders. 2003. Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms* 49, 1 (2003), 114–152.

[23] NVIDIA Corporation. 2018. *nvGRAPH library user's guide*. NVIDIA Corporation.

[24] NVIDIA Corporation. 2018. *NVIDIA CUDA C Programming Guide V 9.2* (v 9.2 ed.). NVIDIA Corporation.

[25] Sreepathi Pai and Keshav Pingali. 2016. A Compiler for Throughput Optimization of Graph Algorithms on GPUs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 1–19. https://doi.org/10.1145/2983990.2984015

[26] Christian Sommer. 2014. Shortest-path Queries in Static Networks. *ACM Comput. Surv.* 46, 4, Article 45 (March 2014), 31 pages. https://doi.org/10.1145/2530531

[27] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 11.

[28] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and John D. Owens. 2017. Gunrock: GPU Graph Analytics. *ACM Transactions on Parallel Computing* 4, 1, Article 3 (Aug. 2017), 49 pages. https://doi.org/10.1145/3108140

[29] Andy Diwen Zhu, Xiaokui Xiao, Sibo Wang, and Wenqing Lin. 2013. Efficient Single-source Shortest Path and Distance Queries on Large Graphs. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '13)*. ACM, New York, NY, USA, 998–1006. https://doi.org/10.1145/2487575.2487665