Customizing Driving Directions with GPUs*

Daniel Delling[1], Moritz Kobitzsch[2], and Renato F. Werneck[1]

[1] Microsoft Research
{dadellin,renatow}@microsoft.com
[2] Karlsruhe Institute of Technology
kobitzsch@kit.edu

Abstract. Computing driving directions interactively on continental road networks requires preprocessing. This step can be costly, limiting our ability to incorporate new optimization functions, including traffic information or personal preferences. We show how the performance of the state-of-the-art customizable route planning (CRP) framework is boosted by GPUs, even though it has highly irregular structure. Our experimental study reveals that our method is an order of magnitude faster than a highly-optimized parallel CPU implementation, enabling interactive personalized driving directions on continental scale.

1 Introduction

The past decade has seen intense research on the computation of driving directions in road networks [2, 20]. This problem can be modeled as computing shortest paths on a weighted graph and solved by classical algorithms such as Dijkstra's [4]. For continental road networks (with tens of millions of arcs), however, queries can take seconds, which is too slow for interactive applications. To overcome this, modern specialized algorithms [1, 3, 12, 14] generally work in two phases: a *preprocessing stage* precomputes some auxiliary data, which is then used to answer on-line *queries*. The fastest algorithms [1, 3] answer queries in microseconds or less after a few minutes of preprocessing on a standard server.

Such queries are certainly fast enough, but since preprocessing must be rerun whenever arc weights change, these methods do not support dynamic scenarios such as real-time traffic. The recent *customizable route planning* (CRP) algorithm [7, 10] (see also [8]) offers a different trade-off by working in *three* phases. The initial *preprocessing* phase is *metric-independent*: it takes as input only the graph topology. The *customization* phase takes as input the cost function (metric) and the output of the previous phase. Finally, *queries* use the outputs of both phases to compute point-to-point shortest paths. Queries are just fast enough (milliseconds rather than microseconds) for interactive applications, but a new cost function can be incorporated in mere seconds (by running only the customization phase), enabling CRP to handle frequent traffic updates. The algorithm is currently used by Bing Maps to compute driving directions.

We investigate how we can use GPUs to accelerate customization even further. Our approach is to set up all necessary data structures on the GPU during the metric-independent preprocessing, such that we only need to invoke a few GPU kernels when

* The second author worked on this project while at Microsoft Research.

a metric change occurs. This enables a degree of personalization well beyond what is available in current systems. Most notably, one could define a cost function at query time and still obtain driving directions in a fraction of a second. At first sight, computing driving directions is not a natural application for GPUs. Through careful engineering, however, we can harness the power of GPUs to make customization not only faster, but also more energy-efficient than CPU-based (even multicore) implementations.

We are not aware of previous work that uses GPUs to process dynamic continental road networks effectively. PHAST [6] can efficiently answer one-to-all (rather than point-to-point) queries on a GPU, but only after heavy CPU-based *metric-dependent* preprocessing, and is thus not dynamic. Parallelizing a single shortest-path computation on sparse and high-diameter graphs (such as road networks) is generally hard [16, 18] even on multicore CPUs. It is even harder on GPUs [5, 17, 19], since access patterns and operations are far from regular. We get around this issue by parallelizing more than a single shortest-path computation.

2 Preliminaries

The standard representation [2] of a road network is as a directed graph $G = (V, A)$, where each vertex $v \in V$ represents an intersection (junction) and each arc $a \in A$ represents a (directed) road segment. A *cost function* (or *metric*) $\ell : A \to \mathcal{N}$ maps each arc $a \in A$ to a positive *cost* (or *length*) reflecting the effort to traverse it. We use a more realistic model that incorporates turn costs and restrictions. The customization phase takes as input an *expanded graph* where vertices correspond to the heads of original arcs, and arcs are the concatenation of an original turn and an original arc. For queries and to store the graph in main memory, we use a more compact representation [7, 13].

Each original arc in our application is modeled as a collection of *static properties*, such as physical length (in meters), road category (freeway, local road, or ferry, for example), number of lanes, and speed limit. Similarly, each turn has a *type* (left turn, right turn, and so on). A *metric decoder* is a function that maps these properties to the cost of traversing the arc or making the turn. We could model special cases (such as traffic) by storing costs explicitly for some exceptional arcs. We assume all costs are integral and that the length of any shortest path fits in 32 bits.

A *path* in the graph is a sequence of arcs of the form (v_0, v_1), (v_1, v_2), (v_2, v_3), $\ldots$, (v_{k-1}, v_k). The *cost* (or *length*) of a path is the sum of the costs of its arcs and the turns between them. The *point-to-point shortest path* problem takes as input the graph $G = (V, A)$ and two arcs a_s and a_t, and returns the shortest (minimum-length) path that starts at a_s and ends at a_t in G.

Customizable Route Planning. The *preprocessing phase* of CRP starts by computing a nested L-level partition. A partition of V is a collection of *cells* such that each vertex $v \in V$ belongs to exactly one cell. A nested L-level partition of V is a family of partitions such that, for any level $i < L$ and each cell C, there exists a cell C' on level $i+1$ that contains C; we say that C is a subcell of C'. (For simplicity, define a level-0 partition consisting of singletons.) CRP uses the PUNCH [9] graph-partitioning algorithm to generate an L-level partition top-down, partitioning level L first, then (recursively) each

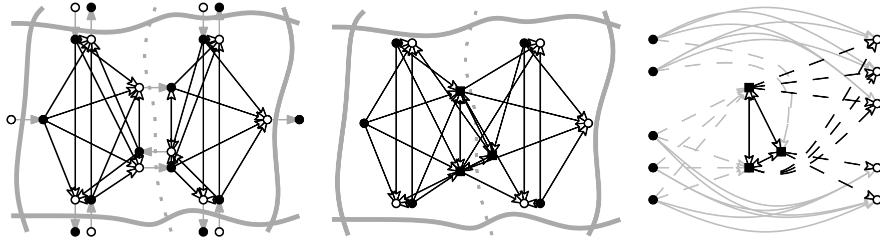

Fig. 1. A cell C with the overlays of its two subcells (left), its compact graph (center), and the abstracted subgraph with entry (filled circles), inner (squares), and exit (hollow circles) vertices.

cell thus created. For each level l, PUNCH finds cells with no more than U_l (an input parameter) vertices and minimizes the number of boundary arcs between cells.

The CRP preprocessing phase also sets up the topology of a multilevel overlay graph [15]. Figure 1 (left) shows a cell C on some level. Every incoming boundary arc (u,v) (with $u \notin C$ and $v \in C$) corresponds to an *entry vertex* for C, and every outgoing boundary arc (v,w) (with $v \in C$ and $w \notin C$) defines an *exit vertex* for C. The overlay of a cell is the complete bipartite graph with directed *shortcuts* (black arrows in the figure) between its entry (filled circles) and exit (hollow circles) vertices. The overlay of level l is the union of all cell overlays and boundary arcs (gray arrows) on this level.

The CRP *customization phase* computes the costs of all shortcut arcs on the overlay. It computes, for each cell C, the distances between each entry vertex and each exit vertex: a shortcut (p,q) in C represents the shortest p–q path restricted to C. The algorithm processes cells bottom-up, starting at level one; when processing a level-l cell C, it works on the (small) overlay graph for level $l-1$. One could simply run Dijkstra's algorithm from each entry vertex until all exit vertices are scanned [7], but one can do better (even on CPUs [10]) using Bellman-Ford [4] or contraction [12]. Sections 3 and 4 detail these approaches and show how they can be effectively realized on the GPU.

A point-to-point CRP *query* runs bidirectional Dijkstra on the overlay graph, but only entering cells that contain either the source s or the target t.

3 Search-Based Customization

The main subroutine of the CRP customization phase computes arc lengths of bipartite graphs. The fastest [10] approach based on graph traversal is based on the classical Bellman-Ford algorithm. To process a cell C at level i, it first builds a compact graph $G_C = (V_C, A_C)$ consisting of the shortcuts and boundary arcs on level $i-1$ that are in C, but with the head vertices of the internal boundary arcs (i.e., those not on level i) removed and their incident arcs concatenated. See Figure 1 (center). Let N_C and X_C be the sets of entry and exit vertices of C, respectively. The algorithm maintains, for each vertex in V_C, a *distance array* with $|N_C|$ values; the k-th position for the k-th entry vertex is initialized with zero, and all others with infinity. Then it runs Bellman-Ford as long as there is an improvement on any of the distance labels. Eventually, the distance labels of the exit vertices will reflect their distances from each of the entry vertices.

Basic Algorithm. On small diameter graphs, Bellman-Ford works well on GPUs [5], but we can make it even more efficient for our purposes. We can classify the vertices in V_C into three categories: entry (N_C), exit (X_C), and inner (I_C). Figure 1 (center) shows that entry vertices have only outgoing arcs, and exit vertices only incoming arcs. Moreover, there are four types of arcs in A_C (illustrated in Figure 1 (right), obtained by rearranging Figure 1 (center)). The *init* arcs A_C^j (dashed gray) link entry to inner vertices, the *inner* arcs A_C^i (solid black) connect inner vertices to one another, the *collection* arcs A_C^c (dashed black) link inner to exit vertices, and the *mapping* arcs A_C^m (solid gray) link entry to exit vertices. Note that init and mapping arcs are shortcuts, while each inner or collection arc is the concatenation of a shortcut and a cut arc (all from level $i-1$). When running on G_C, Bellman-Ford touches each mapping and init arc only once, at which point it sets exactly one distance value at its head vertex. We can exploit this.

For a cell C, let $G_C^i = (I_C, A_C^i)$ be its *inner graph* and $V_C^c = (X_C, A_C^c)$ be its *collection graph*. In general, on level i, we compute the costs of shortcuts on level i (to be stored in a *shortcut array* S_i) from costs of level-$i-1$ shortcuts and boundary arcs (stored in a *boundary array* B). Our algorithm processes a cell in five phases. The *mapping phase* copies the costs of the mapping arcs from S_{i-1} to S_i. The subsequent *aggregation* phase computes the costs of the inner arcs from S_{i-1} and B. The third phase (*injection*) copies the init arc costs from S_{i-1} into the distance array (which now has size $|N_C| \cdot |I_C|$). The fourth phase, *search*, runs Bellman-Ford on the inner graph, stopping when there is no improvement. The final *collection* phase first aggregates the costs of the collection arcs (as in the aggregation phase); then, for each exit vertex v, it iterates over its incoming collection arcs to compute the costs of the level-i shortcuts ending at v, updating S_i. We propose two GPU implementations of this approach: global and local.

Global Implementation. The global implementation is orchestrated by the CPU and invokes multiple kernels per level i. We maintain one global distance array representing the distance values of all inner vertices of all cells on level i.

For each of the first three phases of customization (mapping, aggregation, and injection), we create a single kernel with one thread for each relevant arc. We support these threads by maintaining *auxiliary arrays* with the relevant information in global memory; thread t reads position t from this array. For aggregation, we arrange the data in global memory such that threads also write their output to consecutive positions.

More precisely, the *mapping* phase has one thread per mapping arc: it uses the auxiliary array to learn the position it must read from (in S_{i-1}) and write to (in S_i). During the *aggregation* phase, thread t computes the length of inner arc t; the corresponding auxiliary array contains the positions in B_{i-1} and S_{i-1} the thread must read from. Similarly, *injection* has one thread per init arc, and its auxiliary array stores a position in S_{i-1} (for reading) and another in the distance array (for writing).

The *search* phase uses one thread per distance value. Recall that we have one distance per pair (inner vertex, entry vertex). A thread processes all incoming arcs for a fixed inner vertex v and updates a single distance value (for one entry vertex). The corresponding index array contains the vertex ID it has to process, as well as an index (a number from 0 to $|N_C|-1$) indicating which of its distances to update. This information can be packed into 32 bits. Also, rather than storing the tail ID, an arc stores the position

of the first distance of its tail; the thread then uses the index as an offset. Since global synchronization is required, each Bellman-Ford iteration runs as a single kernel. Each thread writes to a timestamp array (indexed by cell number) whenever it updates some value; Bellman-Ford stops after an iteration in which this array does not change.

The *collection phase* is similar to the search phase, but operates on the exit vertices and is limited to one round. Moreover, it stores its information directly to S_i. To make these accesses more efficient, shortcuts are ordered by tail in global memory.

Note that our implementation has no write-write conflict. During Bellman-Ford, a thread may read a position that is being updated by another. Since integers are atomic and values only improve from one round to the next, this does not affect correctness.

Local Implementation. The local implementation invokes one kernel per level and operates block-wise. For simplicity, we first describe our algorithm assuming it processes one cell per thread block, then generalize it. Since we no longer have one thread for each value we deal with, we use a small *block header* to store relevant information the threads require. It includes the numbers of all types of arcs (mapping, injection, inner, and collection) and vertices (inner, entry, and exit). It also has pointers to the positions in global memory where we store the topology of the inner and collection graphs.

The algorithm starts by reserving space in shared memory for the distance values it will compute (initialized with ∞). The mapping phase is exactly as before. The *aggregation phase* is also similar, but stores the values in shared memory; it also copies the inner graph topology to shared memory. Similarly, *injection* works as before, but writing into the distance array in shared memory. The *search phase* now operates entirely in shared memory and uses the GPU block-based synchronization between Bellman-Ford rounds. Note that thread t (within the block) can deduct from the block header both the vertex it has to work on ($\lfloor t/|I_C| \rfloor$) and the entry vertex number ($t \bmod |I_C|$). The *collection phase* first copies the collection graph to shared memory (overwriting the inner graph, which is no longer needed), aggregating the costs of the collection arcs. It then performs a single Bellman-Ford round and stores the final distances to global memory. We use global memory as fallback if any of these phases does not fit in shared memory.

We use 16 bits for indexing; if that is not enough for a given cell, we process the entire level using the global implementation instead. This happens only very rarely, and can usually be avoided by optimizations we introduce later.

Since we know in advance how much shared memory each cell occupies, we can often group multiple cells into the same block. We reorder the cells in GPU memory to ensure their shortcuts appear consecutively. For regularity, we only group cells that have the same number of entry vertices. The algorithm works exactly as before: it just sees the input as a bigger, disconnected graph.

Comparing the local and global approaches, the latter is more space-consuming, since it needs to store additional data for each distance value in global memory. It is still a good option when there are few cells or when graphs are too large to fit in shared memory. We thus use the global implementation on levels with fewer than 100 cells (about 6 times the typical number of multi-processors of current GPUs), or when the number of collection or inner arcs exceeds 65 536, the maximum number the local approach can index with 16 bits.

We have been assuming that we can use a level-$i-1$ overlay to compute the overlay of level i, but this is not true for the first level, when we must operate on the underlying original graph. We can handle this by adapting the routine that aggregates arc costs. Mapping and init arcs represent an original graph arc, and all other arcs are a concatenation of a turn and an original arc. Therefore, for a mapping or init arc, we store its physical properties (rather than a position in S_{i-1}); for other arcs, we store the turn type as well. In all cases, we apply the current metric decoder during aggregation.

An important optimization is to use *mezzanine levels* [10], partition levels that are used to accelerate customization, but discarded for queries (to save space). Mezzanine levels help reduce the size of inner graphs (which are expensive to deal with) by turning more arcs into init, mapping, or collection arcs (which are accessed a constant number of times). This reduces the number of Bellman-Ford iterations, our main bottleneck. Mezzanine levels are not free, though: there is some overhead for mapping the extra levels, but this is very cheap on the GPU (not so on CPUs [10]). Moreover, they increase both the number of cells and the space consumption on the GPU. Note, however, that we can overwrite shortcut weights for mezzanine level i as soon as level $i+1$ is processed.

4 Contraction-Based Customization

For lower levels of the hierarchy, customization is faster [10] if one uses *graph contraction* instead of graph searches (Dijkstra or Bellman-Ford). We first recap how the CPU-based approach works on the CPU, then explain how it can be adapted to the GPU.

When processing a cell C on the CPU, we can compute the lengths of the shortest paths (in G_C) from its entry vertices to its exit vertices using the *shortcut* operation [12]. Shortcutting an inner vertex v means removing it from the graph and, for each incoming arc (u,v) and outgoing arc (v,w), creating a *shortcut arc* (u,w) with length $\ell(u,w) = \ell(u,v) + \ell(v,w)$. If (u,w) does not yet exist, we insert it; if it does, we update its length if the new arc is shorter. By repeatedly applying this operation to all inner vertices in G_C, one ends up with a bipartite graph with arcs between entry and exit vertices of C, where arc lengths represent the corresponding distances (missing arcs represent infinite distances). Any contraction order leads to the same final topology, but a carefully chosen (during preprocessing) order based on nested dissections leads to fewer operations overall and a faster algorithm [10].

The fundamental operation of contraction is to read the costs of two arcs, add them up, compare the result with the cost of a third arc, and update its cost if needed. Instead of using a graph during customization, Delling and Werneck [10] propose simulating the contraction process during preprocessing to create an *instruction array* representing these fundamental operations (*microinstructions*) compactly as triples (a,b,c), where a and b are the positions to be read and c the position to write to. These positions refer to a *memory array* M and correspond to arc costs. Each cell C has its own instruction and memory arrays. Moreover, they use an *arc instruction* array to initialize M.

Building the GPU Microinstructions. Microinstructions provide a natural starting point for implementing contraction-based customization on the GPU. Although the microinstruction array can be fairly large, it is only read once (and sequentially), so we

keep it in global memory. Since M is much smaller and has a less rigid access pattern (each position can be accessed multiple times), we keep it in shared memory. For optimal performance, however, we must address several issues: decreasing the space used by microinstructions (for fewer accesses to slower memory), reducing the memory array (to keep multiple cells in shared memory at once), and parallelization within a cell (for efficiency on GPU). We do so by preprocessing and enriching the microinstructions before copying them to the GPU (the arc instructions can be copied essentially as is).

First, we make the microinstructions more compact. Since each entry in the memory array M takes 32 bits of shared memory, it can have at most 12 288 positions in the GPUs we test. These can be addressed with 14 bits, or 42 bits per triple in the instruction array. For most cells, however, 32 bits are enough. To achieve this, we first ensure that $a < b$ in each instruction triple (a,b,c) (we swap a and b otherwise), then store the triple $(a,b-a,c-b)$ using 14, 8, and 9 bits, respectively (we reserve the 32nd bit for later). This means a can be any position in shared memory, b can refer to positions $a+1$ to $a+256$, and c can refer to $b-256$ to $b+255$. If a cell has at least one instruction that cannot use this compact representation (with b too far from a or c too far from b), we use a full 48-bit representation for all of its microinstructions.

To parallelize within a cell, we group independent instructions by layers. Note that two instructions in a cell are independent if they do not write to the same memory position. We create these layers by working in rounds, from latest to earliest, greedily assigning instructions to the latest possible layer (after accounting for the dependencies between them); we then apply a postprocessing step to make the layers more balanced.

Next, we reduce the memory array. Once a shortcut is eliminated by the contraction routine, the memory position that stores its cost could be reused for another shortcut, thus saving on shared memory. We identify such reusage opportunities during preprocessing as follows. We process the layered microinstructions from earliest to latest. We interpret each entry in a triple (a,b,c) as a *shortcut* (rather than positions in M, which is what we are trying to determine). We keep counters of pending reads and writes for each shortcut and a candidate pool of free memory positions (initially, all but those used by the arc instructions); when a read counter becomes zero for some shortcut, we add its position to the pool for potential reuse in future layers. When processing an instruction (a,b,c) that writes to shortcut c for the first time, we assign c to the free position that is closest to b; in addition, we use the 32nd bit (mentioned above) to *mark* this instruction, indicating that the GPU must simply write to the target position (ignoring the value already there) when executing this instruction. As an optimization, if an instruction (a,b,c) performs the last read from a (or b) and the first to c, we can immediately assign c to a's (or b's) position. If after running this basic algorithm the new instructions still cannot be represented in compact form (32 bits), we perturb the positions of the original arcs and retry; this is cheap and helps in some cases. Since the final shortcuts do not necessarily have consecutive positions in M, we use a map to translate them to the corresponding (consecutive) positions in S_1, the shortcut array on level 1. Note that we use microinstructions only to compute the shortcuts on the lowest level.

Finally, for better block utilization, we greedily pack cells as long as their combined memory arrays fit in shared memory. For better memory access patterns, we do not mix compact and full cells. We prefer to group cells with the same number of layers within

a block, but we may combine blocks with different depth if needed. When we finally store the instruction array on the GPU, we reorder it to reflect the block assignments: instructions within the same block are sorted by layer (across cells). Since the GPU must synchronize between layers, we store layer sizes in the block header.

GPU Execution. With the data structures set up, we compute S_1 on the GPU as follows. We invoke one kernel for the full computation, since synchronization is only needed within a block. On each block, we first run the arc instructions. The block header stores the number of arc instructions in each of its cells; each thread can use this information (and its own ID) to determine where in shared memory to store the result of the arc instruction it is responsible for. We then execute the microinstructions, layer by layer, also with one thread per instruction. Finally, we map the costs of the output shortcuts to S_1, using one thread per value. For each block, we store its first position in S_1, allowing each thread to determine (using its own ID) where to write to.

5 Putting Everything Together

During the metric-independent phase of CRP, we set up all necessary data structures on the GPU, including arc instructions to aggregate the costs of the boundary arcs.

The work flow of the customization phase is as follows. We start by transferring the current metric decoder (less than a kilobyte) from main to GPU memory. Then we invoke two streams on the GPU, one computing the lowest level (using either Bellman-Ford or microinstrutions), and one setting the costs of the boundary arcs of the overlay graph. When both are done, one stream processes all remaining levels, while another asynchronously copies shortcut levels to main memory as soon as they are ready. This hides the time needed for the GPU-CPU data transfer almost completely.

Our implementation can use multiple GPUs in a single machine simply by allocating all top-level cells (and their subcells) among them so as to balance the (estimated) work. This approach requires no GPU to GPU communication during customization.

6 Experiments

We implemented all algorithms in C++ and CUDA, and compiled them with Visual C++ 2012 and CUDA 5.5. We ran most tests on a desktop computer running Windows 8.1. It has an Intel Core-i7 4770 (4 cores, 8 threads, 3.4 GHz, 4x64 KB L1, 4x256 KB L2, and 8 MB L3 cache) and 32 GiB of 1600-DDR3 RAM. Moreover, it has an ASUS NVIDIA GTX Titan with 6144 MiB of DDR5 RAM (6 GHz) and 14 multiprocessing units, each with 192 cores (2688 cores in total). The GPU has a normal clock rate of 837 MHz, but operates at 1 GHz as long as it stays cool enough (which was the case for all of our experiments).

Our focus is on the overall *customization time*, the total time from a metric change to the point we can compute driving directions (on the CPU). Thus, in our GPU setting, we include the time needed for data transfer (copying the metric decoder to the GPU and the shortcut costs back). All GPU times are averages over 1000 executions.

Table 1. Impact of mezzanine levels on customization done by local and global Bellman-Ford.

	LOCAL BELLMAN-FORD								GLOBAL BELLMAN-FORD							
	TIME ON LEVEL [MS]						TOTAL		TIME ON LEVEL [MS]						TOTAL	
Z	0	1	2	3	4	5	[ms]	[MiB]	0	1	2	3	4	5	[ms]	[MiB]
0	73.8	37.9	23.0	25.5	40.7	—	—	2212	157.6	82.1	56.9	45.8	43.7	40.4	477	3679
1	52.2	26.5	14.4	12.6	16.7	65.8	244	2816	94.9	48.9	28.6	25.9	25.6	22.8	295	4363
2	51.9	26.8	14.6	11.3	16.5	50.7	228	3412	99.3	47.9	26.1	23.0	23.7	21.3	289	5559
3	56.0	27.2	14.0	10.6	12.4	38.5	212	3911	114.9	47.7	25.5	20.4	20.0	18.8	297	5913
4	61.7	28.7	15.3	10.6	13.9	42.3	224	4342	133.1	52.0	44.7	21.3	21.2	19.7	344	7318

Our default input represents the road network of (Western) Europe and was made available by PTV AG for the 9th DIMACS Implementation Challenge [11]. This graph has $|V| = 18 \cdot 10^6$ vertices, $|A| = 42 \cdot 10^6$ arcs, and travel times as the cost function. As in previous work [7], we augment it by U-turn costs of 100 s (other turns are free). Our default CRP setup has 5 levels, with maximum cell sizes of $U_1 = 2^8$, $U_2 = 2^{11}$, $U_3 = 2^{14}$, $U_4 = 2^{17}$, and $U_5 = 2^{20}$; it requires about 72 MiB to store all shortcut costs.

Table 1 evaluates the global (GBF) and local (LBF) Bellman-Ford implementations, as well as how mezzanine levels affect them. As in previous work [7], we always keep two *phantom* levels (these are fixed mezzanine levels) of size $U_{-1} = 4$ and $U_0 = 32$. We always use LBF to compute the lowest level (cell size 4); this takes about 50 ms. We then vary the number of mezzanine levels (Z) between two consecutive levels; maximum mezzanine cell sizes are set so that their ratios across levels remain roughly constant. The table reports the times spent on each level (starting from the level below) for $0 \leq Z \leq 4$. A "—" entry means that LBF could not be executed because at least one cell has more than 65 535 inner arcs (see Section 3). We also report the total customization time (including all mezzanine levels) and the space consumption on the GPU.

We observe that mezzanine levels reduce customization times in general. One mezzanine level is enough on lower levels, but we can use up to three on higher levels, since more mezzanine levels can make more inner graphs fit into shared memory. Moreover, LBF is faster than GBF for all levels but the highest one, on which the number of cells is small and LBF is unbalanced. GBF consumes more space, mostly due to the distance array and thread data we need to store in global memory. For the rest of the paper, our default setting is to use $Z = 1$ up to level 1 and $Z = 3$ for higher levels; moreover, we use GBF for levels with fewer than 100 cells, and LBF otherwise. With this combination, customization takes 182.0 ms and uses 3034 MiB of GPU memory.

With this default setup, we now evaluate the effect of microinstructions. Figure 2 (left) reports the (relative) increase in customization time and GPU space when we use microinstructions up to a certain (possibly mezzanine) level, and Bellman-Ford afterwards. Using microinstructions up to cell size 32 reduces customization times by up to 20% (to 150.4 ms), but increases the overall space consumption by 25% (to 3792 MiB). Interestingly, using microinstructions for bigger or smaller cells does not help: many bigger cells cannot use instructions packed into 32 bits, and for smaller cells the overhead for initializing the memory array by arc instructions is too high. For the remaining experiments, we use microinstructions to process cells of size up to 32.

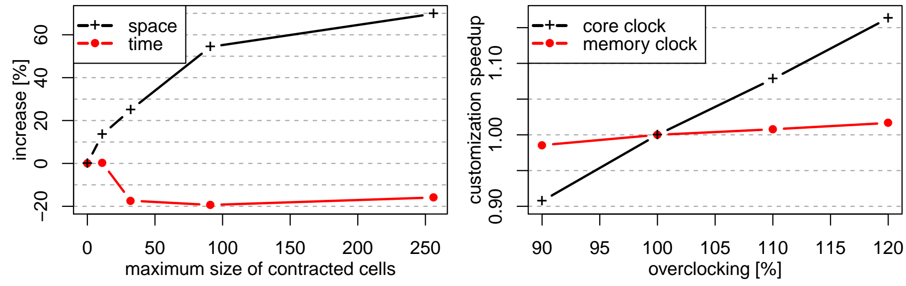

Fig. 2. Left: Impact of microinstructions on GPU space consumption and customization times for varying cell sizes. Right: Impact of clock rates on customization times.

Figure 2 (right) reports the speedup when we vary the core clock rate of the GTX Titan between 900 and 1200 MHz (recall that 1000 is the default) and the memory clock rate between 5400 and 7200 MHz (the default is 6000). We observed no data errors when overclocking in these ranges. The results indicate we are computation bound: increasing the core clock by 20% accelerates customization by almost 17%, whereas the memory clock rate has very little impact on the overall performance.

Table 2 compares our novel GPU implementation of CRP with the previous (highly tuned) CPU implementation [10], which uses microinstructions up to cell sizes of 256. We test various machines and GPU setups: M1-4 is our default machine (Core i7 4770), M2-12 has two 6-core Intel Xeon X5680 (3.33 GHz, 6x64 KB L1, 6x256 KB L2, and 12 MB shared L3 cache) with 96 GiB of DDR3-1333 RAM, and M2-16 has two 8-core Intel Xeon E5-2690 (2.9 GHz, 8x64 KB L1, 8x256 KB L2, and 20 MB shared L3 cache) with 384 GiB of DDR-1066 RAM. (We turn hyperthreading off for M2-12 and M2-16 because it does not help performance in our setting.) Finally, we test different GPU setups in M1-4: our default Titan, the Titan with core clock rate overclocked by 20%, two EVGA GTX 780 Ti OC (15 multiprocessors, 2880 CUDA cores, 1.2 GHz core, and 3 GiB of 7 GHz memory), as well as four GTX 780 Ti. (Note that a single GTX 780 Ti does not have enough memory for our default setup.) Besides customization times, we report the number of CPU threads used (t), the space occupied by all data structures in

Table 2. Key figures for various hardware setups.

			RATE [GHZ]		MEM [MIB]		TIME	POW	ENER.
machine	t	GPU	core	mem	main	GPU	[ms]	[W]	[J]
M1-4	1	Titan	1.0	6.0	484	3791	150.4	248	37.3
M1-4	1	Titan	1.2	6.0	484	3791	129.3	280	36.2
M1-4	2	2x780 Ti	1.2	7.0	484	3800	67.3	574	38.6
M1-4	4	4x780 Ti	1.2	7.0	484	3821	35.8	1045	37.4
M1-4	1	–	–	–	3119	–	2654	54	143.3
M1-4	8	–	–	–	3119	–	645	94	60.6
M2-12	12	–	–	–	3119	–	371	332	123.2
M2-16	16	–	–	–	3119	–	346	401	141.5

Table 3. Performance of CRP with GPU customization on other inputs.

				DATA STRUCT.			CUSTOM		QUERIES		
		$\|V\|$	cost	setup	space[MiB]		time	space	nmb.	dist	path
source	input	$[\times 10^6]$	func	[s]	main	GPU	[ms]	[MiB]	scans	[ms]	[ms]
PTV	Europe	18.0	distance	1736	484	3821	36.3	72.3	2993	1.30	5.95
	Europe	18.0	time	1736	484	3821	35.8	72.3	3050	1.17	3.17
TIGER	US	23.9	distance	2005	682	6939	57.9	113.1	3149	1.30	8.05
	US	23.9	time	2005	682	6939	56.6	113.1	3006	1.14	5.47
Bing	N. America	30.3	default	2767	908	8590	68.4	139.1	3387	1.12	3.98
	Europe	47.9	default	3618	1128	6707	62.8	124.1	3661	1.35	4.03

main and GPU memory, the (system-wide) power usage during customization, and the resulting average energy consumption for a single customization.

The GPU implementation always outperforms the CPU implementation. Using a single GPU, our algorithm is about 20 times faster than a sequential CPU execution. Increasing the number of GPUs linearly decreases customization times; with 4 GPUs, we are still 10 times faster than the best CPU setup (on 16 cores). Moreover, GPU customization is 2 to 3 times more energy-efficient, which is consistent with previous observations on related problems [6]. We also note that, since we store microinstructions on the GPU, the memory footprint on the CPU is reduced significantly.

Finally, we test more benchmark instances. Besides PTV Europe, we use TIGER USA from the 9th DIMACS Implementation Challenge, both with two cost functions: driving times (enriched by 100s U-turns) and distances. We also evaluate instances from Bing Maps, which build on Navteq data and include actual turn costs and restrictions; the proprietary “default” metric correlates well with driving times. We use the 4xGTX 780 Ti setup. Table 3 reports, besides customization times and CPU/GPU space consumption, the overall time spent (using all CPU cores) in the metric-independent phase (partitioning, microinstruction generation, and setting up the GPU data structures). For reference, it also reports the average performance for 10 000 random queries, given by the number of vertices scanned and the times to find the distance and a full description of the path (including the distance). Note that we do not cache path unpacking [10], thus capturing the average time to execute the *first* query after a metric change.

We can apply a metric change on every input in less than 70 ms, which is 10–12 times faster than on all 12 cores of M2-12 [10]. Preprocessing takes an hour or less, which is fast enough to incorporate topology changes in a timely manner. About 90% of that time is spent partitioning the graph; setting up our GPU data structures only takes a few minutes. On our Bing instances, computing (on the CPU) the first path after a metric changes takes about 4 ms, still considerably less time than customization.

7 Final Remarks

We have shown how to use GPUs to quickly incorporate a new cost function when computing shortest path on road networks. Although computing shortest paths on arbitrary graphs is not a natural fit for GPUs (given its irregular nature), we can still take advantage of their architecture by carefully exploiting various features of our application.

Since we work on a graph with fixed topology, we use preprocessing to carefully plan the computation and prepare GPU-friendly data structures. Instead of operating on the entire graph at once, we decompose it into small graphs (cells) with low diameter, which usually fit in shared memory and can be processed in parallel. Finally, cost functions are described compactly, saving on communication overhead.

References

1. I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Hierarchical hub labelings for shortest paths. In *ESA*, LNCS 7501, pp. 24–35. Springer, 2012.
2. H. Bast, D. Delling, A. V. Goldberg, M. Müller–Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck. Route planning in transportation networks. MSR-TR-2014-4, 2014.
3. H. Bast, S. Funke, P. Sanders, and D. Schultes. Fast routing in road networks with transit nodes. *Science*, 316(5824):566, 2007.
4. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2009.
5. A. A. Davidson, S. Baxter, M. Garland, and J. D. Owens. Work-efficient parallel GPU methods for single-source shortest paths. In *IPDPS*. IEEE, 2014.
6. D. Delling, A. V. Goldberg, A. Nowatzyk, and R. F. Werneck. PHAST: Hardware-accelerated shortest path trees. *Journal of Parallel and Distributed Computing*, 73(7):940–952, 2013.
7. D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Customizable route planning. In *SEA*, LNCS 6630, pp. 376–387. Springer, 2011.
8. D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Customizable route planning in road networks. Submitted for publication, 2013.
9. D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. Werneck. Graph partitioning with natural cuts. In *IPDPS)*, pp. 1135–1146. IEEE, 2011.
10. D. Delling and R. F. Werneck. Faster customization of road networks. In *SEA*, LNCS 7933, pp. 30–42. Springer, 2013.
11. C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, DIMACS Book 74. AMS, 2009.
12. R. Geisberger, P. Sanders, D. Schultes, and C. Vetter. Exact routing in large road networks using contraction hierarchies. *Transportation Science*, 46(3):388–404, August 2012.
13. R. Geisberger and C. Vetter. Efficient routing in road networks with turn costs. In *SEA*, LNCS 6630, pp. 100–111. Springer, 2011.
14. M. Hilger, E. Köhler, R. H. Möhring, and H. Schilling. Fast point-to-point shortest path computations with arc-flags. In Demetrescu et al. [11], pp. 41–72.
15. M. Holzer, F. Schulz, and D. Wagner. Engineering multilevel overlay graphs for shortest-path queries. *ACM Journal of Experimental Algorithmics*, 13(2.5):1–26, December 2008.
16. K. Madduri, D. A. Bader, J. W. Berry, and J. R. Crobak. Parallel shortest path algorithms for solving large-scale instances. In Demetrescu et al. [11], pp. 249–290.
17. P. Martin, R. Torres, and A. Gavilanes. CUDA solutions for the SSSP problem. In *ICCS*, LNCS 5544, pp. 904–913. Springer, 2009.
18. U. Meyer and P. Sanders. Δ-stepping: A parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, 2003.
19. H. Ortega-Arranz, Y. Torres, D. Llanos, and A. Gonzalez-Escribano. A new GPU-based approach to the shortest path problem. In *HPCS*, pp. 505–511, 2013.
20. C. Sommer. Shortest-path queries in static networks. *ACM Comp. Surveys*, 46(4), 2014.