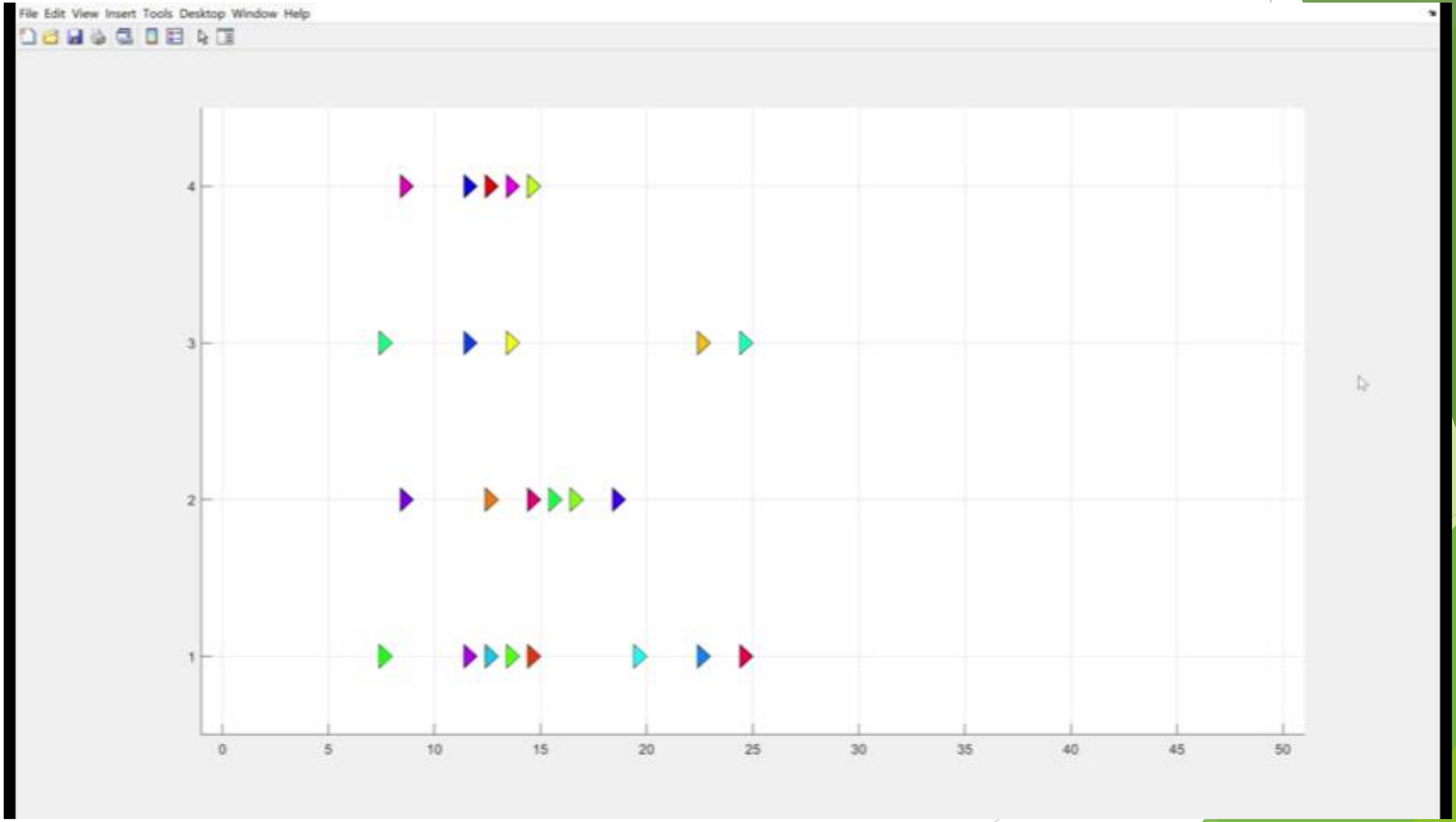
Traffic Simulation

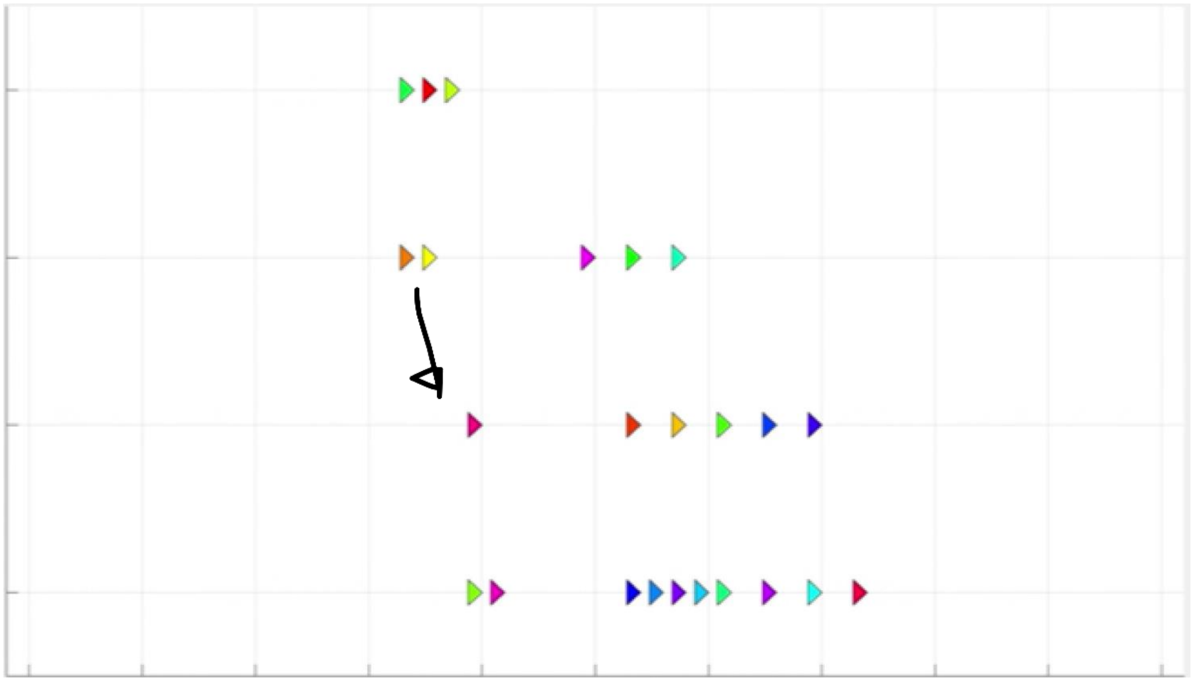
Tommy Poek

47192085

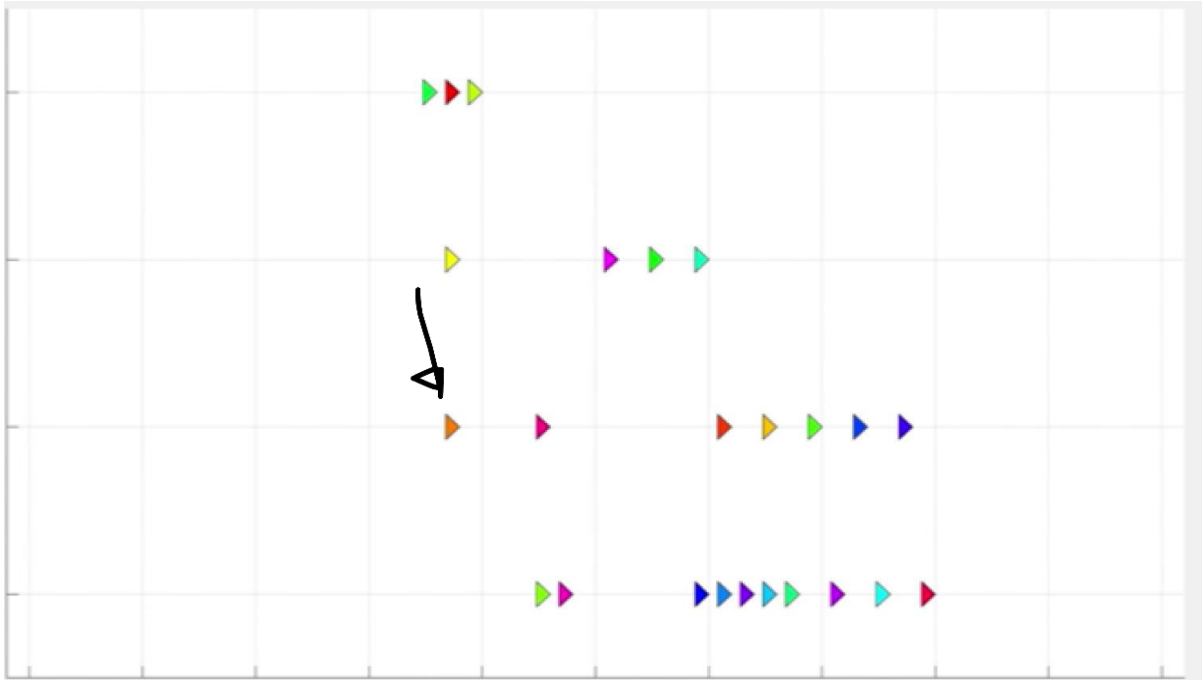
Intro – What this project is



Intro – Visualization

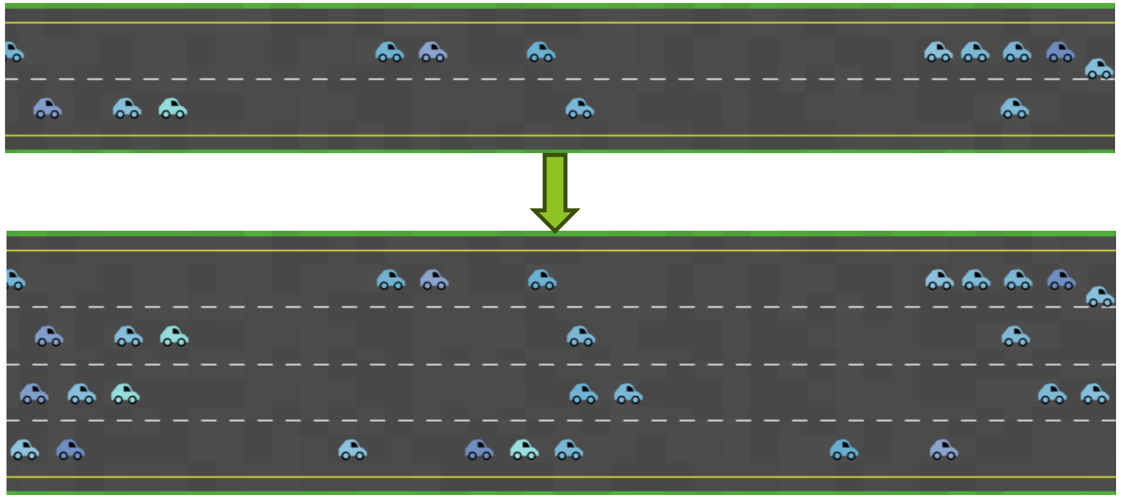


Intro – Visualization

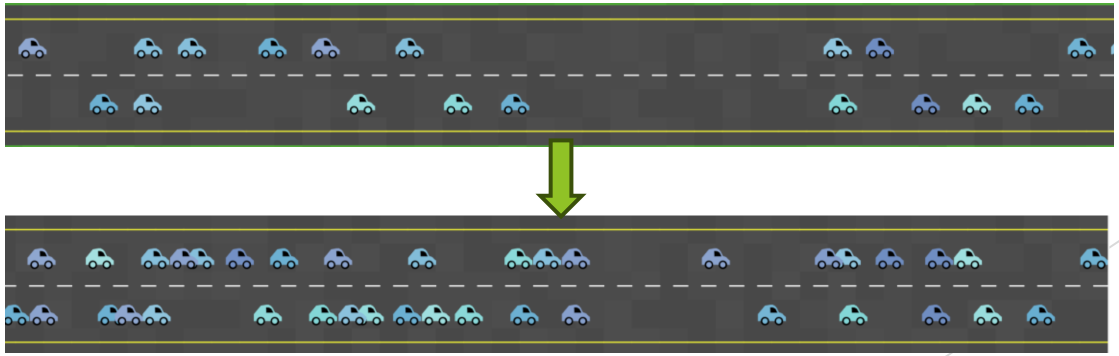


Intro – What we want to scale

- A one-way traffic
  - Scales up NUM_LANES

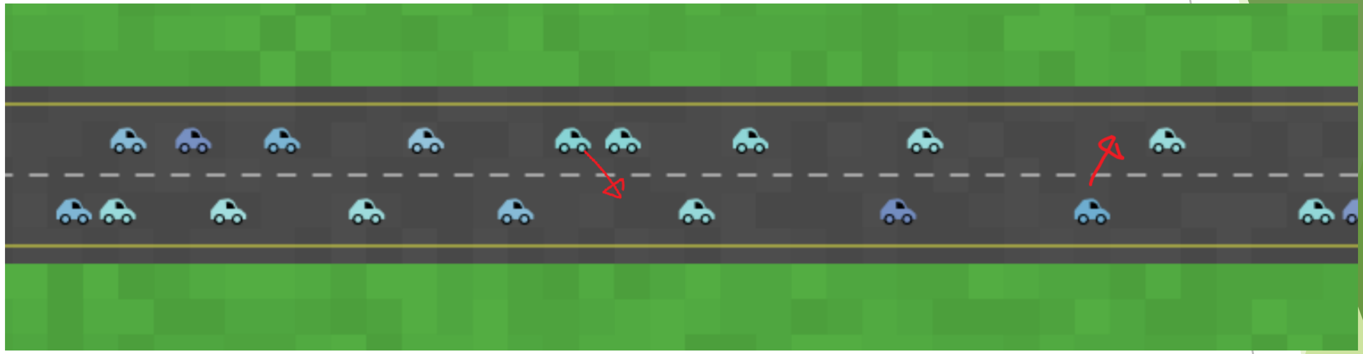


  - Scales up NUM_CARS

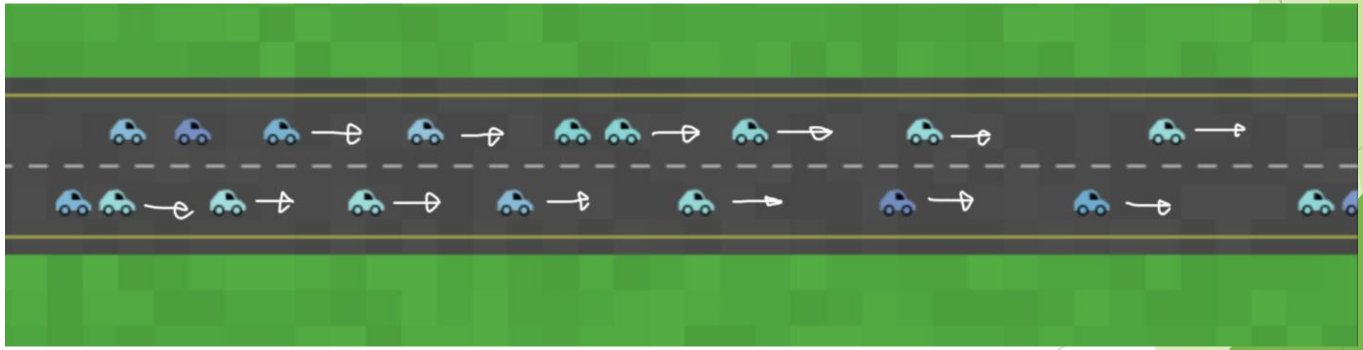


Intro – How the model works

- for each simulation step:
    - **// All cars try lane change**
    - for each lane:
        - // clock starts
        - for each car:
            - …
        - // time the clock



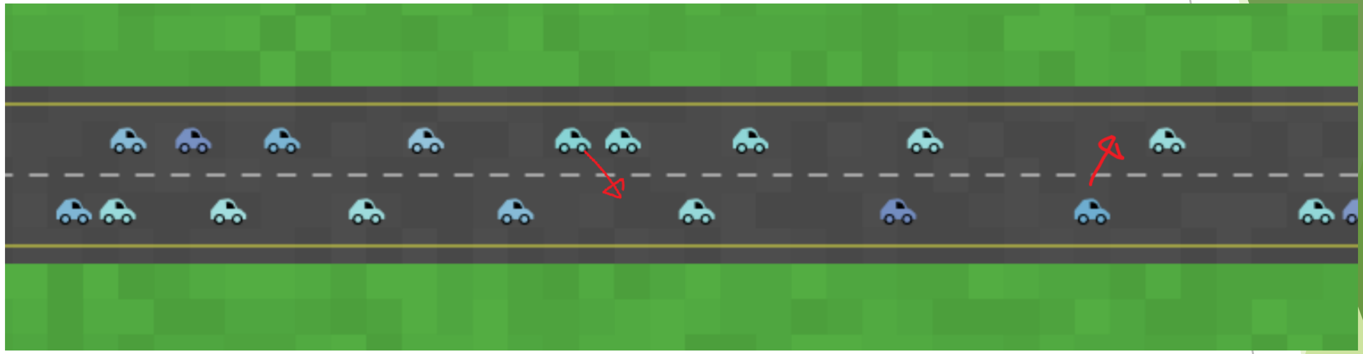
-
    - **// All cars drive forward**
    - for each lane:
        - // clock starts
        - for each car:
            - …
        - // time the clock

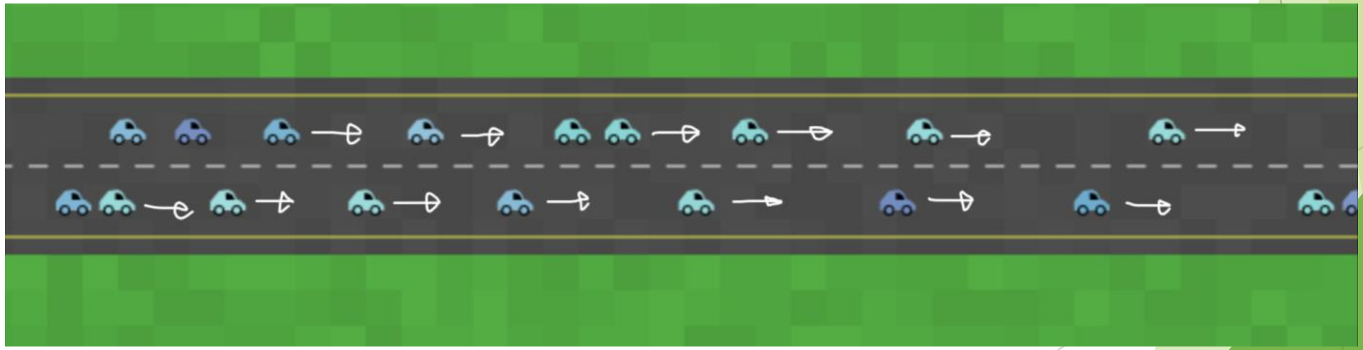


Intro – How the model works

- for each simulation step:
    - **// All cars try lane change**
    - for each lane:
        - // clock starts
        - for each car:
            - …
        - // time the clock
    - **// All cars drive forward**
    - for each lane:
        - // clock starts
        - for each car:
            - …
        - // time the clock

Functions to benchmark





Implementations + Optimizations

- v1: Traffic = Lanes[],
  - Lane = GridSpaces[], GridSpace = 0(no car) / >0(speed of car)
- v2: Traffic = Lanes[],
  - Lane = Cars[], Car = struct{Position, Speed}
- v3: Traffic = Cars[],
  - Lane = CarsIndices[], Car = struct{Position, Speed}

Implementations + Optimizations

- ~~v1: Traffic = Lanes[],~~
  - ~~Lane = GridSpaces[], GridSpace = 0(no car) / >0(speed of car)~~

Implementation failed

- v2: Traffic = Lanes[],
  - Lane = Cars[], Car = struct{Position, Speed}
- v3: Traffic = Cars[],
  - Lane = CarsIndices[], Car = struct{Position, Speed}

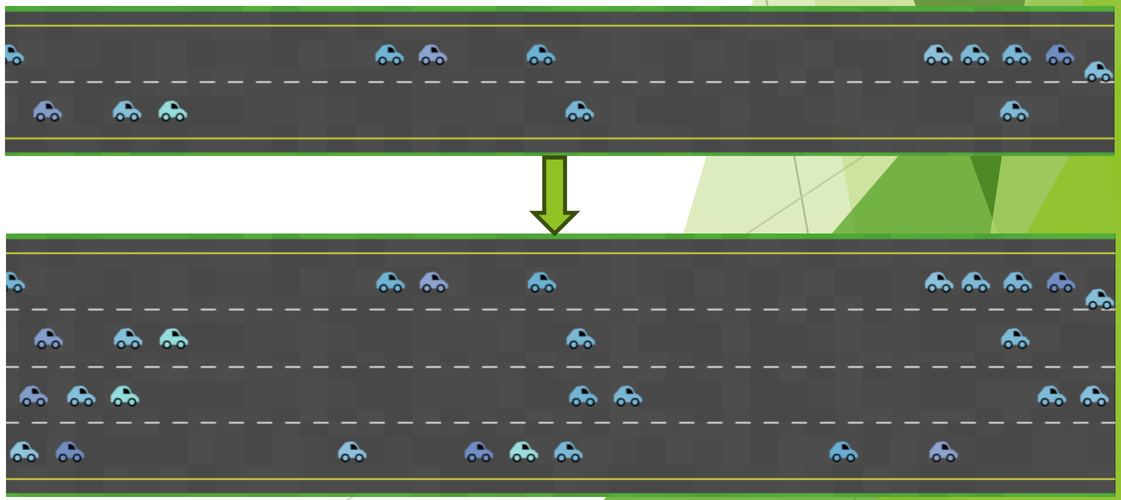
Implementations + Optimizations

- v2: Traffic = Lanes[],
  - Lane = Cars[], Car = struct{Position, Speed}, with no optimization flag (-O0)
  - v2.1: v2 with –O1 optimization
  - v2.2: v2 with –O2 optimization
- v3: Traffic = Cars[],
  - Lane = CarsIndices[], Car = struct{Position, Speed}, with no optimization flag (-O0)
  - v3.1: v3 with –O1 optimization
  - v3.2: v3 with –O2 optimization

To benchmark!

Benchmarking – V2 –O0, 20 Cars

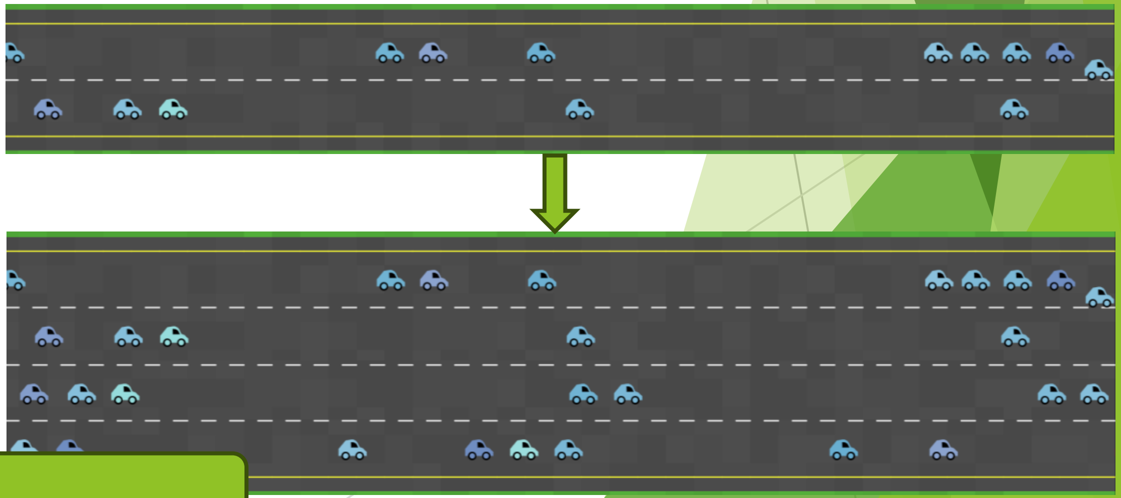
#Lanes	Cumulative runtime (us) // all cars try lane change	Cumulative runtime (us) // all cars drive forward
2 Lanes	37121	2628
4 Lanes	38465	2504
8 Lanes	18825	2647
16 Lanes	4735	3469



Benchmarking – V2 –O0, 20 Cars

#Lanes	Cumulative runtime (us) // all cars try lane change	Cumulative runtime (us) // all cars drive forward
2 Lanes	37121	2628
4 Lanes	38465	2504
8 Lanes	18825	2647
16 Lanes	4735	3469

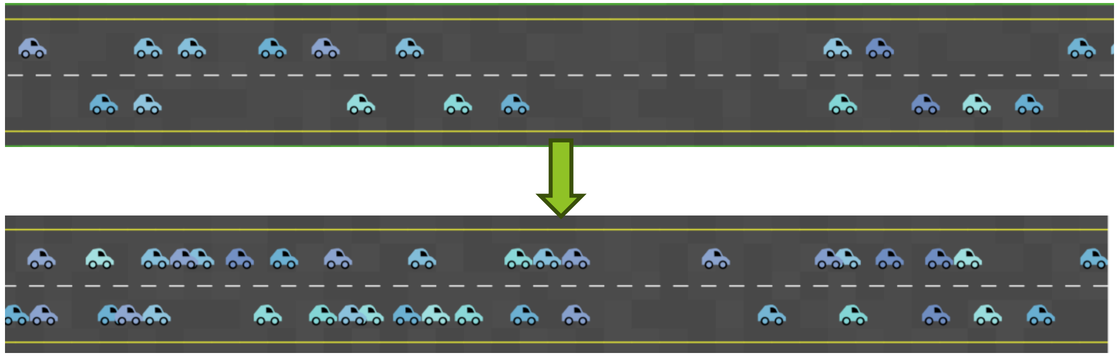
More lanes -> sparser traffic -> fewer lane change -> shorter runtime

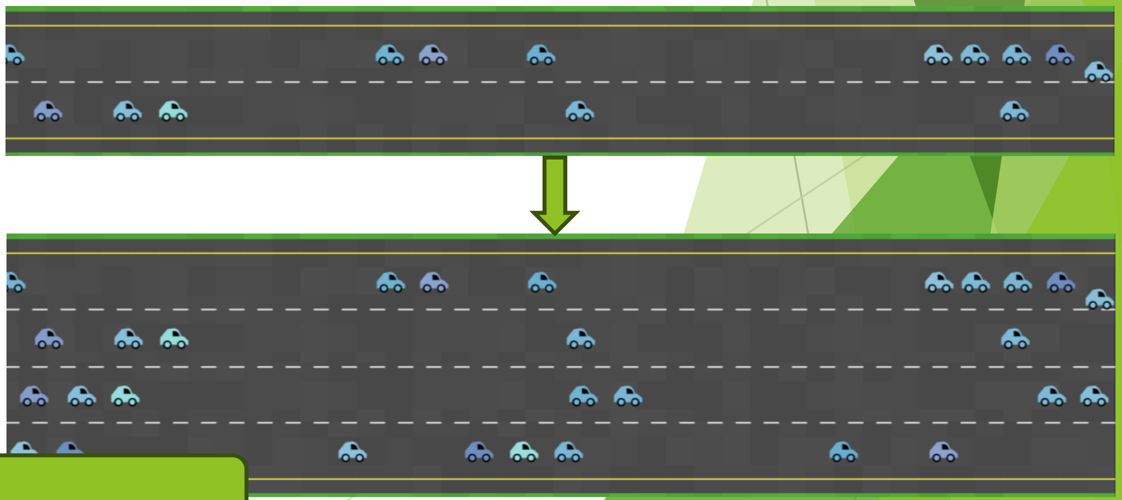


What if we scale up the traffic size? (both #lanes and #cars)

Benchmarking – V2 –O0

#Lanes	#Cars	Cumulative runtime (us) // all cars try lane change	Cumulative runtime (us) // all cars drive forward
2 Lanes	50 Cars	616116	279713
4 Lanes	100 Cars	1739478	574062
8 Lanes	200 Cars	3928817	1234818
16 Lanes	400 Cars	10379542	2672994
32 Lanes	800 Cars	26368452	12223719

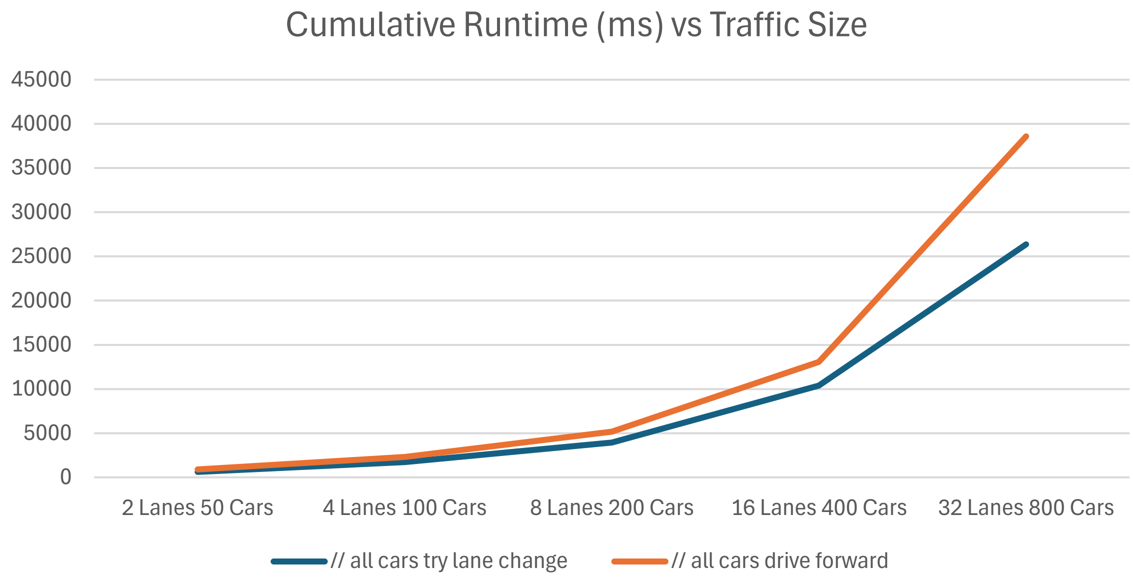




Scale up the traffic size: Runtime increases accordingly.

Benchmarking – V2 –O0

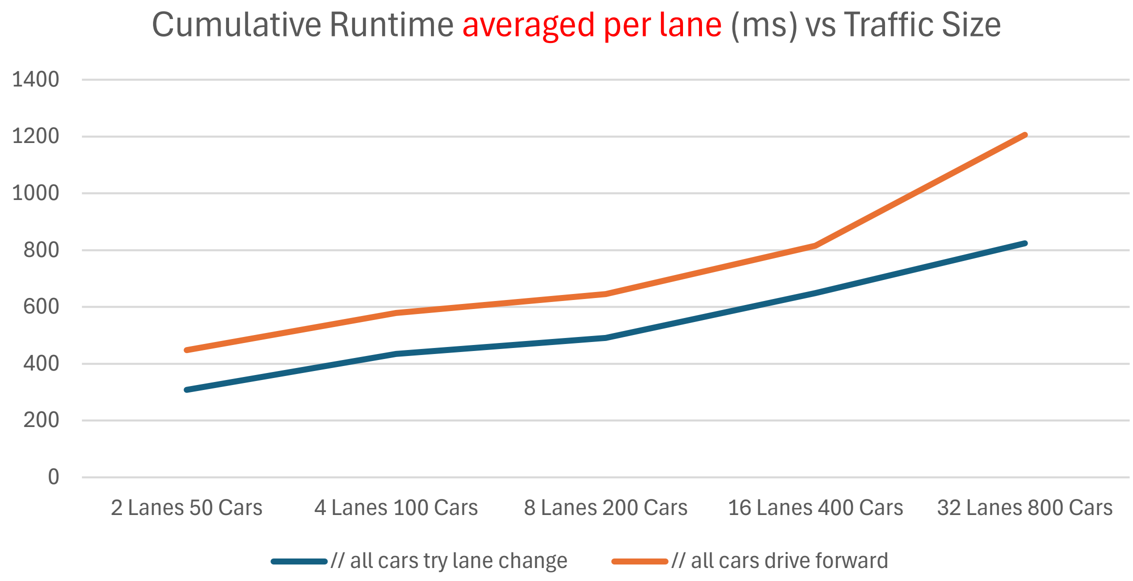
#Lanes	#Cars	Cumulative runtime (us) // all cars try lane change	Cumulative runtime (us) // all cars drive forward
2 Lanes	50 Cars	616116	279713
4 Lanes	100 Cars	1739478	574062
8 Lanes	200 Cars	3928817	1234818
16 Lanes	400 Cars	10379542	2672994
32 Lanes	800 Cars	26368452	12223719



Scale up the traffic size: Runtime increases accordingly.
… but not linearly!

Benchmarking – V2 –O0

#Lanes	#Cars	Cumulative runtime averaged per lane (us) // all cars try lane change	Cumulative runtime averaged per lane (us) // all cars drive forward
2 Lanes	50 Cars	308058	139857
4 Lanes	100 Cars	434870	143516
8 Lanes	200 Cars	491102	154352
16 Lanes	400 Cars	648721	167062
32 Lanes	800 Cars	824014	381991



Scale up the traffic size: Runtime increases accordingly.
… but not linearly!
… increase in cars interaction? OR decrease in performance?

Benchmarking – V2 –O0

#Lanes	#Cars	Cumulative runtime averaged per lane (us) // all cars try lane change	# successful lane changes	Cumulative runtime averaged per lane (us) // all cars drive forward
2 Lanes	50 Cars	308058	709125	139857
4 Lanes	100 Cars	434870	1490086	143516
8 Lanes	200 Cars	491102	2926277	154352
16 Lanes	400 Cars	648721	4939473	167062
32 Lanes	800 Cars	824014	4635999	381991

increase in traffic

decrease in performance

decrease in performance

Scale up the traffic size: Runtime increases accordingly.
… but not linearly!
… It’s both the increase in traffic (#lane changes) and decrease in performance!

Reflections – What I’ve learnt

- Failure in V1 implementation
  - Write readable codes, then think of optimizing!

Reflections – What I’ve learnt

- **Ensure accurate benchmarking!**
  - To increase traffic size: Must increase #Lanes and #Cars at the same time
  - Model performance can be affected by initial conditions!
- Increasing traffic size: Runtime (per lane) increases, two factors:
  - Increase in traffic interaction (lane changes)
  - AND
  - Decrease in performance… to be addressed in parallel implementation

Benchmarking – 2 Lanes, 20 Cars

Version	Cumulative runtime (us) // all cars try lane change	Cumulative runtime (us) // all cars drive forward
v2 (-O0)	10033	2749
v2.1 (-O1)	3138	959
v2.2 (-O2)	2743	883
v3 (-O0)	21242	3390
v3.1 (-O1)	3047	1514
v3.2 (-O2)	2742	1270

All benchmarking are simulated for 100000 steps.

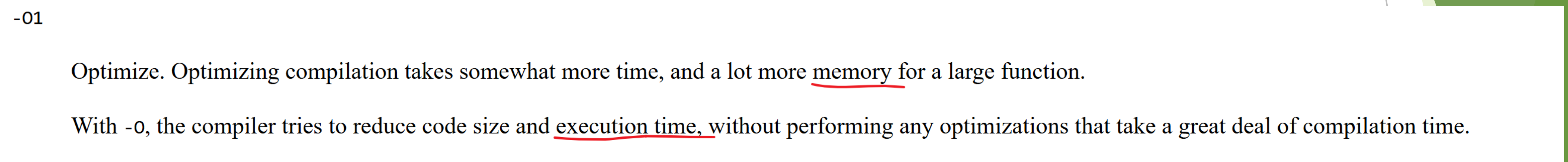
Benchmarking – 2 Lanes, 20 Cars

Version	Cumulative runtime (us) // all cars try lane change	Cumulative runtime (us) // all cars drive forward
v2 (-O0)	10033	2749
v2.1 (-O1)	3138	959
v2.2 (-O2)	2743	883
v3 (-O0)	21242	3390
v3.1 (-O1)	3047	1514
v3.2 (-O2)	2742	1270

–O0 -> -O1: Huge improvement, -O1 -> -O2: Less improvement

Benchmarking – Compiler Optimization*

- -O0 -> -O1: Huge improvement (memory usage optimized)

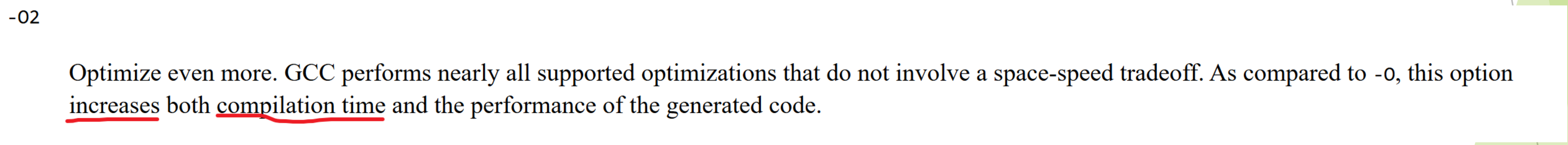


-O1

Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function.

With -O, the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.

- -O1 -> -O2: Less improvement in runtime, more in build time



-O2

Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. As compared to -O, this option increases both compilation time and the performance of the generated code.

* https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Benchmarking – 2 Lanes, 20 Cars

Version	Cumulative runtime (us) // all cars try lane change	Cumulative runtime (us) // all cars drive forward
v2	10033	2749
v2.1	3138	959
v2.2	2743	883
v3	21242	3390
v3.1	3047	1514
v3.2	2742	1270

v2 -> v3: Even makes it worse (optimization won’t save it)

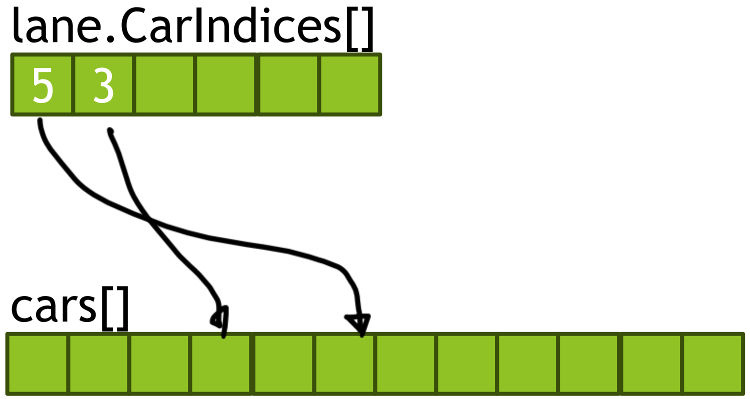
Benchmarking – 16 Lanes, 400 Cars

Version	Cumulative runtime (us) // all cars try lane change	Cumulative runtime (us) // all cars drive forward
v2	9776288	2872727
v2.2	3730578	802068
v3	11647956	3909340
v3.2	4594167	1156458

v2 -> v3: with more cars, still worsens performance (verified!)

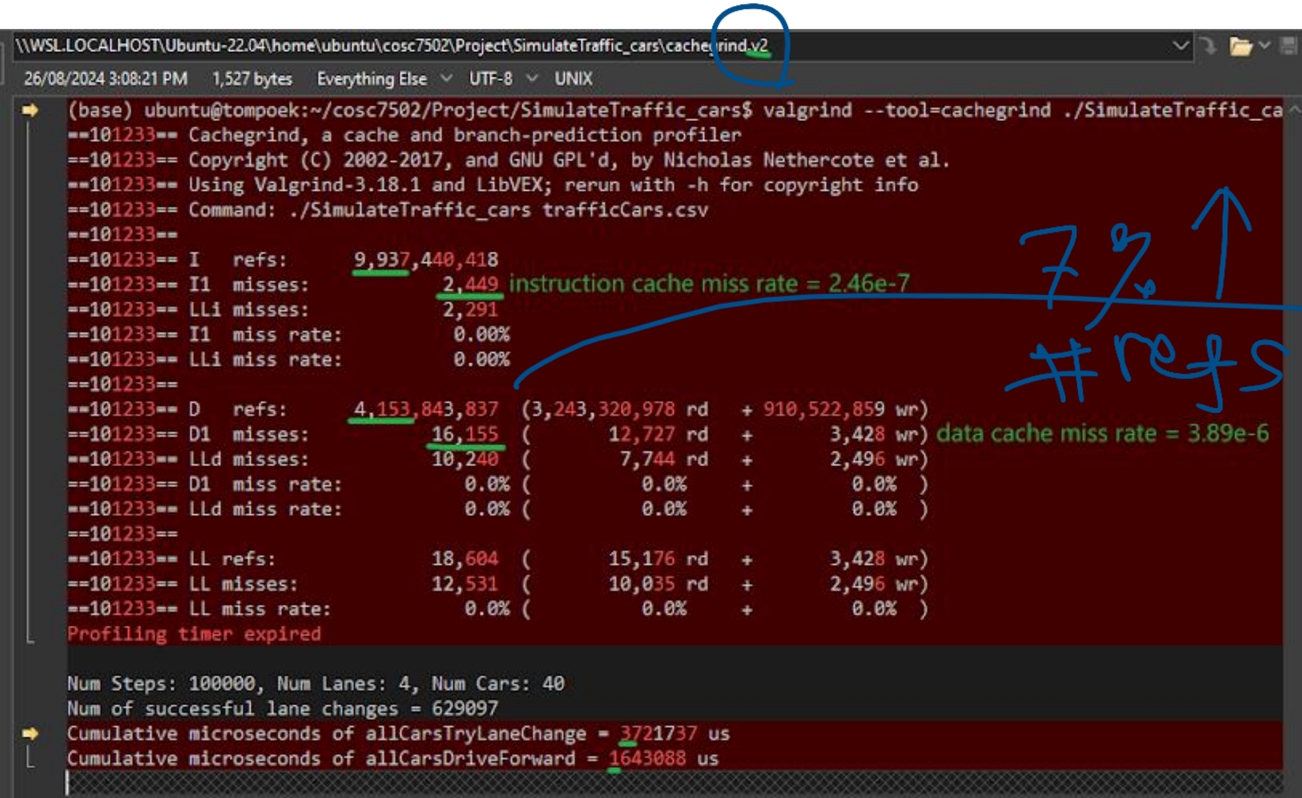
Benchmarking – v2 vs v3

- v2: Traffic = Lanes[], Lane = Cars[], Car = struct{Position, Speed}
  - Accessing cars: “`lane.Cars[j].Position`”
- v3: Traffic = Cars[], Lane = CarsIndices[], Car = struct{Position, Speed}
  - More indexing going on: “`cars[lane.CarIndices[j]].Position/Speed`”



Benchmarking – v2 vs v3

- v2: Traffic = Lanes[], Lane = Cars[], Car = struct{Position, Speed}
  - Accessing cars: “`lane.Cars[j].Position`”
- v3: Traffic = Cars[], Lane = CarsIndices[], Car = struct{Position, Speed}
  - More indexing going on: “`cars[lane.CarIndices[j]].Position/Speed`”

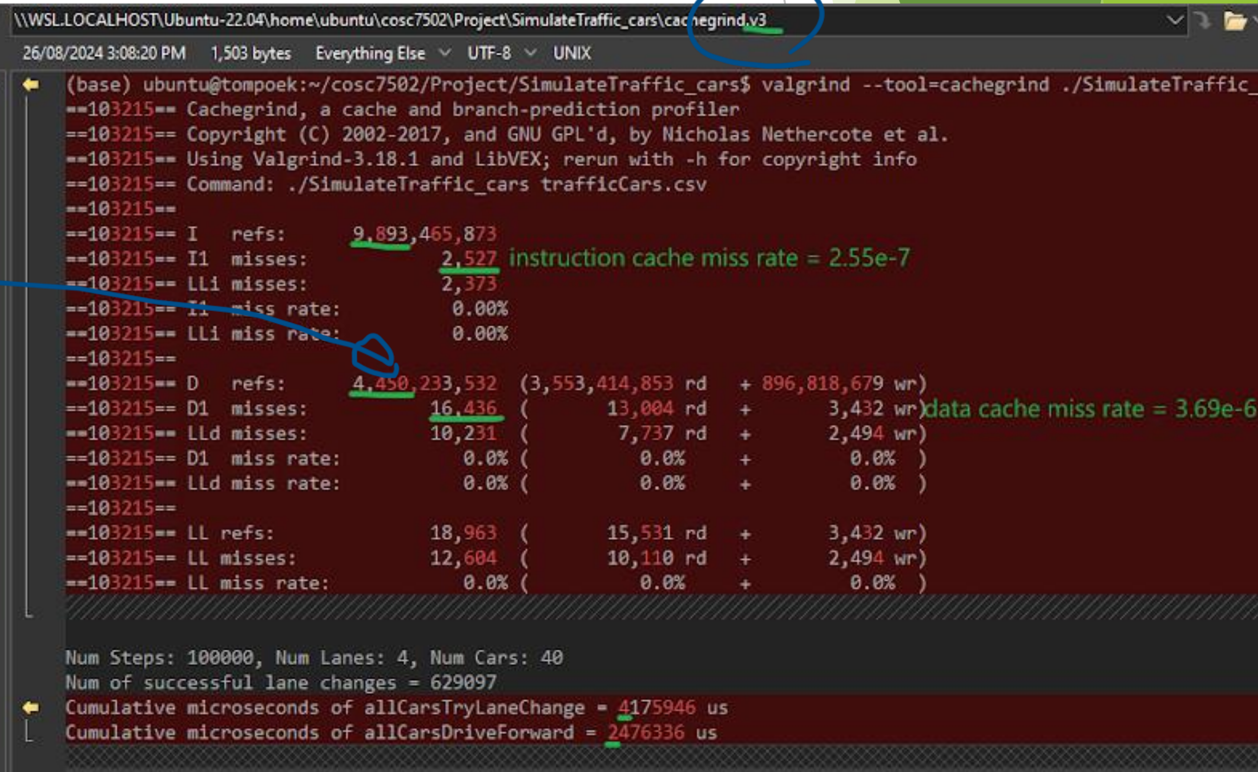


\\WSL.LOCALHOST\Ubuntu-22.04\home\ubuntu\cosc7502\Project\SimulateTraffic_cars\cachegrind.v2
26/08/2024 3:08:21 PM 1,527 bytes Everything Else UTF-8 UNIX

```
(base) ubuntu@tompoek:~/cosc7502/Project/SimulateTraffic_cars$ valgrind --tool=cachegrind ./SimulateTraffic_ca
==101233== Cachegrind, a cache and branch-prediction profiler
==101233== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==101233== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==101233== Command: ./SimulateTraffic_cars trafficCars.csv
==101233==
==101233== I   refs:      9,937,440,418
==101233== I1  misses:            2,449 instruction cache miss rate = 2.46e-7
==101233== LLi misses:            2,291
==101233== I1  miss rate:          0.00%
==101233== LLi miss rate:          0.00%
==101233==
==101233== D   refs:      4,153,843,837  (3,243,320,978 rd   + 910,522,859 wr)
==101233== D1  misses:           16,155  (       12,727 rd   +       3,428 wr) data cache miss rate = 3.89e-6
==101233== LLd misses:           10,240  (        7,744 rd   +       2,496 wr)
==101233== D1  miss rate:           0.0% (          0.0%     +         0.0%  )
==101233== LLd miss rate:           0.0% (          0.0%     +         0.0%  )
==101233==
==101233== LL refs:              18,604  (       15,176 rd   +       3,428 wr)
==101233== LL misses:            12,531  (       10,035 rd   +       2,496 wr)
==101233== LL miss rate:            0.0% (          0.0%     +         0.0%  )
Profiling timer expired

Num Steps: 100000, Num Lanes: 4, Num Cars: 40
Num of successful lane changes = 629097
Cumulative microseconds of allCarsTryLaneChange = 3721737 us
Cumulative microseconds of allCarsDriveForward = 1643088 us
```

7% ↑
#refs



\\WSL.LOCALHOST\Ubuntu-22.04\home\ubuntu\cosc7502\Project\SimulateTraffic_cars\cachegrind.v3
26/08/2024 3:08:20 PM 1,503 bytes Everything Else UTF-8 UNIX

```
(base) ubuntu@tompoek:~/cosc7502/Project/SimulateTraffic_cars$ valgrind --tool=cachegrind ./SimulateTraffic_
==103215== Cachegrind, a cache and branch-prediction profiler
==103215== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==103215== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==103215== Command: ./SimulateTraffic_cars trafficCars.csv
==103215==
==103215== I   refs:      9,893,465,873
==103215== I1  misses:            2,527 instruction cache miss rate = 2.55e-7
==103215== LLi misses:            2,373
==103215== I1  miss rate:          0.00%
==103215== LLi miss rate:          0.00%
==103215==
==103215== D   refs:      4,450,233,532  (3,553,414,853 rd   + 896,818,679 wr)
==103215== D1  misses:           16,436  (       13,004 rd   +       3,432 wr)data cache miss rate = 3.69e-6
==103215== LLd misses:           10,231  (        7,737 rd   +       2,494 wr)
==103215== D1  miss rate:           0.0% (          0.0%     +         0.0%  )
==103215== LLd miss rate:           0.0% (          0.0%     +         0.0%  )
==103215==
==103215== LL refs:              18,963  (       15,531 rd   +       3,432 wr)
==103215== LL misses:            12,604  (       10,110 rd   +       2,494 wr)
==103215== LL miss rate:            0.0% (          0.0%     +         0.0%  )

Num Steps: 100000, Num Lanes: 4, Num Cars: 40
Num of successful lane changes = 629097
Cumulative microseconds of allCarsTryLaneChange = 4175946 us
Cumulative microseconds of allCarsDriveForward = 2476336 us
```

Reflections – What I’ve learnt

- More #cache referencing in v3
- Pay attention when parallelized (next milestone)