
Article:
Heywood, P., Maddock, S., Bradley, R. et al. (2019) A data-parallel many-source shortest-path algorithm to accelerate macroscopic transport network assignment. Transportation Research Part C: Emerging Technologies, 104. pp. 332-347. ISSN: 0968-090X

https://doi.org/10.1016/j.trc.2019.05.020

Transportation Research Part C 104 (2019) 332–347

Contents lists available at ScienceDirect

Transportation Research Part C

journal homepage: www.elsevier.com/locate/trc

A data-parallel many-source shortest-path algorithm to accelerate macroscopic transport network assignment

Peter Heywood[a,*], Steve Maddock[a], Richard Bradley[b], David Swain[b], Ian Wright[b], Mark Mawson[c], Graham Fletcher[d], Roland Guichard[d], Roger Himlin[e], Paul Richmond[a]

[a] The University of Sheffield, United Kingdom
[b] Atkins Ltd., United Kingdom
[c] Science and Technology Facilities Council, United Kingdom
[d] Transport Systems Catapult, United Kingdom
[e] Highways England, United Kingdom

ARTICLE INFO

Keywords:
Macroscopic road network simulation
Assignment
Data-parallel
Shortest path
Algorithms

ABSTRACT

The performance and scalability of macroscopic assignment and simulation software limits the quantity, scale and complexity of simulations of transport networks that are used to aid the planning, design and operation of transport systems. Through the application of many-core processing architectures (such as Graphics Processing Units (GPUs)) and data-parallel algorithms, the performance of such software can be dramatically increased. In this paper, a work-efficient, Multiple Source Shortest Path (MSSP) implementation of the Bellman-Ford algorithm is proposed to dramatically increase the performance of shortest path calculations for low-density high-diameter graphs, which are characteristic of transportation networks. This is achieved through the use of an Origin-Vertex Frontier to increase the level of parallelism within the shortest path algorithm for multiple source vertices in low-density graphs, when executed on GPUs. The algorithm is implemented within the SATURN transport simulation software and benchmarked on a range of real-world transportation networks. Our algorithm improves hardware utilisation of massively-parallel GPUs; demonstrating performance improvements of up to 7.8x over a multi-core CPU (Central Processing Unit) implementation.

1. Introduction

Computer simulations of transport networks play a key role in the planning, maintenance and operation of transport systems. The insight provided by simulation software informs the decision making process regarding transport network improvements; including improvements which facilitate the ever-increasing demand on our transport networks (such as dynamic infrastructure optimisation (Hao et al., 2018; Frejo et al., 2019)). Simulations are used to predict journey length and delay given specific demand on a transport network. Microscopic, Mesoscopic or Macroscopic approaches can be used for such simulations. Microscopic approaches perform modelling of individual vehicles, which is computationally expensive prohibiting their use for large scale (national) modelling. Macroscopic simulations of transport networks use a top-down approach to accurately model the flow of transport through a network using abstract representations, with reduced data and computational requirements. However, even macroscopic simulations can take several days of processing using current state-of-the-art simulation software to produce results for a single large simulation. This issue

* Corresponding author.
E-mail addresses: p.heywood@sheffield.ac.uk (P. Heywood), p.richmond@sheffield.ac.uk (P. Richmond).

https://doi.org/10.1016/j.trc.2019.05.020
Received 9 March 2018; Received in revised form 28 March 2019; Accepted 16 May 2019
Available online 25 May 2019

is compounded as multiple simulations may be required to cover a range of scenarios and candidate initiatives being considered. Computational constraints can reduce the effectiveness of simulations, as modelling practitioners are restricted in the quantity, variety and, therefore, scale of simulations which can be completed within a reasonable time-frame (Antoniou et al., 2014).

To improve simulation performance and enable a broader range of simulations to be executed, greater levels of parallelism (common in all modern processing architectures) must be applied to transport network simulation approaches. A potential solution is modern highly-parallel many-core processing architectures, such as Graphics Processing Units (GPUs). GPUs offer greater levels of computational performance than multi-core architectures, with relatively low cost and energy usage. However, to utilise the high levels of parallelism and computational performance available from modern many-core architecture, the algorithms and data structures used within macroscopic simulation software must demonstrate excellent scaling to utilise highly parallel many-core design. Previous multi-core parallelisation approaches must therefore be re-considered in favour of alternative approaches which better exploit modern architectures.

This paper describes how the performance of shortest path calculations for low-density high-diameter graphs, typical of transport networks, can be significantly accelerated for large scale models, through the adoption of data-parallel algorithms and data structures using highly-parallel many-core GPUs. The specific contributions are: (i) The use of an *Origin-Vertex Frontier (OVF)* in conjunction with the Bellman-Ford algorithm to maximise parallelism and solve the shortest path problem for multiple sources concurrently using many-core processors. This includes the use of *Cooperative Thread Arrays (CTA)* to balance the workload within the many-core architecture's SIMD (Single-Instruction Multiple-Data) execution units, caused by the varying degree of vertices in the OVF; and (ii) a demonstration of significant performance improvements for transport network simulations using GPUs through the OVF-based shortest path implementation to accelerate the dominant assignment phase of the macroscopic assignment and simulation process.

To demonstrate the capability of modern GPU architectures on macroscopic transport simulation, we have modified the SATURN (Simulation and Assignment of Traffic to Urban Road Networks) software suite, with the objective of reducing the run-time for large scale macroscopic simulations. SATURN is a macroscopic assignment and simulation package used for the analysis and evaluation of traffic management schemes (Van Vliet, 1982), used for a broad range of tasks within the transport modelling sector including large-scale regional models of the UK (Cox, 2016). Originally developed as a combined simulation and assignment model, most suited for analysing relatively minor network changes, it has been extended to function as a "conventional" traffic assignment simulator or as a junction simulator (Van Vliet, 2015). Assignment-simulation models use an iterative process to combine both the assignment of traffic demand to the transport network and the simulation of the assigned demand to update network costs until a state of equilibrium is achieved (Hall et al., 1980). Various statistical measures which are used to evaluate the simulated transport network are produced. The proposed techniques used to accelerate the assignment process in SATURN could be applied to any macroscopic transport network assignment software which relies on shortest path computation as a part of the assignment process. Our results show assignment speed-ups of up to 53x and 7.8x compared to the existing serial and multi-core CPU implementations on real-world transport networks. The GPU accelerated implementation demonstrates greater levels of performance, with improved performance-scaling.

Section 2 provides a description of the assignment-simulation process, a summary of relevant shortest path algorithms and an introduction to software parallelisation. Section 3 describes macroscopic assignment and simulation, the shortest path problem and the use parallel processing. Section 4 provides the details of the algorithms used to accelerate assignment using GPUs, and Section 5 contains the results of a set of performance benchmarks on real-world transportation networks. Section 6 discusses the benchmark results and future work, with Section 7 providing the final conclusion.

2. Related work

This section covers existing work and terminology related to GPU accelerated macroscopic assignment and simulation. Section 2.1 describes the macroscopic assignment and simulation process. Section 2.2 provides terminology relevant to network graphs in a transport simulation context. Section 2.3 describes the shortest path problem and the algorithms which may be used to solve it. Section 2.4 provides an overview of parallel processing methodologies; how they can be applied to the transport simulation and assignment domain; and the impact of parallel processing on algorithm selection.

2.1. Assignment and simulation

Macro-scale simulations are used predict the impact of changes to transport networks informing stakeholders of the potential benefits of implementing such changes. This can include the effects of physical changes to the transport network, optimisation of existing control infrastructure (Aboudina et al., 2018), or the addition of new dynamic infrastructure (Frejo et al., 2019). Combined assignment-simulation applications use an iterative process to assign traffic demand to a network and simulate the assigned demand until a level of equilibrium has been achieved. Applications which use the methodology can typically be decomposed into several phases:

(i) Input; (ii) Pre-Processing; (iii) Assignment-Simulation Loop; (iv) Post-Processing; (v) Output.

The input phase of the application is the loading of the required data such as network, demand data and simulation parameters from disk. A pre-processing phase is used to ensure data input into the application is suitable for use with the application, or to improve the performance of the application. This may include modifications to the data structure used to represent the road network, to improve software performance or check for errors in the input data. The computationally demanding portion of the application is typically the assignment-simulation loop. This loop is an iterative process, assigning demand to the transport network and simulating the assigned routes until equilibrium is achieved, following Wardrop's principal of traffic equilibrium (Wardrop, 1952). The assignment phase uses the current network costs from the previous simulation phase (or initial network data) to compute the routes

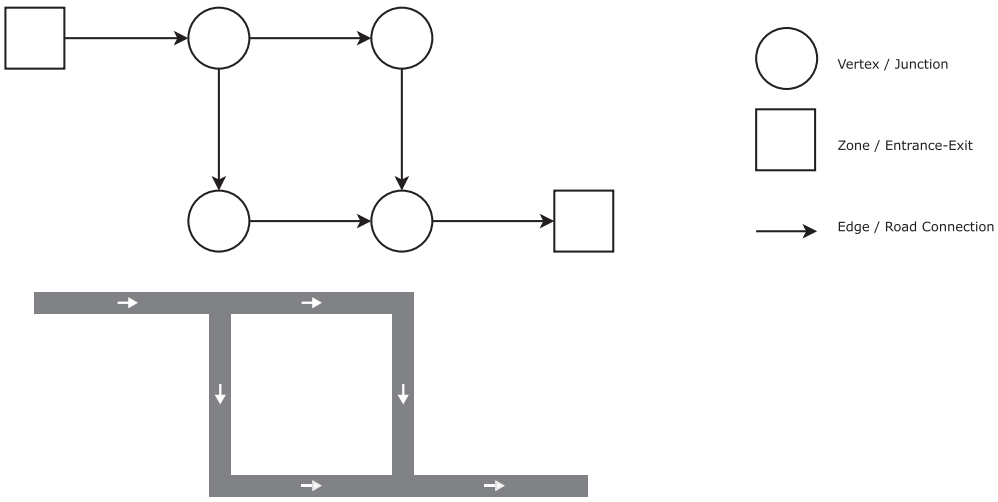


Fig. 1. The correlation between edges, vertices and zones for a small section of a one-way road network.

through the transportation network used by traffic. The assigned routes are then used as input to a simulator, which models the transport system as flows of vehicles moving along the sections of roads and turns within the transport network. The simulation phase produces a modified version of the transport network, with updated edge-costs, which, as part of the iterative process, is then used as input to the assignment phase. The assignment process has a high computational cost (see Section 3) and relies on the calculation of shortest paths within the road network between each origin-destination pair. Lastly the output of the assignment simulation loop is post-processed, reversing any modifications made by the pre-processing phase, and the final outputs such as network flow and delay are produced.

2.2. Road network graph terminology

Before moving on to the details of the shortest path problem, we define relevant terminology. A directed weighted graph (G) is comprised of a set of vertices (V), directed edges (E) and edge cost values (C). Within transport modelling, the set of vertices V is made of two categories of vertex: (i) a set of real world vertices representing the intersections in the road network, and (ii) a set of Zones – virtual vertices which represent an area to allow aggregate origin-demand information. Fig. 1 shows the relationship between edges, vertices and zones for an example section of a road network. The characteristics of a graph such as *density*, *diameter* and *vertex degree* must be considered during algorithm selection, as different algorithms may perform differently based on these characteristics. The density of a graph is given by $D = \frac{|E|}{|V|(|V|-1)}$, whilst the diameter of the graph is the greatest distance between any pair of vertices and the *degree* of a vertex is the number of edges connected to the vertex. Typically graphs used to represent transport networks are very *sparse* (*low-density*) with *large diameters*. This differentiates them significantly from other graphs such as social media interaction graphs, which tend to be very dense graphs with low diameters.

2.3. Shortest path problem

Within macroscopic transport network simulation the vehicle demand can be specified as a flow of vehicles between pairs of zones using an *Origin-Destination* (OD) matrix, which may be from observed real world data, or more-likely an estimated OD matrix Ma and Qian (2018). The assignment process is the application of this demand information onto the transport network in a realistic fashion. To achieve this goal, the path through the transport network between each pair of origin and destination zones must be found. Shortest path algorithms can be classified as either (i) All Pair Shortest Path (APSP) algorithms, or as (ii) Single Source Shortest Path (SSSP) algorithms.

APSP algorithms find the minimal cost paths between each pair of vertices in the graph. Several algorithms fall into this category, including the Flloyd-Warshall algorithm (Floyd, 1962) and Johnson's algorithm (Johnson, 1977). The Flloyd-Warshall algorithm incrementally improves an estimate for the shortest path between any two vertices until the optimal estimates are found, with worst-case performance complexity of $O(|V|^3)$. In order to reconstruct the paths through the network between any pair of vertices, two arrays of $|V|$ elements are required. APSP algorithms store the cumulative cost and edge for each path between pairs of vertices within the network, which for large networks requires a large amount of memory to store the results. Unless every shortest path between all pairs of vertices are specifically required, the much greater storage requirements typically limits the use of APSP algorithms.

SSSP algorithms find the path with the minimal cost from a single origin vertex to all destination vertices in the graph (i.e. the minimum cost paths from a single zone to all non-zone vertices and other zones in the network). These algorithms produce both the minimal cost for each route through the network, as well as the route taken to produce the minimal cost. If there are multiple shortest

routes with equivalent cost, only a single path is stored. Dijkstra's algorithm (Dijkstra, 1959), the D'Esopo-Pape algorithm (Pape, 1974) and the Bellman-Ford algorithm (Bellman, 1958; Ford, 1956) are some of the well-established SSSP algorithms.

Currently, Dijkstra's algorithm (Dijkstra, 1959) is asymptotically the fastest serial SSSP algorithm for non-negative directed graphs when implemented using a Fibonacci heap (Fredman and Tarjan, 1987). The high level of serial performance is achieved through the use of a priority queue to process the vertices and edges of the graph in an efficient order, reducing the amount of work required to find a correct result. The D'Esopo-Pape algorithm (Pape, 1974) is another efficient serial algorithm, which attempts to minimise the work required to find a correct result through the use of a *loose-end-table*. The loose-end-table is a double-ended queue, onto which newly encountered vertices are appended during graph traversal from a specified source, whilst previously encountered vertices are prepended. Serial implementations of this algorithm are deterministic, as vertices are processed in the same order, even when there are multiple shortest paths with equivalent cost. Work-efficient serial SSSP algorithms such as Dijkstra's algorithm (Dijkstra, 1959), Dial's algorithm (Dial, 1969) and the D'Esopo-Pape algorithm (Pape, 1974) rely on the order in which edges of the network are processed to minimise work-load and offer high levels of performance. In contrast, the Bellman-Ford algorithm (Bellman, 1958; Ford, 1956) is not a work-efficient algorithm. It uses relaxation, where an approximation of the path cost is incrementally replaced by more accurate approximations until the actual minimal cost is found. All edges in the graph are relaxed iteratively, until the longest possible path (of $|V| - 1$ edges) must have been considered. As each edge is considered many times, the shortest path is guaranteed to have been found. The Bellman-Ford algorithm performs a significant amount of additional processing which may have no impact on the final shortest path results. Several improvements have been made to this algorithm to improve the average performance, however work-efficient algorithms still out-perform these optimisations for serial implementations. The key advantage of the Bellman-Ford algorithm is that it does not rely on order of execution and is therefore an ideal candidate for parallelisation.

2.4. Parallel processing

Modern computing hardware uses parallelism to increase the computational performance available to software, as physical limitations have restricted the limits of frequency scaling as the dominant method of increasing processing performance. To understand the potential impact of GPUs on transport simulation, we must understand: the possible approaches used in parallel processing; the existing impact of parallel processing on transport simulation; and how the parallel processing architecture effects the algorithms used to solve a problem.

2.4.1. Parallel processing approaches

Parallel processing architectures can broadly be categorised as either *Shared-Memory Parallel systems* or *Distributed systems*. Shared-memory parallel architectures can be classified as either *multi-core* architectures or *many-core* architectures. Distributed systems increases total performance by distributing work across a network of connected computers. Individual nodes of the computer network may contain shared-memory processing architectures, and leverage shared-memory parallelism within the processing node.

Multi-core architectures use small numbers of complex processing cores to increase performance, by allowing multiple threads to concurrently work on different tasks. This makes multi-core architectures suitable to low-latency tasks. Many-core architectures use much larger numbers of less complex processing cores to offer much greater levels of data parallelism, but with reduced single-core performance making them more suitable for high throughput tasks. GPUs are an example of a highly-parallel many-core processor architecture. Originally developed for 2D and 3D computer graphics, GPUs have recently become dominant in the high-performance computing sector, with a significant presence in the *Top 500* (Top 500, 2016) and *Green 500* (Green 500, 2016) lists of the most computationally powerful and energy efficient supercomputers in the world.

In addition to the underlying parallel processing architecture, parallel processing approaches can also be classified as either *Task Parallel* or *Data Parallel*. Task parallelism involves distributing independent processing tasks to separate processing threads or processors. Task parallelism is well-suited to distributed systems and multi-core architectures. Data parallelism involves applying the same algorithms and processing operations to different units of data at the same time. Data-parallel algorithms can further be categorised as *coarse-grained* or *fine-grained*, depending on the scale of the unit of data. More formally, parallel systems can also be categorised into four categories, as defined in Flynn's taxonomy (Flynn, 1972). Sequential processing architectures can be categorised as Single-Instruction stream, Single-Data stream (SISD) architectures, where only a single instruction can be processed on one data element at any one time. Single-Instruction stream, Multiple-Data stream (SIMD) architectures apply the same instruction to multiple elements of data concurrently, typically using vector units in modern CPUs. Fine-grained data-parallelism is well-suited to SIMD architectures. Multiple-Instruction stream, Single-Data stream (MISD) involves multiple instructions operating on a single stream of data and Multiple-Instruction stream, Multiple-Data stream (MIMD) architectures include modern multi-core processors and distributed systems, where multiple instructions can be executed concurrently on multiple streams of data, typically used for task-parallelism. GPUs are frequently described as Single-Instruction stream, Multiple Thread (SIMT) architectures (NVIDIA, 2009). SIMT architectures operate a SIMD model within multiple execution groups of threads, more commonly referred to as *warps*.

2.4.2. Application of parallel processing to transport simulation

Parallel processing can be applied to many areas of the transport simulation process in order to reduce the time required for the tasks to be completed. Existing commercial software used for transport simulation may leverage parallel processing to increase application performance. Macroscopic simulation tools such as SATURN (Bradley et al., 2016) and VISUM (PTV AG, 2017) use multi-core CPUs and multi-threading to reduce application run-times. Mesoscopic and Microscopic simulation tools such as Aimsun (Barceló and Casas, 2005), Vissim (Fellendorf, 1994) and MATsim (Balmer et al., 2009) use task-parallelism and coarse-grained data-

parallelism to increase simulation performance through multi-core processing architectures. Distributed computing is not prevalent within the transport modelling industry.

Task parallelism has been successfully applied to many aspects of transport simulation. Online simulations of transport systems, which make use of real-time data to predict the future state of the system, can benefit from distributed task-parallelism. Hunter et al. show that a collection of online transport applications can co-operate using an ad hoc approach to improve performance (Hunter et al., 2009). Aboudina et al. (2018) leveraged task parallelism in a distributed cloud setting for large-scale traffic control optimisation using mesoscopic simulations and genetic algorithms (Aboudina et al., 2018); while Li et al. (2019) propose the use of distributed task-parallelism to mitigate performance loss from higher-fidelity macroscopic simulations (Li et al., 2019).

Using multi-core processors and task-parallelism can improve the performance of transport simulation tasks such as mesoscopic discrete event simulation (Qu and Zhou, 2017), shortest path algorithms (Lawson et al., 2013) and transport management software (Johnson et al., 2016).

Different methods must be used to distribute work across processors dependent upon the tasks to be completed. Lawson et al. demonstrate a task-parallel implementation of a Label Correcting Algorithm (LCA) to solve the all-pair-shortest-path problem, where independent tasks are distributed proportionally across available processors. The individual processor cores in a multi-core CPU are assigned equal shares of the source vertices (Lawson et al., 2013). Alternatively, large tasks can be partitioned into multiple smaller tasks which are distributed across available processors before results must be collated. This may involve boundaries where sub-tasks may need to communicate. For instance, transport network management software may partition large-scale transport networks into a set of smaller, connected networks through a domain decomposition process (Johnson et al., 2016). Each processor is then responsible for a smaller task which may be simpler to solve, however the addition of a boundary region where multiple processors must interact may limit the performance improvements offered using this type of approach. It is therefore important to use an appropriate domain decomposition algorithm for partitioning the domain. Task parallelism in a shared-memory environment has also been demonstrated to offer performance improvements to event-based mesoscopic simulations. Qu and Zhou (2017) showed an order of magnitude improvement in performance of their simulator using a modern multi-core processor (Qu and Zhou, 2017).

The POLARIS (Auld et al., 2016) framework applies multi-core CPU processing to integrated urban simulation. Agent based modelling is applied to an activity-based model, including activity-based demand modelling, route choice and microscopic transport simulation. Multiple processor threads are used to improve application performance.

In a similar vein, GPUs have shown great potential in improving the performance of agent-based microscopic road network simulation. Heywood et al. demonstrated performance improvements of over 40 times for large scale simulations containing hundreds of thousands of individual vehicles when compared to the commercial microsimulation tool Aimsun (Heywood et al., 2017). This was achieved through the use of fine-grained data-parallel algorithms which are more appropriate for the many-core processing architecture than task-parallel algorithms.

2.4.3. Parallel algorithms

The algorithms and data structures used within software applications must be considered carefully based upon the hardware architecture in use, as multi-core and many-core architectures have different requirements. This is demonstrated by the different algorithms used by graph processing libraries depending upon the target architecture. Single-core (serial) and multi-core CPU (Central Processing Unit) or distributed CPU libraries such as the Boost Graph Library (BGL) (Siek et al., 2001) and the Parallel Boost Graph Library (PBGL) (Gregor and Lumsdaine, 2005) use highly work-efficient algorithms such as Dijkstra's algorithm to offer a high performance general purpose SSSP implementation. In contrast, GPU accelerated libraries which provide generalised graph processing using GPUs, such as Gunrock (Wang, 2014), CuSha (Khorasani et al., 2014) & NVGraph (NVIDIA, 2016), use algorithms which expose greater levels of parallelism, such as the Bellman-Ford algorithm, to leverage the performance available from the many-core GPU architecture. These GPU accelerated libraries show considerable performance improvements over CPU-based libraries for the shortest path problem for large, dense, low-diameter graphs such as social network interaction graphs. For such graphs, the Gunrock SSSP implementation uses a work-efficient version of the Bellman-Ford algorithm which offers up to 60x speed-up compared to the serial CPU-based implementation of Dijkstra's algorithm from the Boost Graph Library (Davidson et al., 2014). The implementation of the Bellman-Ford algorithm in Gunrock uses a *vertex-frontier* to track which vertices must be considered at each stage of the parallel algorithm, improving efficiency. The vertex-frontier concept has also been successfully applied to other Graph processing algorithms (Busato and Bombieri, 2015). There has been some success in parallelising highly-efficient algorithms such as Dijkstra's algorithm for GPUs. Algorithms based on Crauser et al.'s work divide Dijkstra's algorithm into several steps which each expose a degree of parallelism (Crauser et al., 1998). GPU-based implementations can launch parallel kernels for each of these steps. This has been shown to provide considerable speed-up compared to CPU-based implementations for large scale graphs containing millions of vertices and with relatively high density compared to typical transport networks (Ortega-Arranz et al., 2013).

However, unlike generalised GPU-accelerated libraries, our work targets low-density, high-diameter graphs, which are characteristic of transport networks. Section 4 describes the alternative modifications and optimisations we made to the Bellman-Ford algorithm to increase the performance of shortest path calculations on low-density networks by increasing the level of parallelism exposed by the algorithm.

3. SATURN

SATURN is an application suite used for the analysis and evaluation of traffic management schemes (Van Vliet, 1982). Originally developed as a combined simulation-assignment model and released in 1982 by the Institute for Transport Studies at the University of Leeds, it has since been extended to support both "pure junction simulation" and execution as a "conventional traffic assignment

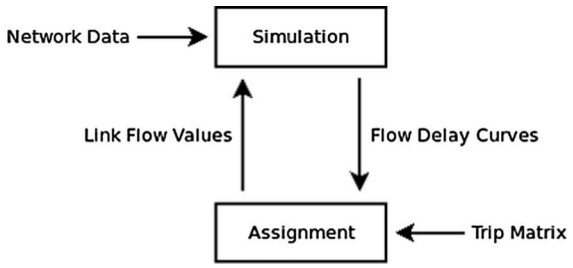


Fig. 2. The assignment-simulation loop within SATURN (Van Vliet, 1982).

model, with or without simulation" (Van Vliet, 2015). The SATURN suite is comprised of many executable files which each cater for different purposes. *SATALL* is the primary application when used as a combined simulation-assignment model, which can be decomposed into several key steps. Details of the broad range of data produced during a SATALL simulation is described in the SATURN user manual (Van Vliet, 2015).

SATURN currently uses multi-core CPU parallelism, characteristic of macroscopic simulations software, to provide increased performance. It is currently available as both a sequential, single-threaded CPU-based application; and also as a multi-threaded CPU-based application. The multi-threaded version of SATALL increases the performance of the assignment process by handling each SSSP calculation as an independent thread (for each origin zone) using OpenMP (Dagum and Menon, 1998). The sequential implementation of SATURN is compiled using the Silverfrost Fortran compiler. The multi-core version of SATURN, which uses OpenMP, is compiled using a mixture of the Silverfrost compiler and the Intel Fortran compiler.

To demonstrate the applicability of many-core architectures and the change in methodology required to adopt data-parallelism within macroscopic transport modelling, we have accelerated SATURN using GPUs. SATURN follows the overall application pattern characteristic of macroscopic road network simulators, as described in Section 2. During the pre-processing phase, SATURN can optionally produce a graph representation of the road network – known as a *SPIDER* network – which is of a higher-density then an unmodified representation. This process is used to increase the performance of the application, as the resulting graph is more well-suited to processing by the D'Esopo-Pape algorithm used within the existing serial and multi-core implementations. The vertices and edges in the network are modified to produce an equivalent graph with increased density and reduced diameter, by replacing chains of individual edges with a single SPIDER edge, reducing the number of edges in the network. If all of the edges connected to a vertex are replaced, the vertex can also be removed. This generally results in a reduction of the time required for the key graph processing algorithms required for assignment and simulation, whilst allowing any results to be reconstructed for the original network.

The Assignment-Simulation loop used in SATALL (Fig. 2) is based on Wardrop's principal of traffic equilibrium (Wardrop, 1952); traffic will settle down into an equilibrium where no driver can reduce their journey time by choosing an alternate route.

To understand the performance of SATURN, the application was profiled for a large-scale real-world network (5194 zones) using typical parameters. Table 1 contains the time taken for the longest-running routines within SATALL, for a total application runtime of greater than 12 h. A single routine (*A*) accounts for over 97% of the total application run time, and is the clear limit on performance.

Routine *A* is part of the assignment phase of SATALL. It calculates the flow per-edge in the network for all trips in the OD matrix, based on the edge costs from the previous simulation phase. This process is repeated for each user class (UC) of vehicles within the simulation, and is repeated at each iteration of the assignment-simulation loop. It can be decomposed into two parts: (i) *Shortest Path Calculation* and (ii) *Flow Accumulation*. The shortest path through the network for each pair of zones in the OD matrix is found using the D'Esopo-Pape algorithm. The flow across each edge in the network is then calculated by tracing the route through the network for each trip in the OD matrix. Additional profiling of this routine shows that the shortest path calculation accounts for over 95% of time within this function during serial execution. Algorithm 1 shows the process for assigning demand to the graph representing the transportation network in the serial version of SATALL.

Algorithm 1. Existing Serial SATALL algorithm

Input: Graph G with Z zones,
UC User classes,
Origin-Destination Matrix OD of dimensions Z by Z
Output: Flow per edge in G, F
1: **gpu parallel for** User-Class **do**
2: **for** origin zone $o \in \{1, ...Z\}$**do**
3: $paths \leftarrow$ call *CalculatePaths*(o, G, OD)
4: **for** destination zone $d \in \{1, ...Z\}$**do**
5: $flow(o, d) \leftarrow$ call *AccumulateFlow*$(d, o, paths, G, OD)$
6: **end for**
7: **end for**
8: call *AggregateAndPostProcessFlowData*$(flow)$
9: **end for**

Table 1
Serial Fortran per-routine run-time performance for large real-world benchmark model (5194 zones) sorted by time. The six longest-running routines are shown.

Subroutine	CPU Time (seconds)	CPU Time (Percent)
A	39337.911	97.392%
B	135.981	0.337%
C	107.589	0.266%
D	58.087	0.144%
E	57.774	0.143%
F	55.587	0.138%

The existing serial and multi-core implementations both use the D'Esopo-Pape algorithm. This algorithm was selected for use within SATURN as it out-performed other SSSP algorithms for the networks being simulated at the time of development (Van Vliet, 1978). Replacing the existing D'Esopo-Pape algorithm with a more recent work-efficient SSSP algorithm may lead to a performance improvement. However the scale of improvement would not be significant (Głkabowski et al., 2013) and would be dependent upon the network in question. For some networks the D'Esopo-Pape algorithm could potentially still outperform more recent algorithms (Zhan and Noon, 1998).

4. GPU accelerated assignment

To increase the performance of SATURN using GPUs, the performance-critical routine which relies on the calculation and use of shortest path results was parallelised for many-core GPUs using the CUDA framework. CUDA is the parallel computing API and runtime produced by NVIDIA enabling general purpose programming on NVIDIA Graphics Processing Units (Nvidia, 2015). CUDA provides access to the highly-parallel GPU architecture, allowing high levels of performance for data-parallel algorithms. Functions (or kernels) are concurrently executed across many threads and cores on the GPU, which operate on varying data. The number of threads is defined by the user, using a hierarchical layout of *threads*, *blocks* and *grids*. Individual threads are transparently scheduled to a single CUDA core within *warps* of 32 threads. Threads within a warp are executed together in *instruction lock-step*, due to the SIMD processing architecture, with each of the 32 threads executing all instructions together regardless of any divergence. Zero cost hardware warp scheduling is very effective at hiding the cost of data movement. By oversubscribing the number of thread blocks, the GPU is able to switch out warps waiting on data dependencies for warps which are ready to execute instructions.

The CUDA API provides the user with fine-grained control over the use of memory and caches, which are required to maximise the performance of parallel code. However, Data must be transferred to and from the GPU, typically using the PCI-E bus, which is a relatively slow process. This cost can be mitigated by asynchronously transferring data whilst performing high intensity arithmetic.

When working with multi-threaded parallel processing systems, care must be taken when multiple threads concurrently modify shared data structures. If multiple threads attempt to modify the same memory locations, errors can occur. To avoid this problem, a special set of operations exist, known as *atomic operations*. These operations are guaranteed to occur in isolation of other attempts to read or modify the relevant memory, but are typically serialised, resulting in a reduction of performance compared to equivalent non-atomic operations. These characteristics of the many-core graphics processing architecture and the CUDA runtime must be carefully considered when selecting and implementing data-parallel algorithms, to ensure high levels of performance are achieved.

The characteristics of the GPU programming model and hardware execution have the following implications for our implementation of the Bellman-Ford algorithm:

1. The hardware must be sufficiently over-subscribed to hide memory management, i.e. high levels of parallelism are necessary.
2. Threads within a warp should minimise divergence to maintain parallel efficiency.
3. Movement of data to and from the GPU should be minimised, especially when asynchronous transfer is not possible due to a data-dependency.

The modifications made to SATURN to produce a GPU accelerated assignment process were developed using an iterative process of Assess, Parallelise, Optimise, Deploy – the APOD cycle (NVIDIA, 2012). Initially the performance limiting factor was the shortest path computation, which was parallelised using the Bellman-Ford algorithm. The level of parallelism exposed by the algorithm for low-density graphs was increased primarily through the use of an Origin-Vertex Frontier (OVF), and load balancing techniques. As the performance of the shortest path computation was improved, the process of *flow accumulation* became the performance critical factor which was also parallelised.

Section 4.1 describes the implementation of the shortest path calculations for multiple concurrent origins using the OVF, and Section 4.2 describes the parallelised techniques used for the flow accumulation process which became critical to performance.

4.1. Many source shortest path

The assignment process requires the calculation of shortest paths from all origin zones to all destination zones in the Origin-Destination trip matrix. Zones only account for a small portion of the total number of vertices within the graph used to represent the

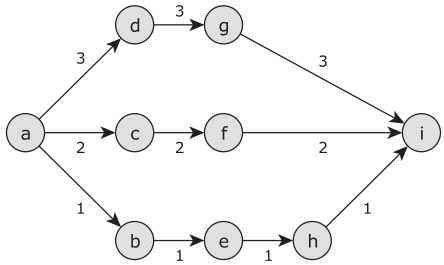


Fig. 3. A simple example graph to illustrate the use of a vertex-frontier.

road network. All pair shortest path algorithms would compute a significant amount of unnecessary paths between non-zone vertices, and so APSP algorithms were dismissed.

The implemented algorithm used to calculate shortest paths is based on the Bellman-Ford SSSP algorithm, with a series of modifications and optimisations to improve run-time performance for low-density high-diameter graphs characteristic of road networks. The main difference between our solution and the traditional Bellman-Ford algorithm is that the implemented algorithm solves the shortest path problem for multiple origin vertices concurrently, rather than a single origin as with SSSP algorithms, or all origins as with APSP algorithms. This increases the level of parallelism exposed and enables data re-use, improving performance for many-core processing architectures. Additionally the work-load caused by varying vertex degree is balanced using a Cooperative Thread Array (CTA) approach.

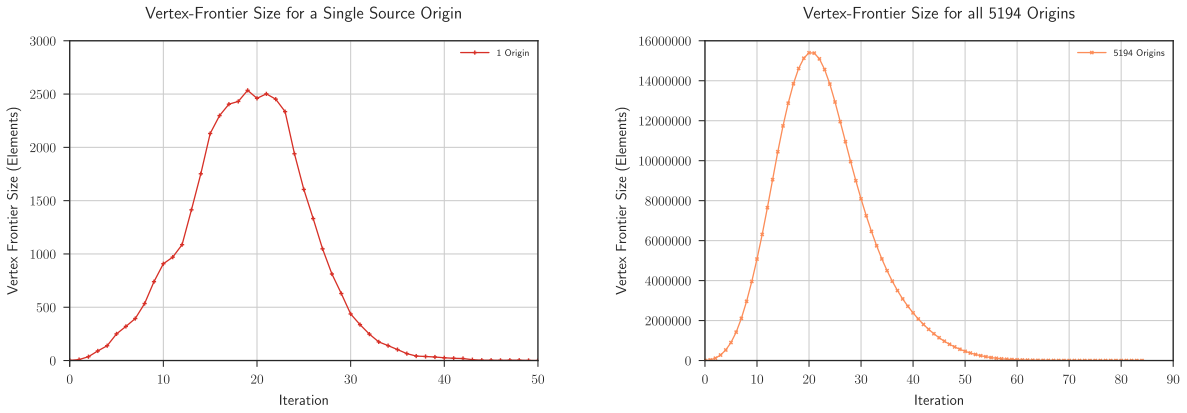
For most graphs, the worst-case number of Bellman-Ford iterations is not required to find the shortest paths, and instead the algorithm can be terminated when no changes are made in a single iteration (Bannister and Eppstein, 2012). The amount of work can further be reduced through the use of a *vertex-frontier* (Wang et al., 2016). By maintaining a list of vertices which were updated in the previous iteration (initialised to contain the source vertex), only edges which leave vertices in the vertex frontier need to be considered. Fig. 3 and Table 2 illustrate the use of a vertex-frontier within the Bellman-Ford algorithm for a simple graph for origin vertex `a`. The vertex frontier is initialised to contain the source vertex, `a`. The edges leaving the vertex are relaxed, resulting in an updates for vertices `b`, `c` and `d` which are inserted into the vertex-frontier for the next iteration. Next the edges which leave the vertices in the vertex-frontier are relaxed, resulting in a new frontier of vertices `e`, `f` and `g` This process is repeated until the vertex-frontier is empty, at which point the algorithm terminates.

However, transport networks are typically represented by sparse, low-density graphs with high-diameter. This means that the vertex-frontier approach may not offer sufficient levels of parallelism to fully utilise the GPU and achieve a performance improvement compared to existing techniques. To address this issue, the vertex-frontier concept can be extended to increase the level of parallelism exposed in cases where the shortest paths from multiple origin vertices are required. For instance, the assignment process within macroscopic transport network simulation, which requires all paths originating from each zone. Rather than sequentially finding each shortest path result with independent vertex-frontiers, it is proposed to instead use a novel *origin-vertex-frontier* (OVF). The OVF is the set of origin-vertex pairs which were updated in the previous iteration (initialised to the origin-vertex pair for each initial origin). For instance, for the graph shown in Fig. 3, if the paths from vertices `a`, and `b` were required the OVF would be initialised as `(a,a), (b,b)`. After the first iteration, the OVF would contain `(a,b), (a,c), (a,d), (b, e)` and after would contain `(a, g), (a,f), (a, g), (b, h)` the subsequent iteration. This can lead to a dramatic increase in the number of edges being relaxed within a single iteration, greatly increasing the level of parallelism exposed. Fig. 4 shows the respective single origin vertex-frontier and many-origin origin-vertex-frontier sizes for a large road network containing 5194 zones, as the iterative algorithm progresses. The many-origin frontier contains a much greater number of elements than the single origin frontier and therefore a much higher level of parallelism and GPU utilisation is achieved. The OVF is sorted at each iteration of the algorithm in parallel, by origin and vertex index. This ensures high memory bandwidth through good memory access patterns which the result of coalesced accesses with good locality of data within cache lines.

The frontier contains the vertices which must be considered at each iteration of the algorithm to ensure that all possible updates are found. Each edge connected to the vertices in the frontier must be considered, however the number of edges connected to each

Table 2
The contents of the vertex-frontier for the Bellman-Ford algorithm, for origin vertex `a` of the example graph in Fig. 3.

Iteration	Frontier Vertices
0	a
1	b c d
2	e f g
3	h i
4	i
5	



(a) Vertex Frontier Size for a Single Origin Vertex

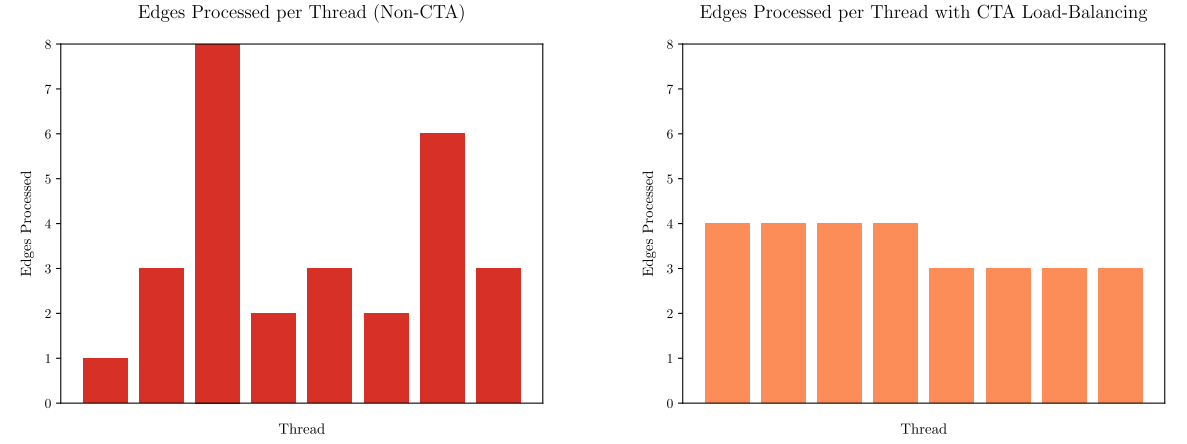
(b) Origin-Vertex Frontier Size for All Origin Vertices

Fig. 4. The size of the vertex-frontier and origin-vertex-frontier at each iteration of the modified Bellman-Ford algorithm for a large road network containing 5194 zones. The frontier size is shown for a single origin vertex and for all 5194 origin vertices. The length of shortest paths through the network varies per origin, and as such the algorithm can complete within fewer iterations for some origin vertices.

vertex varies across the network, leading to an imbalanced work-load. This has a negative impact on performance, due to the SIMD processing architecture at the warp level of the GPU, resulting in divergence due to each thread processing a different number of edges (per iteration in the algorithm). By balancing this work-load across the active threads within a block, performance can be improved.

Using a CTA (Lindholm et al., 2008), a block of threads from the grid-block configuration work collaboratively to relax each edge from a group of vertices in the origin-vertex frontier. From a high level perspective, this can be seen as a re-distribution of all edges to be evaluated over the block of threads, to balance the workload for each thread. The result is a minimisation of divergence within the SIMD warps. A single thread for each origin-vertex in the frontier is used to load the edges to be addressed into a low latency per block shared memory cache. Each thread then accesses the pool of edges to be processed stored in shared memory to find an edge to relax. This is repeated until all edges have been relaxed. Fig. 5 illustrates the load-balancing effect from the use of CTAs, showing the reduction in work load variance between threads and the decreased level of divergence between threads.

Algorithm 2 details the multiple source shortest path algorithm, based on the Bellman Ford algorithm. In our data-parallel implementation of this algorithm, a single GPU kernel is executed to initialise the result arrays (lines 5–14), reducing kernel launch overhead. The for loop used to initialise the origin vertex frontier *OVF*, lines 15–17, is also parallelised using a CUDA kernel. Additionally, lines 20–32 are parallelised as a CUDA kernel. The inner while-loop used to process edges leaving vertices in the OVF (lines 21–31) is implemented using a CTA. This is implemented using shared memory within a block of threads. A binary search is used to select the next edge from shared memory to load-balance the non-uniform degree of vertices within the graph. A single atomic transaction is used to update the results arrays on lines 25–26 to avoid race conditions.



(a) Unbalanced work-load

(b) CTA Balanced work-load

Fig. 5. The load-balancing effect of the cooperative thread array for 28 edges distributed across 8 threads. The imbalance of work has been minimised from a maximum divergence of 7 edges to a single edge.

Algorithm 2. Many Source Shortest Path (MSSP) algorithm based on the Bellman-Ford algorithm, using an Origin-Vertex Frontier for execution on SIMD architectures.

Input:
V is the number of vertices,
E is the number of edges,
Z is the number of zones,
G is Graph with edge costs C,
OD is Origin-Destination Matrix of dimensions Z by Z
Output:
BE is Back-Edge array containing one element per vertex per origin,
BC is Back-Cost array containing one element per vertex per origin
1: Let OVF be the Origin-Vertex Frontier, a set of origin-vertex pairs
2: Let NF be the Next Origin-Vertex Frontier, a set of origin-vertex pairs
{Initialise the frontiers to empty}
3: $OVF \leftarrow \emptyset$
4: $NF \leftarrow \emptyset$
{Initialise the Back-Edge and Back-Cost arrays, using one thread per array element}
5: **gpu parallel for** $idx \leftarrow 1$ to $V \times Z$ **do**
6: Let $v \leftarrow idx \div Z$
7: Let $o \leftarrow idx \bmod Z$
8: **if** $o = v$ **then**
9: $BC(o, v) \leftarrow 0.0$
10: **else**
11: $BC(o, v) \leftarrow \infty$
12: **end if**
13: $BE(o, v) \leftarrow |G_E| + 1$
14: **end for**
{Initialise the origin-vertex frontier with one element per zone}
15: **gpu parallel for** $idx \leftarrow 1$ to Z **do**
16: $OVF.\ append(\{idx, idx\})$
17: **end for**
{While the origin-vertex frontier is not empty, iterate the bellman-ford algorithm}
18: **while** $OVF \neq \emptyset$ **do**
19: $NF \leftarrow \emptyset$
{For each element of the origin vertex frontier, using 1 thread per element}
20: **gpu parallel for** $idx \leftarrow |OVF|$ **do**
{Co-operatively select an edge from the pool of vertex-frontier edges}
21: **while** $e \leftarrow$ edgeFromCooperativePool(idx) **do**
22: $v, v' \leftarrow e$ {Source and destination vertices from edge}
23: $c \leftarrow C(e)$ {Cost of edge}
{If the edge results in a lower cost to the vertex, atomically update}
24: **if** $BC(v) + c < BC(v')$ **then**
25: $BC(v') \leftarrow BC(v) + c$
26: $BE \leftarrow e$
{If the next vertex is not a zone, insert into the next frontier}
27: **if** $v' > Z$ **then**
28: $NF.\ append(\{o, v'\})$
29: **end if**
30: **end if**
31: **end while**
32: **end for**
33: **end while**

4.2. Flow accumulation

The shortest path results are used to calculate the flow per-edge within the network for the demand specified in the Origin-Destination matrix. Within SATURN, flow values are stored as double-precision floating point numbers, to ensure that numerical precision does not result in inaccurate results. This is necessitated by the varying magnitude of values within the OD matrix, which contains both very small and very large numbers.

The simple approach to parallelise this algorithm is for each processing thread to trace a single route through the network, using one thread per trip in the OD matrix, updating the flow-value for each encountered edge in a common architecture. To ensure the results are correct, these additions must be completed atomically using 64-bit floating point precision. Algorithm 3 details this flow accumulation algorithm. Parallelism is applied to the for-each loop on lines 1–9, with a single SIMD thread used to process each trip from the OD matrix. To avoid race conditions between competing threads, the addition operation on line 5 is implemented using an atomic operation, which may experience very high levels of atomic contention, as a large portion of threads may attempt to

concurrently update the same element due to the characteristics of road networks. The results of this algorithm are correct but non-deterministic for networks which, at any point in the iterative process, contain multiple shortest-paths with equivalent total costs. This is a result of the concurrent evaluation of many edges within the network.

Algorithm 3. Flow Accumulation using for SIMD devices with hardware-support for 64-bit atomic addition.

Input:
Z is the number of zones,
G is Graph,
P are Shortest paths from all zones,
OD is Origin-Destination Matrix of dimensions Z by Z
Output:
F is Flow per edge in G
{For each trip in the matrix, in parallel using one thread per trip}
1: **gpu parallel for** trip t from origin o to destination d **in** OD **do**
2: $f \leftarrow OD(t)$
3: $e \leftarrow$ call PredecessorEdgeFromVertex(o, d, P, G)
{While the next edge in the trip is valid, accumulate flow}
4: **while** $e \leqslant |G_E|$ **do**
5: $F(e) \leftarrow F(e) + f$
6: $v, v' \leftarrow e$
7: $e \leftarrow$ call PredecessorEdgeFromVertex(o, v, P, G)
8: **end while**
9: **end for**

The atomic addition operation for double-precision floating point numbers is only implemented as a hardware instruction in NVIDIA GPUs beginning with the Pascal architecture (Compute Capability 6.0 and above) and CUDA 8.0 (NVIDIA Corporation, 2016). Previous architecture generations must instead use a software-based workaround using a *critical section*, which comes with a significant performance penalty. The performance degradation caused by the highly contested critical section is often sufficient to negate any performance improvement from calculating shortest paths on the GPU.

To support older GPU architectures, and avoid the use of critical sections which serialise the execution of threads to ensure correctness, an alternative flow accumulation algorithm was devised, as shown in Algorithm 4. This algorithm minimises the use of the expensive software-based double-precision operation, to reduce atomic contention. A trip-frontier, a set of active trips from the trip matrix which are to be processed containing the trip id and the next edge to be processed, enables active paths to be traced one edge at a time and is initialised to contain all trips with a valid path (lines 4–9). The trip frontier is sorted by the edge index within the frontier element (line 10) to align memory for reduction. While the trip-frontier is not empty (lines 11–24), a single step of the path for each trip in the frontier is processed concurrently, by paralellising the for loop on lines 13–21. If origin vertex has not yet been reached, the trip and next edge are inserted into the next trip-frontier (lines 16–18), which is later sorted (lines 22) and used as the trip frontier on the next iteration.

As this is parallelised for the for-each loop on lines 13–21, the addition operation on line 20 must be performed atomically, as the results data structure is shared by all threads. To reduce the number of software-based atomic operations, the flow values for each edge are accumulated into a single value within the thread-block. This reduction is possible as the trip frontier is sorted by edge, allowing parallel primitives such as the inclusive prefix sum to be used to reduce the value, operating using warp and shared memory reductions. A single thread for each edge within the block then performs an atomic addition operation to update the result array, greatly reducing the number of atomic operations required. When using warp and shared memory reductions with a maximum block size of 1024 threads, the number of atomic operations can be reduced by up to a factor of 1023.

Algorithm 4. Flow Accumulation Algorithm for SIMD hardware devices without double precision atomic addition support

Input:
Z is the number of zones,
G is the Graph,
P are the shortest paths from all zones,
OD is the Origin-Destination Matrix of dimensions Z by Z
Output:
F is the flow per edge in G
1: Let TF be the trip-frontier, a set of trip-edge pairs
2: Let NF be the next trip-frontier, a set of trip-edge pairs
3: Let BLF be local flow value within shared memory at the block level {Initialise the trip-frontier TF with all valid trips in the OD matrix}
4: **gpu parallel for** trip t from origin o to destination d **in** OD **do**
5: $e \leftarrow$ call get-edge-from-path(o, d, P, G)
6: **if** $e \leqslant |G_E|$ **then**
7: $TF.\ append(\{t, e\})$
8: **end if**
9: **end for**

```
    {Sort the trip frontier so edges are in the same block of threads}
10: parallel sort TF by e
    {Trace each trip in parallel, one edge at a time}
11: while TF ≠ ∅ do
12:   NF ← ∅ {Reset the next frontier}
      {Update the flow for each trip in the frontier}
13:     gpu parallel for (t, e) in TF do
14:       f ← OD(t)
15:       e' ← call PredecessorEdgeFromVertex(o, d, P, G)
        {Update the next trip frontier if required}
16:         if e' < = |G_E| then
17:           NF. append({t, e'})
18:       end if
        {Accumulate flow for edge at the block level}
19:       BLF(e) ← blockReduce(f)
20:       F(e) ← F(e) + BLF(e)
21:     end for
22:     parallel sort NF by e
23:     TF ← NF
24: end while
```

5. Results

To assess the performance and impact of our GPU accelerated software implementation on real-world transportation networks, a set of three networks used by transport authorities in the UK were used as benchmark models. Table 3 gives the details of the three models (Derby, CLoHAM & LoHAM) which vary in size with varying numbers of zones, vertices, edges and user classes. Table 4 describes the computing hardware configurations used to benchmark the application. Several sets of hardware were used to evaluate performance, with multiple CPUs and GPUs used in multiple combinations.

The existing serial and multi-core CPU versions of the SATALL application were benchmarked using each CPU. The GPU accelerated application was benchmarked using each CPU-GPU combination. Table 5 and Fig. 6 show the results of these benchmarks. To ensure that the results of our GPU accelerated models are correct the output of simulations using the GPU have been cross-validated against reference CPU results, using metrics commonly used to validate SATURN models (Bradley et al., 2016). The validation results can be seen in Table 6, with all validation metrics falling within 2% of the reference multi-core CPU-based application.

The benchmark performance results from Table 5 and Fig. 6 show that our GPU-based implementation has drastically reduced the runtime for the largest model (LoHAM) compared to both serial and multi-core CPU implementations, of up to 53x and 7.8x, respectively. The smaller CLoHAM and Derby models do not show as significant increases in performance, showing up to 23x and 4.1x for the CLOHAM model compared to serial and multi-core implementations, and 8.6x and 2.5x speed-up for Derby. This can be attributed to the still relatively low level of parallelism exposed by the smaller networks, which do not fully occupy the highly-parallel many-core processing architecture consistently throughout the application. Additionally, the overhead costs associated with transferring data to and from the GPU is a larger portion of the total runtime for these smaller models. Furthermore, it is worth noting that the multi-core SATURN achieves super-linear speedup compared to the single-core implementation on system I1 (see Table 4) using 4 processing threads for the larger CLoHAM and LoHAM models. This can be attributed to the use of the Intel compiler for the OpenMP-enabled portions of code, which may apply different code optimisations during compilation, resulting in a higher-performance executable.

Between the different generations of GPU, we see much greater performance from the newer Pascal architecture NVIDIA Titan X compared to the Maxwell architecture Geforce GTX Titan X. The Pascal Titan X achieves an assignment speed-up of 2.0x for the LoHAM model compared to the Maxwell-based GPU, with speed-ups of 1.6x and 1.6x for the CLoHAM and Derby models respectively. This can be attributed to both (i) general performance improvements of the new architecture (processor frequency, core count, memory bandwidth, etc.) and (ii) hardware-support for double-precision atomic-addition, critical to the flow-accumulation process.

6. Discussion and future work

We have shown that GPUs are suitable for macroscopic road network assignment and can provide a significant performance improvement, through the use of fine-grained data-parallel algorithms. However, there is still potentially room for further improvement by increasing the level of parallelisation, but this will require further changes to the application.

Table 3
The real-world networks used to benchmark real-world performance.

Network	Zones	Vertices	Edges	User Classes
Derby	547	2700	25,385	9
CLoHAM	2548	15,179	132,600	5
LoHAM	5194	18,427	192,711	5

Table 4
The hardware used to benchmark real-world performance. *I1* and *X1* are do not contain dedicated GPUs and are used for reference CPU results.

ID	CPU	CPU Freq (GHz)	CPU cores	Memory	GPU
I1	i7-4770K	3.5	4	16 GB DDR3	
I2	i7-4770K	3.5	4	16 GB DDR3	Geforce GTX Titan X (Maxwell)
I3	i7-4770K	3.5	4	16 GB DDR3	Titan X (Pascal)
X1	E5-2667	2.9	6	32 GB DDR3	
X2	E5-2667	2.9	6	32 GB DDR3	Titan X (Maxwell)

Table 5
Performance data for real-world benchmarks.

Hardware	Version	Assignment Time (s)			Total Time (s)		
		Derby	CLoHAM	LoHAM	Derby	CLoHAM	LoHAM
I1	Single-Core	625.0	10400.9	43943.9	997.8	10653.6	44632.2
I1	Multi-Core x4	182.6	1887.0	6456.2	376.4	2172.4	7117.410
I2	GPU	113.2	705.3	1657.5	208.9	828.6	1993.4
I3	GPU	72.6	452.2	826.8	153.3	586.9	1151.5
X1	Single-Core	1085.2	14656.0	54500.8	1703.5	15046.1	55441.8
X1	Multi-Core x16	233.1	1848.0	5596.1	487.2	2236.0	6495.5

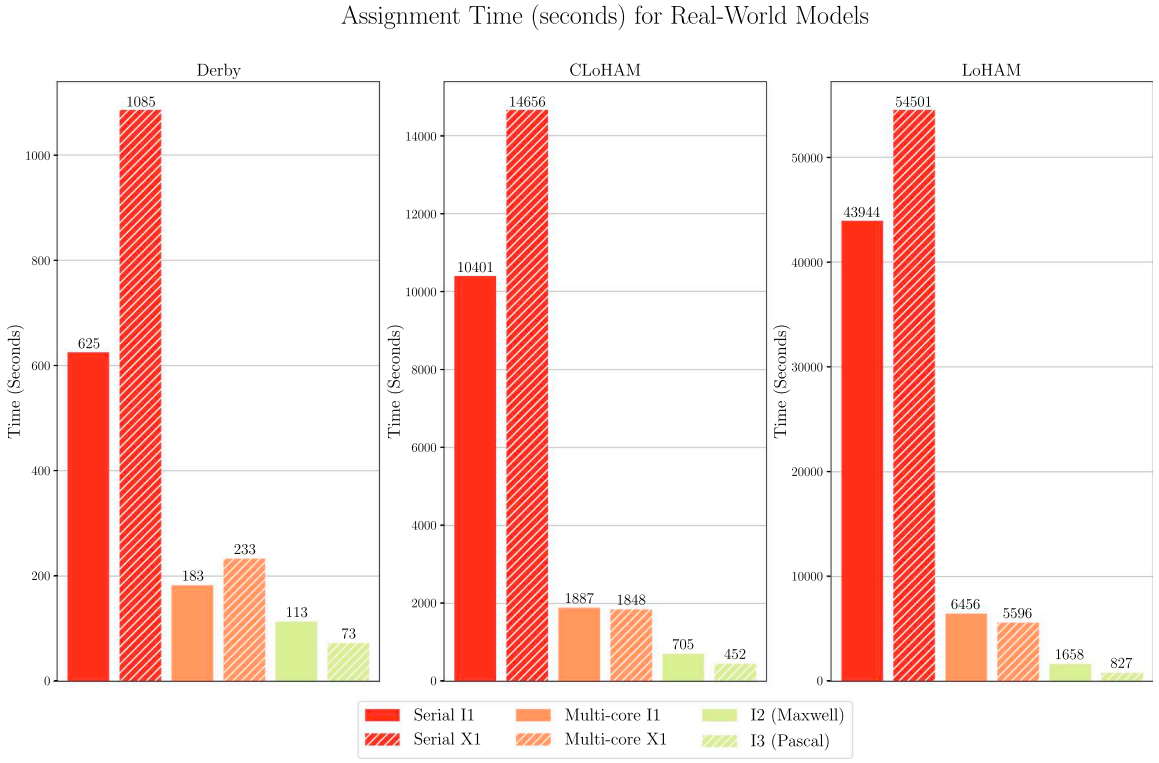


Fig. 6. The runtime of the assignment phase of SATALL for the three real world models on a range of hardware. Lower times are better.

Smaller transport networks do not expose enough work to fully occupy a modern GPU, and therefore cannot access all of the available performance improvements. Currently the assignment within SATALL has only been parallelised at the innermost level (see Algorithm 1). This process is sequentially repeated for each user class of vehicles within the model, with slightly different edge-costs for the different user classes, to represent the physical restrictions imposed on different vehicle types by the transportation network. By processing all user classes concurrently it is envisaged that run-time may be further reduced. This will allow smaller models to use a greater portion of the massively-parallel processing hardware, but may require the use of multiple GPUs for larger models due to memory limitations.

Table 6
Validation results for the LoHAM real-world models, comparing the results of GPU accelerated simulations against the reference multi-core CPU simulation.

Measure	Target	CPU (Reference)	GPU	Difference
Links – GEH < 5	85%	64%	64%	0%
Links – GEH < 7.5	85%	78%	78%	0%
Links – DMRB Flow Criteria	85%	74%	74%	0%
Screenline – Flow Difference < 5%	85%	90%	90%	0%
Enclosure – Flow Difference < 5%	85%	94%	96%	+2%
Mini screenline – GEH < 5	85%	91%	91%	0%
JT Routes – Time Difference < 15%	85%	92.1%	92.6%	+0.5%
Links – GEH < 5	85%	64%	64%	0%

GPU accelerated graph processing algorithms tend to favour larger, denser graphs which expose greater levels of parallelism, as shown by Gunrock where GPUs showed significantly greater speed-ups for social graphs than road networks when compared to CPU implementations (Wang, 2014). The graph used to represent the physical transport network being modelled is preprocessed within SATURN, referred to as *SPIDER* network generation (see Section 3). This process was developed to increase the performance of SATURN, with the D'Esopo-Pape (Pape, 1974) SSSP algorithm in mind. As the underlying SSSP algorithm has been modified and parallelised using a data-parallel algorithm, it may also be beneficial to modify the pre-processing phase, including SPIDER network generation, to produce graphs more favourable to the Bellman-Ford based algorithm. Typically higher-order, lower-radius networks show increased levels of performance for data-parallel implementations of the Bellman-Ford algorithm (Wang, 2014). This will of course reduce overall parallelism, so some balancing of the SPIDER generation process will be required.

After the shortest path calculation had been accelerated using GPUs, the flow accumulation process became the performance-limiting factor. Although this is also now accelerated using GPUs, and the shortest path calculations once again dominate the processing requirements of the application, the ratio of work between the two processes is not the same as in the CPU-based implementation. The flow accumulation process shows a smaller speed-up than the shortest path computation, due to the memory-intensive nature of the task.

As graphics processing hardware continues to evolve at a high rate, newer generations of hardware will provide greater levels of parallelism, higher per-processing-core performance and higher memory bandwidth. The implementation of our many source shortest path algorithm is performance limited by memory bandwidth. Newer generations of hardware which use new memory technologies such as HBM2, used in GP100 and GV100 GPUs (NVIDIA Corporation, 2016), with significant increases in memory bandwidth, will provide considerable performance improvement, without the need for any changes to software. Additionally as newer generations of hardware contain greater numbers of processing cores, the considerable level of parallelism exposed by solving the shortest path problem for many sources concurrently will result in further performance improvements. Our software is not designed to target a specific number of processing cores, but rather expose as much parallelism as possible to allow future growth, and therefore improved scaling behaviour.

Further performance improvements could be achieved for sufficiently large work-loads by using greater numbers of GPUs, through high-density GPU nodes such as the NVIDIA DGX-2 (NVIDIA, 2018) or through distributed computing. Models containing large numbers of user-classes or of significantly larger scale, would be able to leverage additional GPUs to improve performance, whilst still providing each GPU with sufficient work to achieve high utilisation. Emerging very-high density GPU nodes such as the DGX-2 system contain up to 16 GPUs and include high-bandwidth, relatively low-latency, GPU interconnect such as NVSwitch or NVLink, which minimise the performance impact of additional communication and synchronisation required when using more GPUs. Alternatively, distributed systems can offer greater numbers of GPUs, which are connected through relatively high-latency HPC interconnect systems such as InfiniBand or Ethernet. This approach could offer much greater numbers of GPUs than available for a single system, but with much higher cost inter-node communication and synchronisation. At the current scale of transport network being modelled it is unlikely that a performance improvement would be achieved.

The transport networks used to benchmark the algorithmic changes are real-world models used in the UK, by practitioners who use SATURN for modelling. Although considerable performance improvements have been shown it is difficult to fairly compare the simulator performance to other macroscopic simulation tools using these networks. A standardised set of transport networks which could be openly used to compare and contrast competing software tools, algorithms and methodologies would be instrumental in enabling transport modellers and developers to more effectively choose the most appropriate tool.

7. Conclusions

A novel enhancement to vertex-frontier-based Graphics Processing Unit (GPU) implementations of the Bellman-Ford algorithm is proposed to improve the performance of shortest path calculations for low-density, high-diameter graphs typical of transport networks. Through the use of an *Origin-Vertex Frontier (OVF)* the degree of parallelism exposed is increased when solving the shortest path problem for many origin vertices concurrently. By improving the degree of parallelism exposed to the GPU, the algorithm is able to fully-utilise the processing hardware, which requires many thousands of concurrent threads, compared to single origin shortest path algorithms. This provides a performance improvement graphs such as transport networks, which are neither large enough or

dense enough for GPU-based implementations to provide a performance improvement compared to work-efficient algorithms on CPUs.

The algorithm has been implemented within SATURN, a modern macroscopic assignment-simulation transport modelling application, which relies on the calculation of shortest paths between all zones within the model. Additionally, the use of the shortest path results to accumulate per-edge flow values for all trips within an Origin-Demand matrix was parallelised, using alternate techniques based on the instruction sets available to different GPU architectures.

The performance of the GPU accelerated version of SATURN has been benchmarked and compared to the more traditional CPU-based implementations of the application for a set of real-world road networks used within industry in the UK. The results showed speed-ups of up to 7.8x when compared to the multi-threaded CPU implementation, and up to 53x compared to a serial CPU implementation for the assignment phase of SATALL using typical hardware.

The benefits and applicability of GPUs as a tool to accelerate transport simulation software have been shown and the requirement of changing algorithms from the commonly used serial algorithms to less-efficient but highly parallel algorithms is shown to fully embrace parallelism and gain access to the available performance. The required algorithmic change to leverage the massively-parallel GPU architecture is a process that may be applicable to other aspects of transportation planning and other fields hoping to harness the computational performance of GPUs. By providing greater levels of performance and reducing the time required for individual simulations to complete, modellers will be able to make better use of their time, evaluate a greater number of scenarios when comparing alternate traffic management strategies or infrastructure and evaluate a broader range of environmental conditions rather than a single or a few aggregate scenarios.

Acknowledgements

This research has been supported through a 50/50 public/private funding agreement between by Atkins Ltd and Highways England, with the public funding awarded jointly to the Transport Systems Catapult, the University of Sheffield and the Science and Technology Facilities Council. Additional support has been committed through an EPSRC Fellowship for Dr. Richmond (EP/N018869/1). The SATURN software has been provided via Atkins Ltd. Transport network data was provided by Highways England and Transport for London.

The authors would also like to thank Dirck Van Vliet for his guidance and input.

References

Aboudina, A., Kamel, I., Elshenawy, M., Abdelgawad, H., Abdulhai, B., 2018. Harnessing the power of hpc in simulation and optimization of large transportation networks: Spatio-temporal traffic management in the greater Toronto area. IEEE Intell. Transp. Syst. Mag. 10 (1), 95–106.

Antoniou, C., Barcelò, J., Brackstone, M., Celikoglu, H., Ciuffo, B., Punzo, V., Sykes, P., Toledo, T., Vortisch, P., Wagner, P., 2014. Traffic simulation: case for guidelines.

Auld, J., Hope, M., Ley, H., Sokolov, V., Xu, B., Zhang, K., 2016. Polaris: Agent-based modeling framework development and implementation for integrated travel demand and network and operations simulations. Transp. Res. Part C: Emerg. Technol. 64, 101–116.

Balmer, M., Rieser, M., Meister, K., Charypar, D., Lefebvre, N., Nagel, K., 2009. Matsim-t: architecture and simulation times. In: Multi-agent Systems for Traffic and Transportation Engineering. IGI Global, pp. 57–78.

Bannister, M.J., Eppstein, D., 2012. Randomized speedup of the Bellman-Ford algorithm. In: Proceedings of the Meeting on Analytic Algorithmics and Combinatorics. Society for Industrial and Applied Mathematics, pp. 41–47.

Barceló, J., Casas, J., 2005. Dynamic network simulation with aimsun. In: Simulation Approaches in Transportation Analysis. Springer, pp. 57–98.

Bellman, R., 1958. On a routing problem. Quart. Appl. Math. 87–90.

Bradley, R., Wright, I., Swain, D., Himlin, R., Heywood, P., Richmond, P., Mawson, M., Fletcher, G., Guichard, R., 2016. Accelerating traffic models using GPU-based technology. In: European Transport Conference 2016Association for European Transport (AET).

Busato, F., Bombieri, N., 2015. Bfs-4k: an efficient implementation of bfs for kepler gpu architectures. IEEE Trans. Parallel Distrib. Syst. 26 (7), 1826–1838.

Cox, A., 2016. Highways Englands Regional Traffic Models challenges over the past year. In: SATURN User Group Annual Meeting November 2016, URL. https://saturnsoftware2.co.uk/uploads/files/SATUGM16-Highways-England-Regional-Models-Update.pdf.

Crauser, A., Mehlhorn, K., Meyer, U., Sanders, P., 1998. A parallelization of dijkstra's shortest path algorithm. In: International Symposium on Mathematical Foundations of Computer Science. Springer, pp. 722–731.

Dagum, L., Menon, R., 1998. Openmp: an industry standard api for shared-memory programming. IEEE Comput. Sci. Eng. 5 (1), 46–55.

Davidson, A., Baxter, S., Garland, M., Owens, J.D., 2014. Work-efficient parallel GPU methods for single-source shortest paths. In: Parallel and Distributed Processing Symposium, 2014 IEEE 28th International. IEEE, pp. 349–359.

Dial, R.B., 1969. Algorithm 360: shortest-path forest with topological ordering [H]. Commun. ACM 12 (11), 632–633.

Dijkstra, E.W., 1959. A note on two problems in connexion with graphs. Nume. Math. 1 (1), 269–271.

Fellendorf, M., 1994. Vissim: a microscopic simulation tool to evaluate actuated signal control including bus priority. In: 64th Institute of Transportation Engineers Annual Meeting. Springer, pp. 1–9.

Floyd, R.W., 1962. Algorithm 97: shortest path. Commun. ACM 5 (6), 345.

Flynn, M.J., 1972. Some computer organizations and their effectiveness. IEEE Trans. Comput. 100 (9), 948–960.

Ford Jr, L.R., 1956. Network flow theory. Tech. rep. DTIC Document.

Fredman, M.L., Tarjan, R.E., 1987. Fibonacci heaps and their uses in improved network optimization algorithms. J. ACM (JACM) 34 (3), 596–615.

Frejo, J.R.D., Papamichail, I., Papageorgiou, M., De Schutter, B., 2019. Macroscopic modeling of variable speed limits on freeways. Transp. Res. Part C: Emerg. Technol. 100, 15–33.

Głkabowski, M., Musznicki, B., Nowak, P., Zwierzykowski, P., 2013. Efficiency evaluation of shortest path algorithms. In: AICT 2013, The Ninth Advanced International Conference on Telecommunications, pp. 154–160.

Green 500, Aug. 2016. Green 500 website. https://www.top500.org/green500/ (last Accessed 2016-08-01).

Gregor, D., Lumsdaine, A., 2005. The parallel BGL: a generic library for distributed graph computations. Parallel Object-Oriented Sci. Comput. (POOSC) 2, 1–18.

Hall, M., Van Vliet, D., Willumsen, L., 1980. SATURN – a simulation-assignment model for the evaluation of traffic management schemes. Traffic Eng. Control 21, (4).

Hao, Z., Boel, R., Li, Z., 2018. Model based urban traffic control, part ii: coordinated model predictive controllers. Transp. Res. Part C: Emerg. Technol. 97, 23–44.

Heywood, P., Maddock, S., Casas, J., Garcia, D., Brackstone, M., Richmond, P., 2017. Data-parallel agent-based microscopic road network simulation using graphics processing units. Simul. Model. Practice Theory URL. http://www.sciencedirect.com/science/article/pii/S1569190X17301545.

Hunter, M., Kim, H.K., Suh, W., Fujimoto, R., Sirichoke, J., Palekar, M., 2009. Ad hoc distributed dynamic data-driven simulations of surface transportation systems. Simulation 85 (4), 243–255.
Johnson, D.B., 1977. Efficient algorithms for shortest paths in sparse networks. J. ACM (JACM) 24 (1), 1–13.
Johnson, P., Nguyen, D., Ng, M., 2016. Large-scale network partitioning for decentralized traffic management and other transportation applications. J. Intel. Transp. Syst. 20 (5), 461–473.
Khorasani, F., Vora, K., Gupta, R., Bhuyan, L.N., 2014. CuSha: vertex-centric graph processing on GPUs. In: Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing. ACM, pp. 239–252.
Lawson, G., Allen, S., Rose, G., Nguyen, D., Ng, M., 2013. Parallel label correcting algorithms for large-scale static and dynamic transportation networks on laptop personal computers. In: Transportation Research Board (TRB) 2013 Annual Meeting (Washington, DC.
Li, P.T., Wang, P., Chowdhury, F.R., Zhang, L., 2019. A scalable computing architecture for solving time-dependent transportation problems based on high-performance computing techniques. In: Transportation Research Board (TRB) 2019 Annual Meeting.
Lindholm, E., Nickolls, J., Oberman, S., Montrym, J., 2008. NVIDIA Tesla: a unified graphics and computing architecture. IEEE Micro 28 (2).
Ma, W., Qian, Z.S., 2018. Estimating multi-year 24/7 origin-destination demand using high-granular multi-source traffic data. Transp. Res. Part C: Emerg. Technol. 96, 96–121.
NVIDIA, 2009. Nvidia's next generation cuda compute architecture: Fermi.
NVIDIA, 2012. Cuda c best practices guide. NVIDIA, Santa Clara, CA.
NVIDIA, 2016. nvGRAPH. https://developer.nvidia.com/nvgraph (last accessed 2016-08-31).
NVIDIA, 2018. Nvidia dgx-2 datasheet (last accessed 2018–11-30). http://images.nvidia.com/content/pdf/dgx-2-datasheet-us-nvidia-848202-r3-web.pdf.
Nvidia, C., Mar. 2015. CUDA C programming guide. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf (last accessed 2017-08-02). http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.
NVIDIA Corporation, 2016. NVIDIA Tesla P100 (whitepaper). https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.
Ortega-Arranz, H., Torres, Y., Llanos, D.R., Gonzalez-Escribano, A., 2013. A new gpu-based approach to the shortest path problem.
Pape, U., 1974. Implementation and efficiency of Moore-algorithms for the shortest route problem. Math. Program. 7 (1), 212–222.
Ptv, A.G., 2017. Ptv visum 17 new features at a glance. URL. http://vision-traffic.ptvgroup.com/fileadmin/files_ptvvision/Downloads_N/0_General/2_Products/1_PTV_Visum/Overview_PTVVisum17_EN.PDF.
Qu, Y., Zhou, X., 2017. Large-scale dynamic transportation network simulation: a space-time-event parallel computing approach. Transp. Res. Part C: Emerg. Technol. 75, 1–16.
Siek, J.G., Lee, L.-Q., Lumsdaine, A., 2001. Boost Graph Library: User Guide and Reference Manual. The Pearson Education.
Top 500, Aug. 2016. Top 500 website. https://www.top500.org/ (last accessed 2016-08-01).
Van Vliet, D., 1978. Improved shortest path algorithms for transport networks. Transp. Res. 12 (1), 7–20.
Van Vliet, D., 1982. SATURN - a modern assignment model. Traffic Eng. Control 23 (HS-034 256).
Van Vliet, D., 2015. SATURN: Version 11.3 Manual.
Wang, Y., 2014. High-performance graph primitives on the GPU: design and implementation of gunrock. In: GPU Technology Conference.
Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A., Owens, J.D., 2016. Gunrock: a high-performance graph processing library on the GPU. In: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM, pp. 11.
Wardrop, J.G., 1952. Some theoretical aspects of road traffic research. Proc. Inst. Civ. Eng. 1 (3), 325–362.
Zhan, F.B., Noon, C.E., 1998. Shortest path algorithms: an evaluation using real road networks. Transp. Sci. 32 (1), 65–73.