ADAS, ML, CV, FPGA, and Healthcare

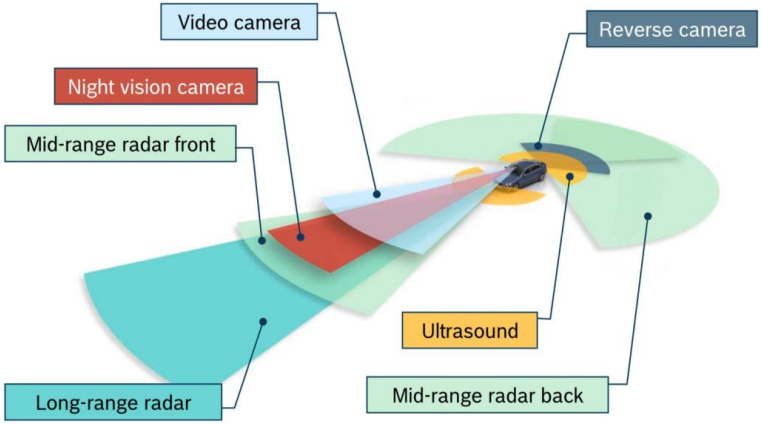
Hersh Godse

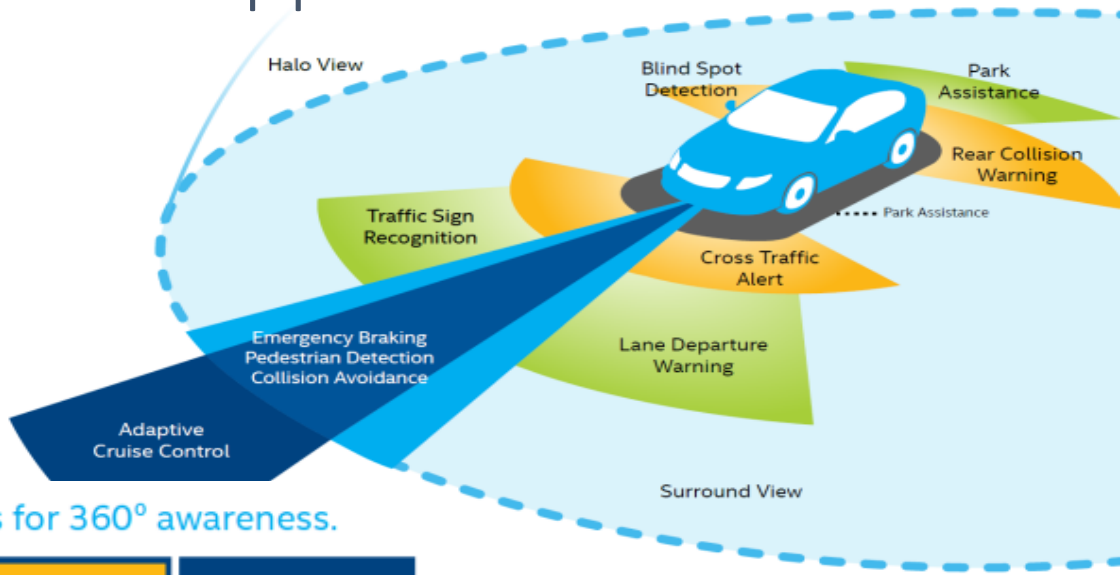
Automated Driver Assistance Systems (ADAS)

Overview:

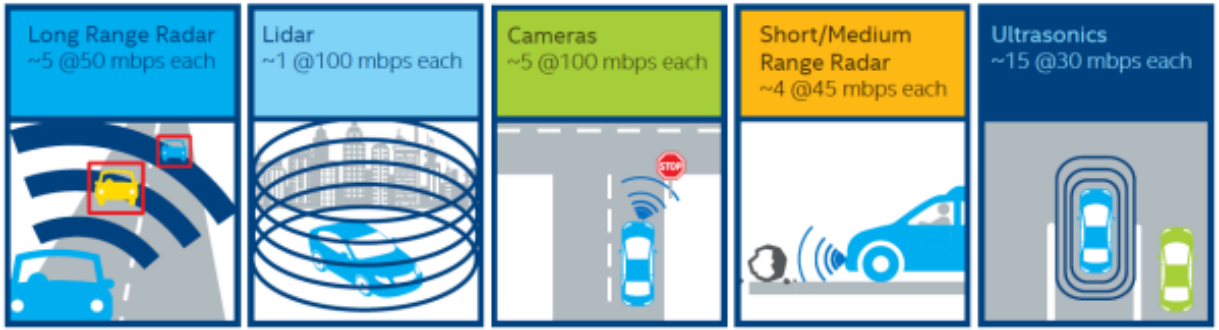
- ADAS is one of the fastest growing segments in automotive field and in general compute area
- Autonomous Driving is an exciting, new, very computationally intense application
- The Autonomous Driving workload is very dynamic; intense learning and trial and error
- ADAS systems have 3 main components: sense, think, and act
- Progressing levels of automation for ADAS systems, from level 0 to level 5

Automotive Sensors and Applications





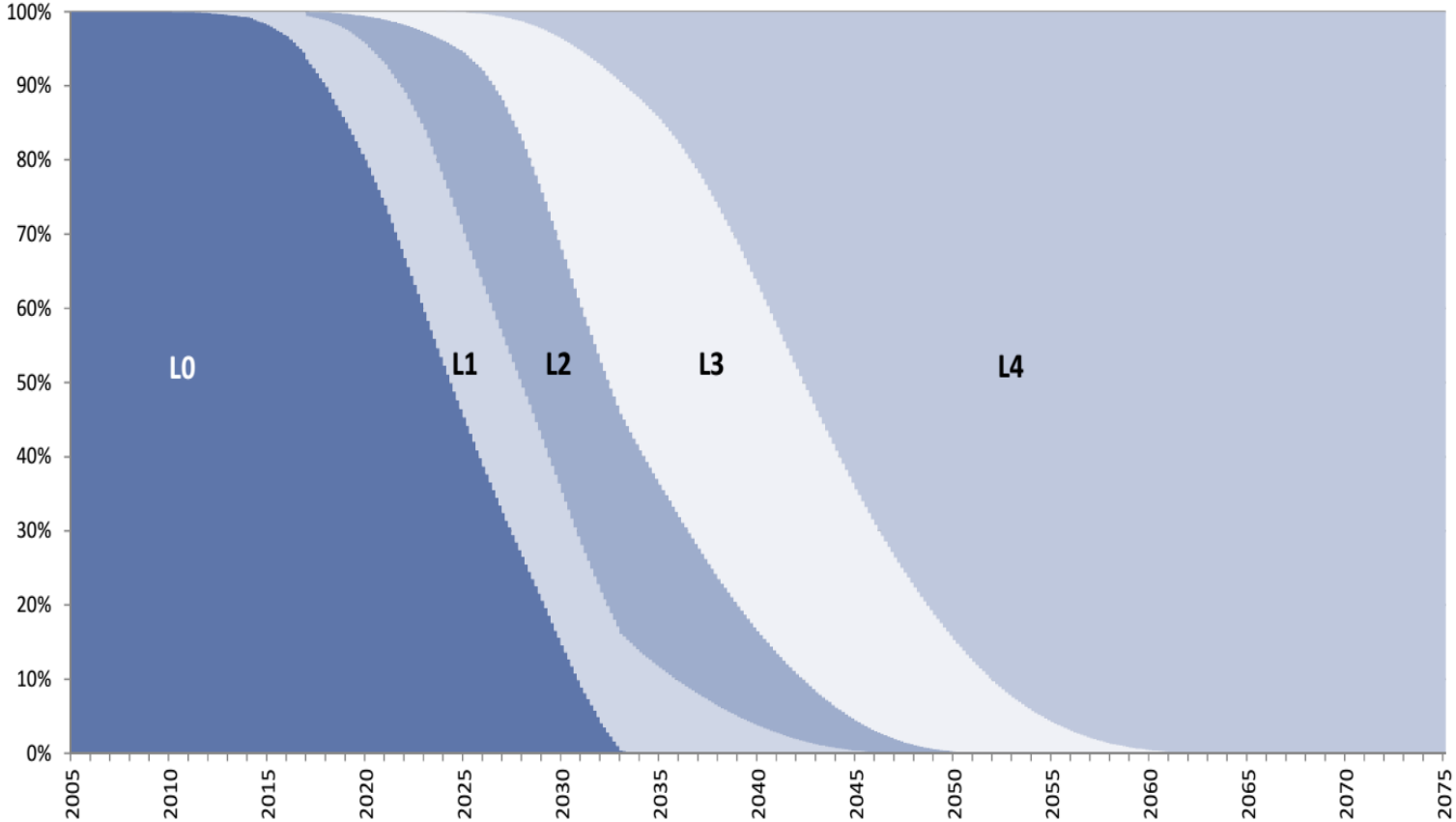
Cars will sense and connect with many things for 360° awareness.



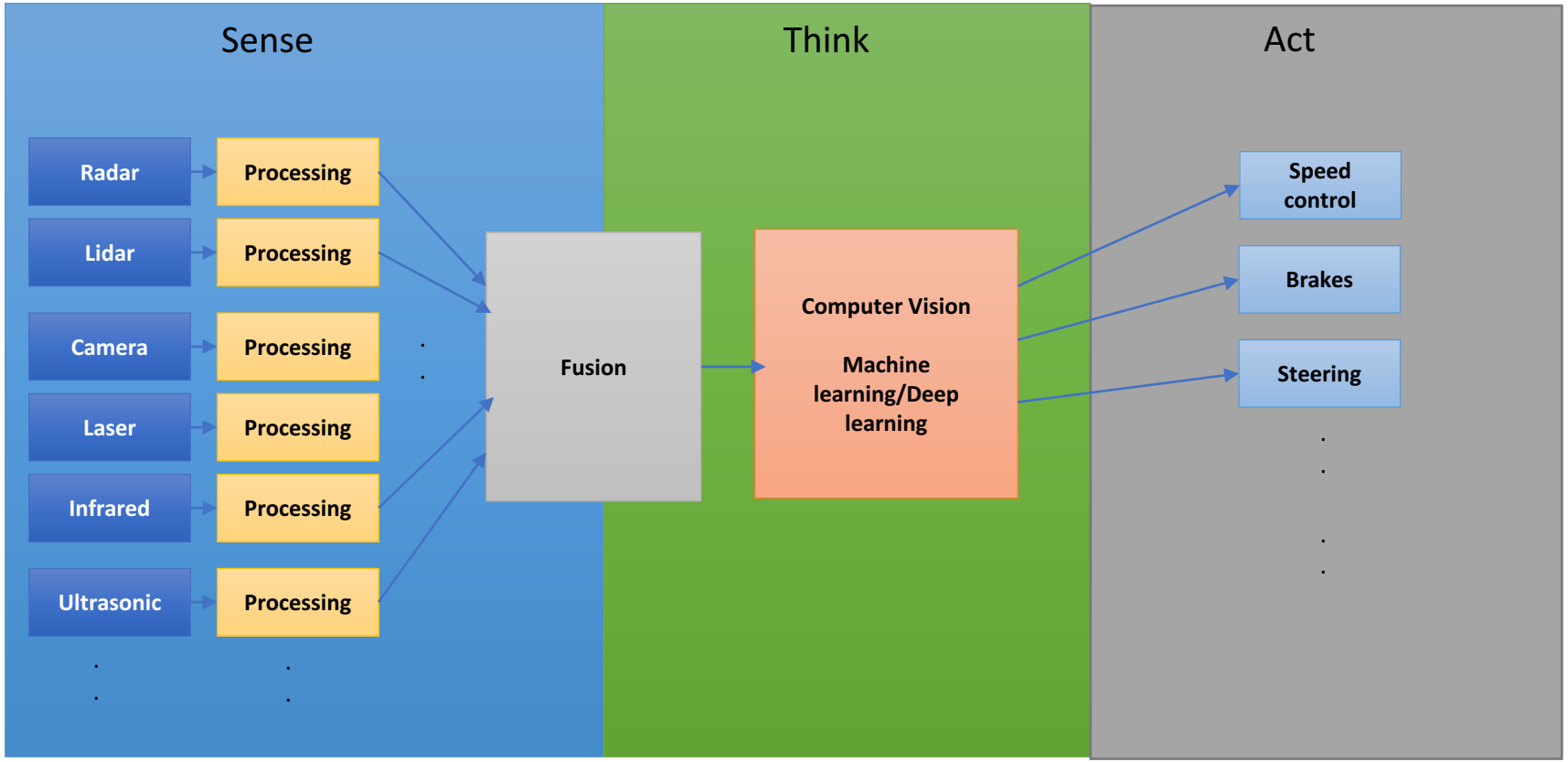
Autonomy level		Driver attentiveness/road monitoring	Comment	Example	
Level 0	**No-Automation**	Driver in **complete and sole control**	Could contain driver support systems, but only warnings; driver never cedes control	Blind Spot Warning	
Level 1	**Function-specific Automation**	Driver maintains overall control, but **can cede limited authority**	One or more specific control functions that operate independent from each other	Adaptive Cruise Control	**ADAS**
Level 2	**Combined Function Automation**	Driver responsible for monitoring the roadway and **available for control at all times on short notice**	At least two primary control functions designed to **work in unison** to relieve driver control	Adaptive Cruise Control with Lane Centering	
Level 3	**Limited Self-Driving Automation**	Driver enabled to cede full control under certain traffic conditions, but is **available for occasional control with a comfortable transition time**	Vehicle designed to ensure safe operation during automated driving mode, **but can determine when the system is no longer able to support automation,** i.e., an oncoming construction area	Autonomous Driving Supporting Multitasking with Transition Time Back to Driver When Necessary	**Autonomous Driving**
Level 4	**Full Self-Driving Automation**	Driver to provide navigation input, but is **not expected to be available for control at any time during the trip**	**Vehicle designed to perform all safety-critical driving functions** and monitor roadway conditions for an entire trip.	Full Autonomous Driving in Any Situation	

Level 5 – no driver

Exhibit 13: The vehicle stock would take much longer to change over, with a full conversion of the US fleet to AV not likely until 2060
North America vehicles in operation forecast by vehicle autonomy level



ADAS Components



Studying Existing ADAS Software Stacks

- LIDAR Vehicle Odometry
  - Calculating position and velocity of vehicle from LIDAR + camera sensor data
- DeepTraffic
  - Trained neural net model determining when to change lanes on highways based on presence of nearby vehicles (LIDAR/radar/ultrasonics)
- Autoware
  - Comprehensive stack with localization, detection, and path planning features
  - Team already analyzed performance of three use cases: traffic light recognition, Single Shot Detection (SSD) object detection, and Fast-RCNN object detection
- Lane Departure Warning System
  - Detect the lane that a vehicle is on, and understand when lane change occurs

Autoware Introduction

- Autoware is an integrated open-source software running on ROS framework
  - Developed by Prof. Shinpei Kato, Nagoya University in Japan; maintained by Tier IV
  - For urban autonomous driving research and development
- Robot Operating System (ROS):
  Middleware framework simplifies communication between various components
  - Catkin build system
  - OpenCV for image processing, computer vision, etc …
  - ROSBAG for time-synched sensor data collection, recording, and playback
  - Rviz to visualize data and software state
  - RQT to visualize communication between ROS *nodes* via ROS message *topics*
- Autoware also shows metrics such as CPU utilization for perf analysis.

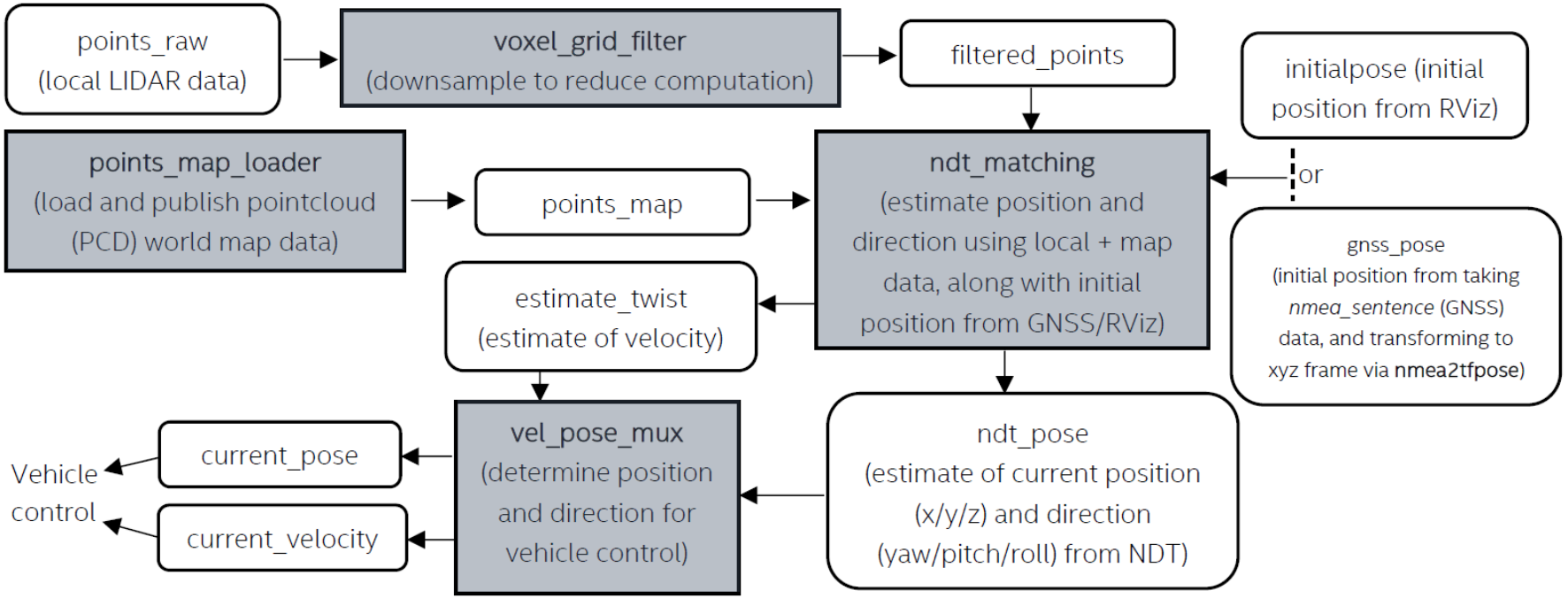
Autoware Features

- 3D Mapping and Localization
- Detection
  - Object/car/pedestrian detection and tracking
  - Traffic signal detection
  - Lane detection
- Path planning and following
- Sensor calibration and fusion
- Data logging
- Steering/brake/acceleration control

Autoware - Mapping and Localization

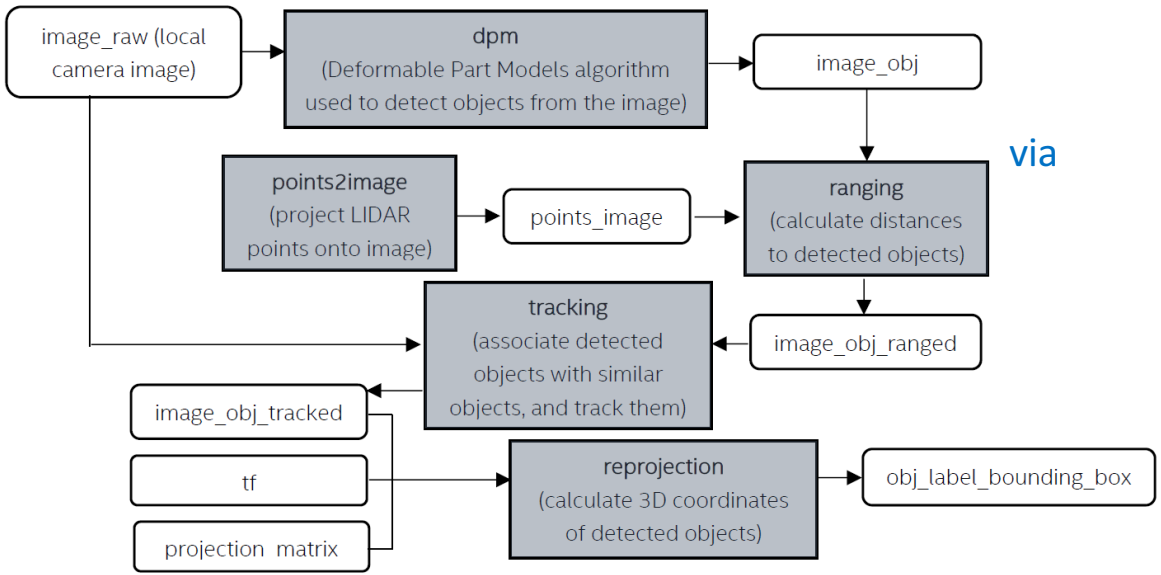
- HD LIDAR 3D world map is preloaded
  - Map is in Point Cloud Data (PCD) format
  - Provided by 3rd party vendor Velodyne
  - Has info on roads, objects around roads, traffic signal/crosswalk locations, etc …
- GNSS (Global Navigation Satellite System) provides localization starting point
  - However, GNSS data has localization error of 30m – much too large
- Localization by cross-matching local LIDAR scan sensor input to the world map
  - Done using Normal Distributions Tracking (NDT) algorithm
    - Determines best fit transform between two large point clouds (world and local maps)
  - As vehicle moves and has more points to compare, localization gets more stable
- Once localization process stabilizes, error of localization is 10cm

Autoware - Mapping and Localization



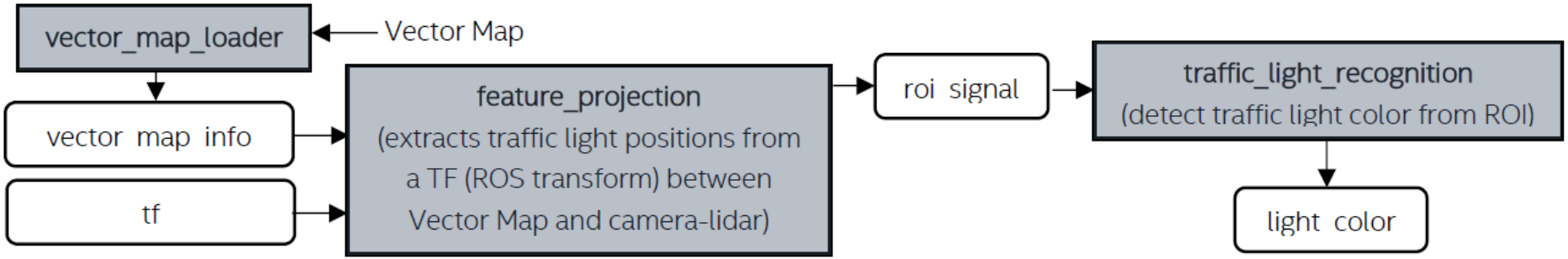
Autoware - Object Detection

- Detect vehicles and pedestrians from camera images via DPM (Deformable Part Models) algorithm
  - Autoware also has alternate algorithms for detection, including SSD and Fast-RCNN
- Calculate distance to detected objects via LIDAR point info
- Tracking uses Kalman filters
  - Improve detection accuracy
- Calculate 3D object coordinates reprojection via



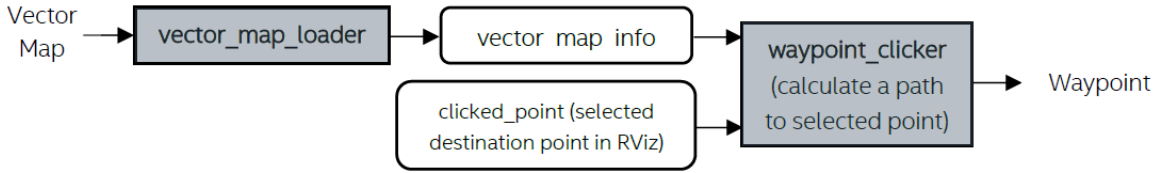
Autoware - Traffic Signal Detection

- Calculate traffic light coordinates from current position + Vector Map info
  - Vector Map is GIS (Global Information System) data representing geometric information of the roads
- Traffic light position ROI is projected onto the camera image via sensor fusion
- Image processing is used to determine the traffic light color in the ROI
  - This is later used in path following (stop/go at intersection)

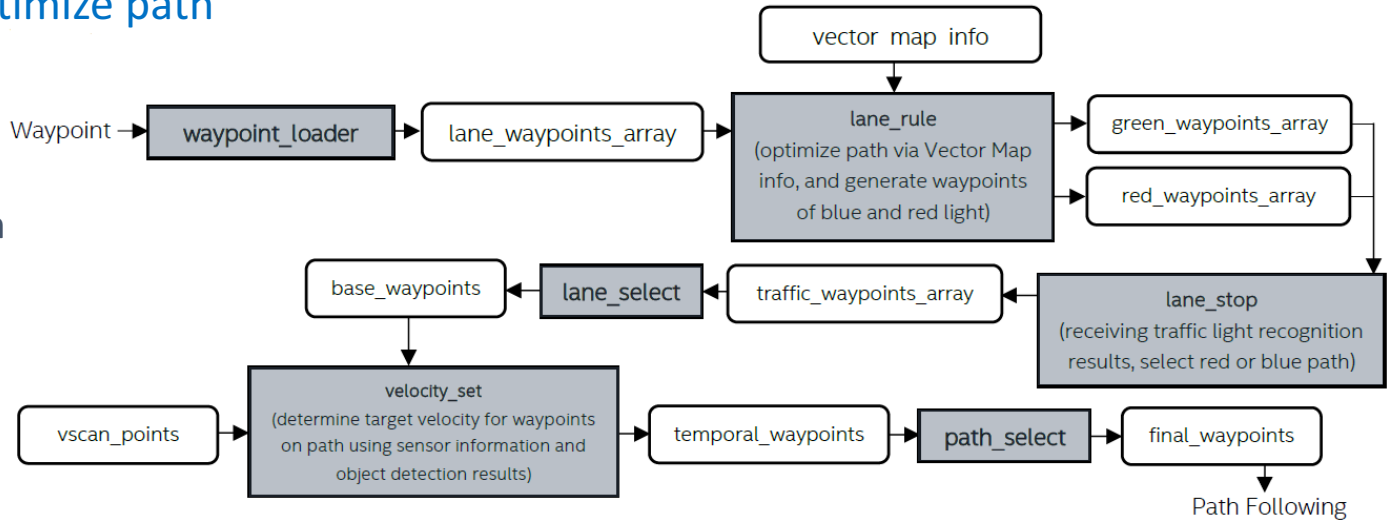


Autoware - Path Planning

- **Path Generation: generate Waypoint file**
  - Data string of “waypoints” containing coordinate, direction, and target velocity info

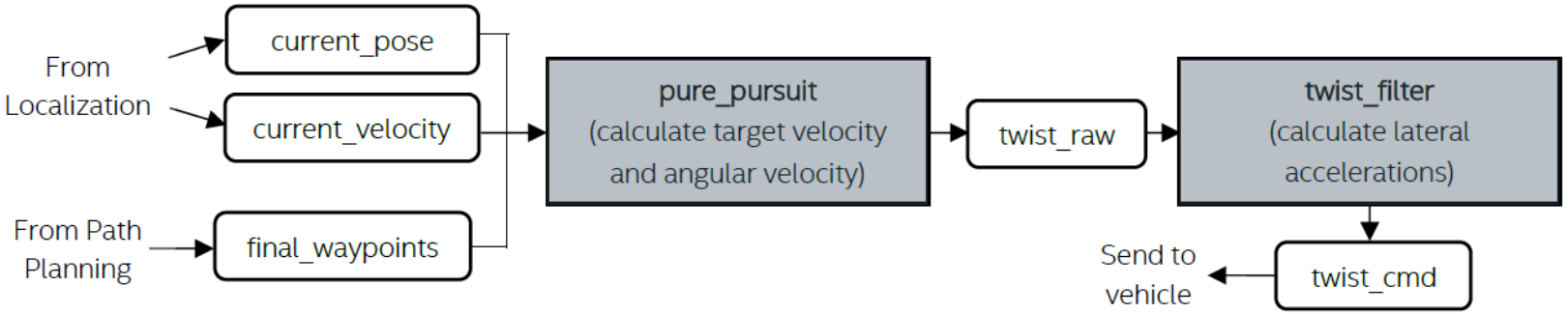


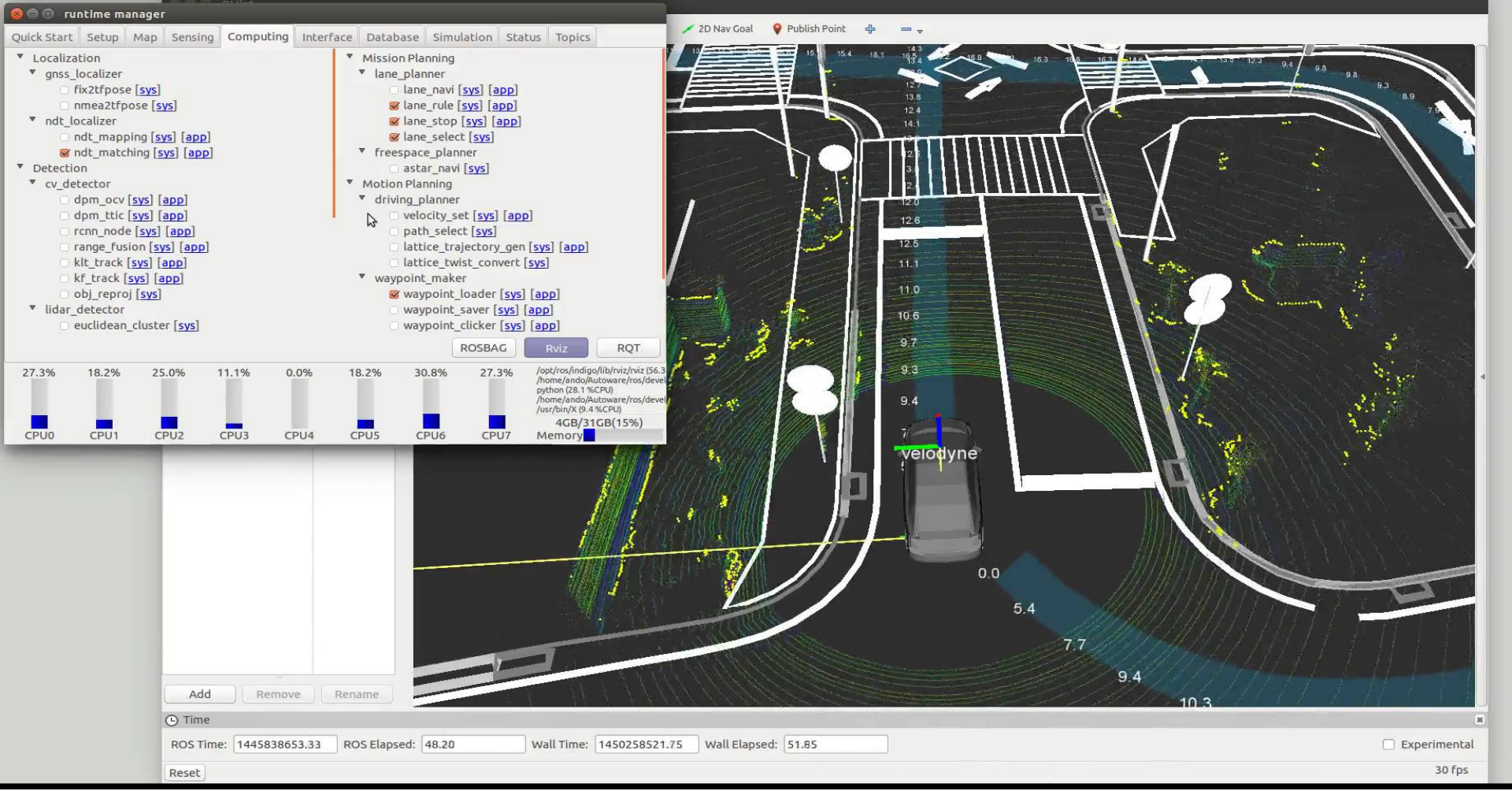
- **Path Planning: optimize path**
  - Lane selection
  - Traffic signal
  - Object detection



Autoware - Path Following

- Path Following happens after path planning, focus just on next waypoint in path
- pure_pursuit node:
  - Calculate curve of circle passing through current position and path target waypoint
  - Compute target and angular velocities from calculated curvature and current velocity
- twist_filter node:
  - Calculate lateral accelerations from target velocity/angular velocity
  - If lateral acceleration is over threshold, slow down target velocity (less vehicle “jerk”)




runtime manager
Quick Start
Setup
Map
Sensing
Computing
Interface
Database
Simulation
Status
Topics
Localization
gnss_localizer
fix2tfpose [sys]
nmea2tfpose [sys]
ndt_localizer
ndt_mapping [sys] [app]
ndt_matching [sys] [app]
Detection
cv_detector
dpm_ocv [sys] [app]
dpm_ttic [sys] [app]
rcnn_node [sys] [app]
range_fusion [sys] [app]
klt_track [sys] [app]
kf_track [sys] [app]
obj_reproj [sys]
lidar_detector
euclidean_cluster [sys]
Mission Planning
lane_planner
lane_navi [sys] [app]
lane_rule [sys] [app]
lane_stop [sys] [app]
lane_select [sys]
freespace_planner
astar_navi [sys]
Motion Planning
driving_planner
velocity_set [sys] [app]
path_select [sys]
lattice_trajectory_gen [sys] [app]
lattice_twist_convert [sys]
waypoint_maker
waypoint_loader [sys] [app]
waypoint_saver [sys] [app]
waypoint_clicker [sys] [app]
ROSBAG
Rviz
RQT
27.3%
18.2%
25.0%
11.1%
0.0%
18.2%
30.8%
27.3%
CPU0
CPU1
CPU2
CPU3
CPU4
CPU5
CPU6
CPU7
4GB/31GB(15%)
Memory
2D Nav Goal
Publish Point
velodyne
0.0
5.4
7.7
9.4
Add
Remove
Rename
Time
ROS Time: 1445838653.33
ROS Elapsed: 48.20
Wall Time: 1450258521.75
Wall Elapsed: 51.85
Experimental
Reset
30 fps

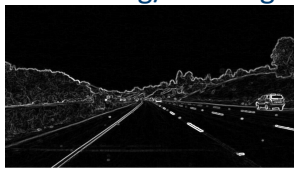
Lane Departure Warning System (LDWS)

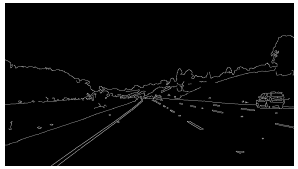
Video Input Stream of Frames

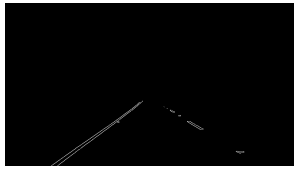
Greyscale

Gaussian Blur

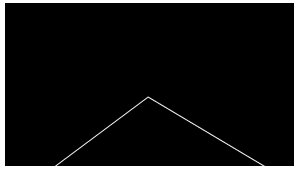
Canny Edge Detection + Filtering/Masking







Hough Line Transform + Overlay



- Uses OpenCV computer vision library
  - ROI – region of interest of video where road is
  - Mode (CPU/OpenCL/Cuda) – decides implementation of OpenCV algorithms used
- Output metric: compute time on each image frame

Performance on Varying Compute Units

Milliseconds of compute time per frame:

(lower is better)

Compute Unit \ Video Load	road-single.avi	road-dual.avi	h-test720.avi
i7-6700 @ 3.40 GHz x 8 CPU	3.32	4.61	8.06
GTX Titan X w/ OpenCL	2.39	3.18	5.21
GTX Titan X w/ Cuda	1.14	1.49	2.95
Partial Arria 10 FPGA + Partial Titan X w/ OpenCL	--	--	--

Note: video loads increase in resolution and thus compute time from left to right

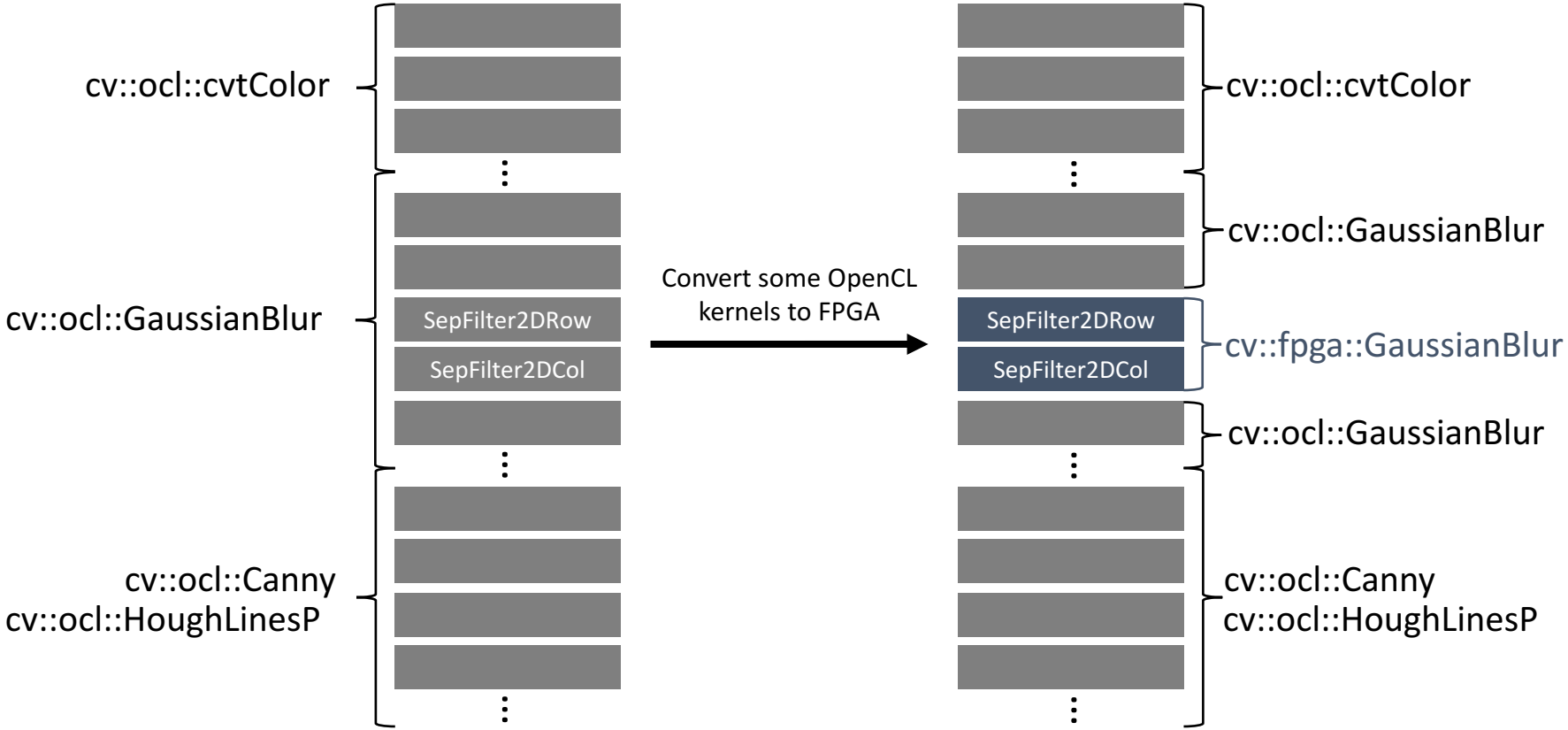
Running LDWS on FPGA Compute Unit

- Intel/Altera FPGA SDK allows for conversion of OpenCL code into FPGA binaries
- Idea: using this, convert the OpenCL mode of LDWS stack to FPGA
  - Problem: LDWS doesn't have explicit OpenCL code. Rather, uses the OpenCL-enabled version of functions in the OpenCV computer vision library
  - As a result, the desired OpenCL code is buried deep inside the OpenCV library
- Rough steps:
  - Extract some OpenCL *kernels* from OpenCV, then compile them on Arria 10 FPGA
  - Modify OpenCV source to properly call these FPGA OpenCL kernels

OpenCL Primer

- OpenCL is used to vectorize and highly parallelize workloads, using devices like GPUs and FPGA accelerators
- With OpenCL, *host* instantiates several threads all running in parallel on the *device*
  - Each thread tackles a different small portion of the workload
- Each thread runs an instance of an *OpenCL kernel*
  - OpenCL kernel: script of instructions that each thread should perform
  - Vector Add kernel example: C[i] = A[i] + B[i];
  - Image processing example: different thread for each pixel of image

OpenCL Kernels in LDWS



LDWS on FPGA – Cont.

- Added print to console debug flags in OpenCV source, to understand call stream
- Extracted SepFilter2DRow and SepFilter2DCol OpenCL kernel source from OpenCV
  - Compiled both onto Arria 10 FPGA via Altera/Intel FPGA SDK and the aoc compiler
- Added Altera OpenCL tools to OpenCV
  - Initially tried creating separate FPGA module that was a duplicate of core module
  - Motivation: since some kernels running on FPGA and others on GPU, concern over conflicts between Intel/Altera and Nvidia OpenCL runtimes
  - This was unnecessary, because installing OpenCL device drivers gives higher-level libOpenCL.so layer (similar across vendors), and lower level ICD layer (installable client driver, specific to vendor)
  - Thus, irrelevant which vendor's libOpenCL.so file is linked to; at the ICD layer, OpenCL will use vendor driver corresponding to device selected
  - By correctly detecting and choosing the proper OpenCL device/context, the proper type of OpenCL will be used. This enables using multiple cross-platform OpenCL devices.

LDWS on FPGA – Cont.

- Modified Cmake files of OpenCV core module to link to some Altera CL dependencies
- Modified OpenCV source code to run kernel on FPGA for SepFilter2DRow/Col
  - in OpenCL call stream, when constructing the ocl::Program object, call createProgramWithSource (for GPU kernels) OR createProgramFromBinary (for FPGA kernels)
- At beginning of code, when OpenCV creates its OpenCL context, changed the OPENCV_OPENCL_DEVICE environment variable to ":ACCELERATOR:" so FPGA detected
- In core module, added #define CV_OPENCL_DATA_PTR_ALIGNMENT 64
  - Necessary because Altera FPGAs only support 64 bit alignment
- Results: the SepFilter2DRow/Col kernels successfully ran on the FPGA, but OpenCV unable to run remaining kernels on the GPU; trying to run GPU kernels on the FPGA. Was unable to overcome this issue in internship time.
- FPGA device kept locking up; had to cold reboot and reprogram firmware to fix

Philips Healthcare ImageWarp Optimization

- Work with Philips Healthcare, optimizing CT-scan imaging solution
  - Effort to convince move from Nvidia/Cuda based stack to IA based stack
  - Goal: achieve comparable performance for sample workload (ImageWarp kernel) on IA
- Sample workload performance of 42ms on FPGAs (PSG team)
- Nvidia GTX Titan X GPU with Cuda kernel had performance of 36ms
  - Improved vectorization, reducing this to 15ms
- Xeon server CPU kernel without any optimization had 8599ms run
  - Helped CDS team optimize to 37ms (performance gain of 232x)

Philips Healthcare Kernel Performance

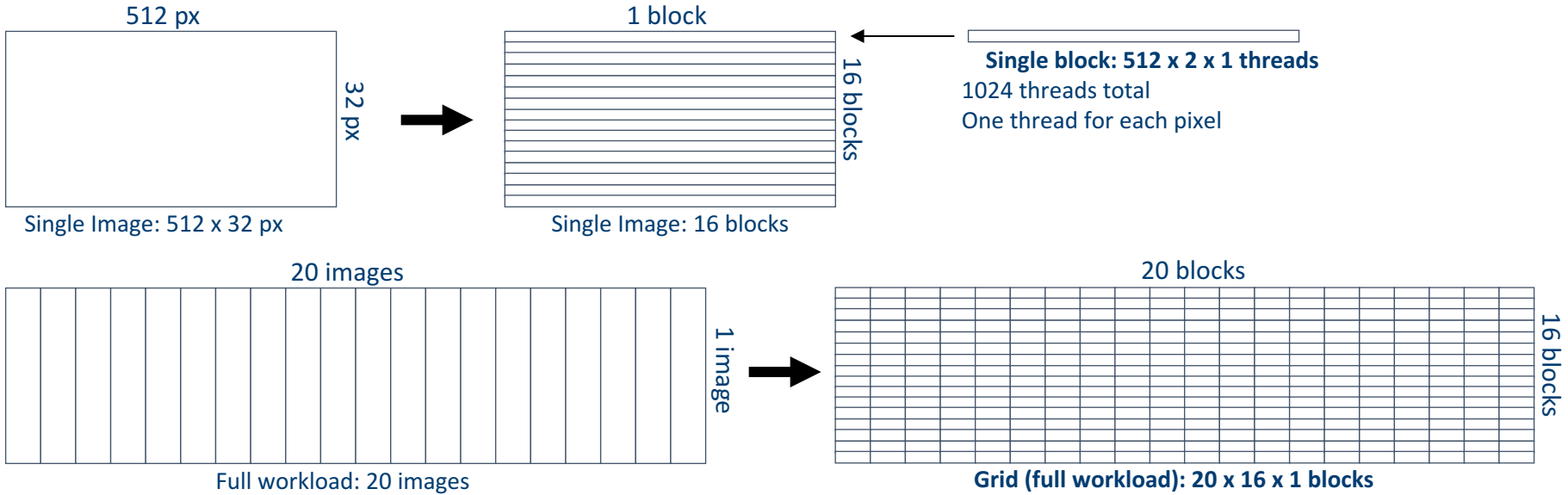
FPGA/OpenCL code (Arria 10)	Cuda code (GTX 580)	Cuda code (GTX Titan X)	Optimized Cuda code (GTX Titan X)	Optimized Cuda code (Quadro M5000)
42ms	52.475ms	36.011ms	15.589ms **(2.6x)**	24.391ms
			Improved vectorization	lower perf: 2048 Cuda cores vs. Titan X's 3048

C/C++ code, Xeon 2S/56C	C/C++ code, Xeon 2S/56C	C/C++ code, Xeon 2S/56C	C/C++ code, Xeon 1S/28C	C/C++ code, Xeon 1S/28C	C/C++ code, Xeon 1S/28C
8599ms	676ms **(13x)**	116ms **(74x)**	98ms **(88x)**	51ms **(169x)**	37ms **(232x)**
CPU kernel as is from Philips	OpenMP parallelization	OpenMP, Intel ICC compiler	OpenMP, ICC, HT On, single CPU socket	OpenMP, ICC, HT On, 1S, copy reduction	Compiled for SKX instruction set

Vectorization of GPU Kernel

- Cuda structure: Workload represented by a **grid**

Grid contains several **blocks**; each block contains (upto) 1024 **threads**



cuLaunchKernel parameters: Grid dims (20x16x1), Block dims (512x32x1), and Cuda Kernel