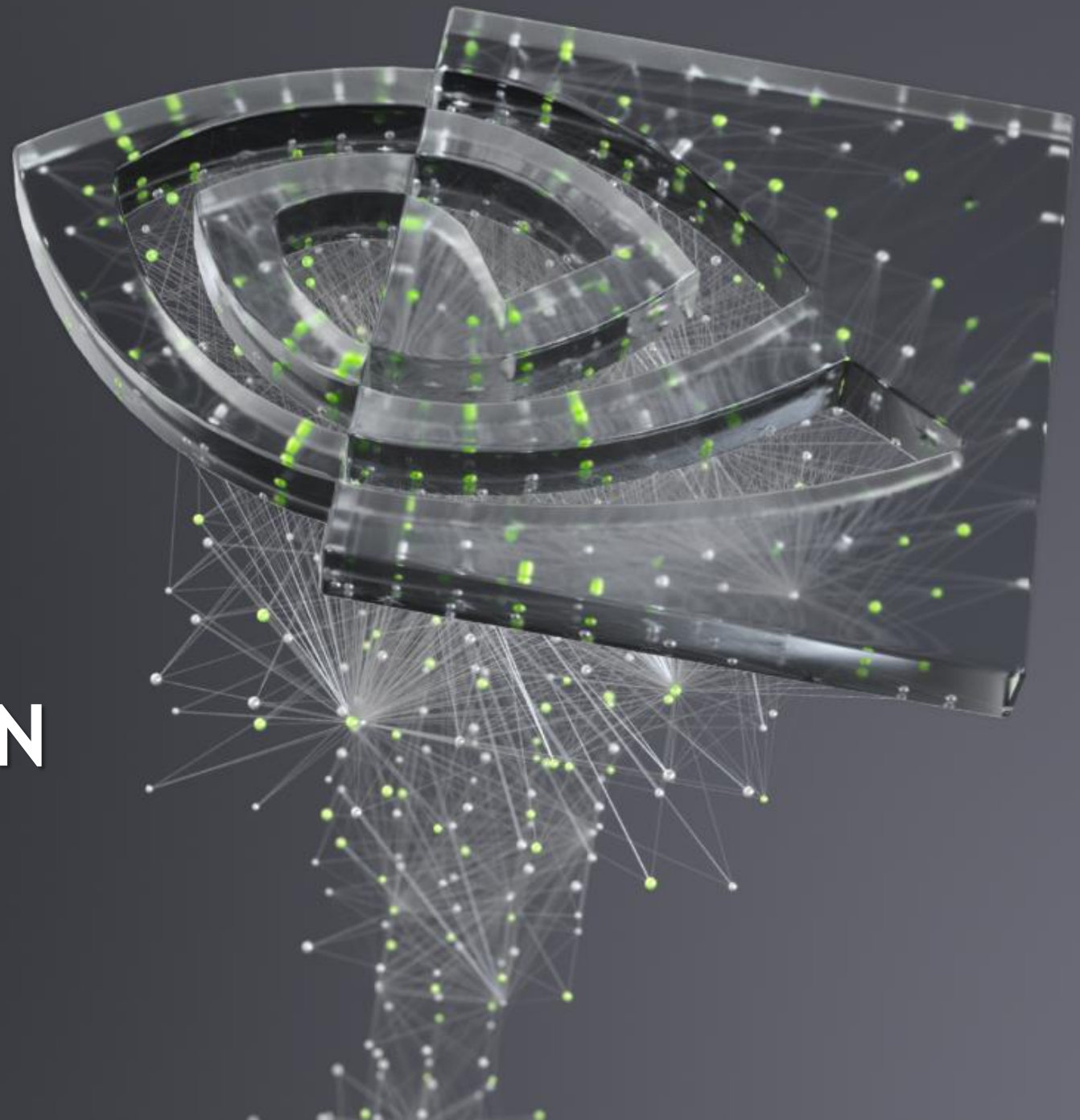


MIGRATING FROM X86 TO JETSON

SA Jeff

AGENDA

Jetson hardware introduce

Migrating analysis for software stack

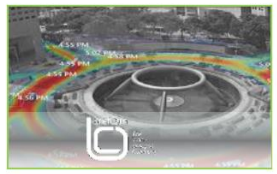
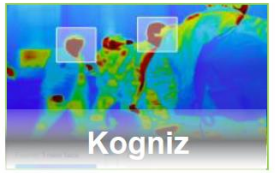
Migrating analysis for DL model

Migrating analysis for CUDA

Perf profiling

JETSON HARDWARE INTRODUCE

EDGE AI MOMENTUM BUILDING ON NVIDIA JETSON

AMR	RETAIL	SMART CITY	AGRICULTURE/ CONSTRUCTION	SERVICES	LOGISTICS	MANUFACTURING / INSPECTION	HEALTHCARE
							
							
							
							

700,000+
Developers

3,000+
Customers

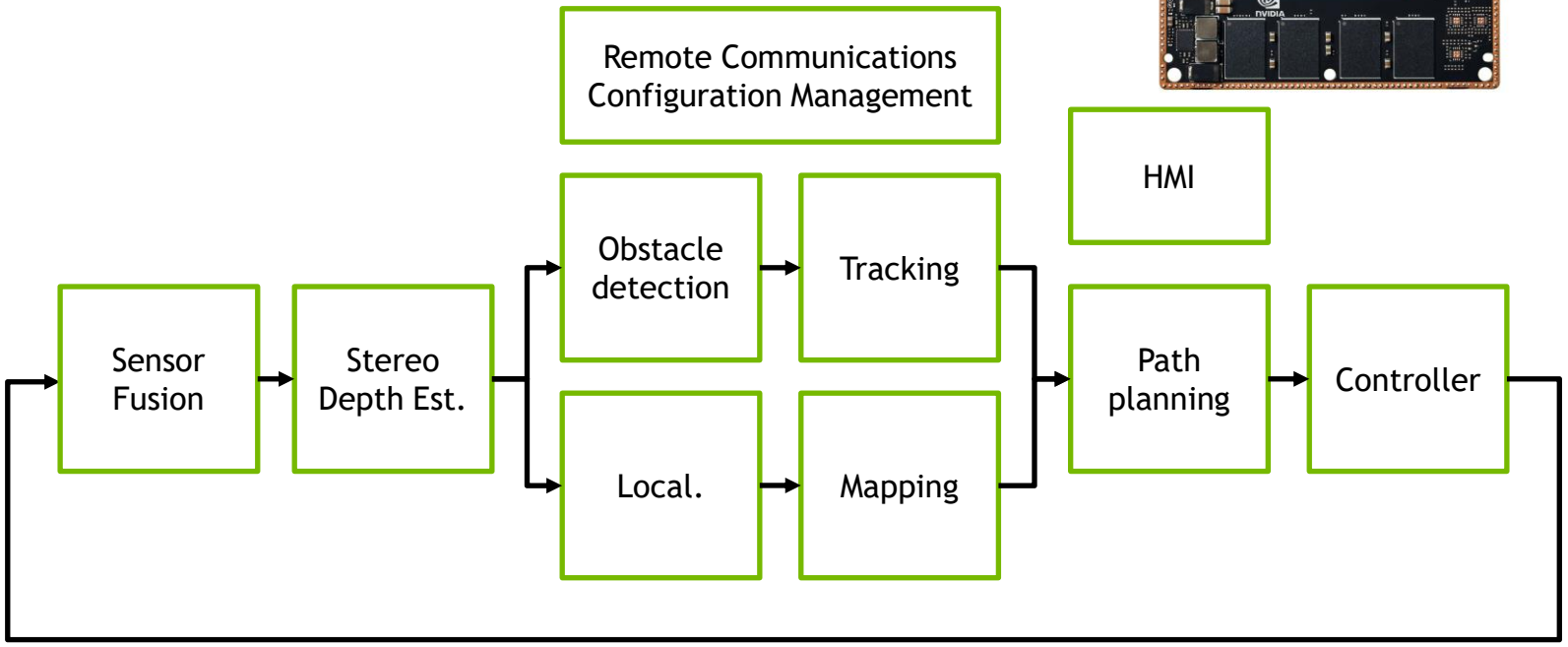
AUTONOMOUS MACHINES PROTOTYPE

Intel Core i7 Skylake + GTX 1070
>200 W
28 TOPS
4000 cm^3
Consumer grade

JETSON AGX XAVIER PLATFORM

1 Jetson AGX Xavier
30 W
32 TOPS
600 cm^3
Commercial grade

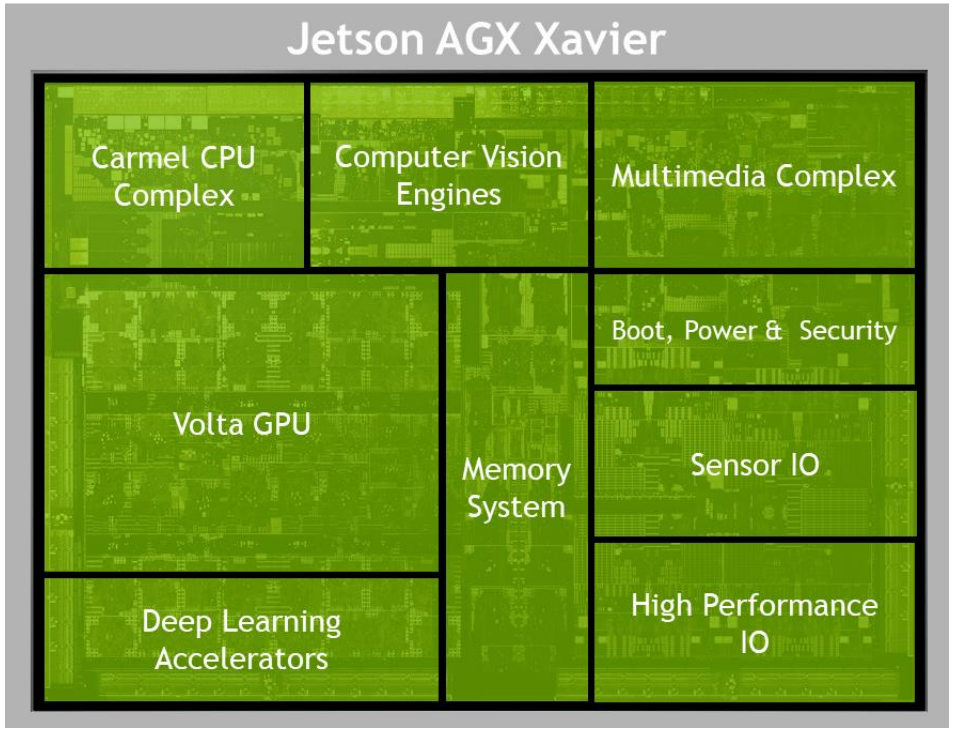
AGX Xavier Dev Kit ($699)
JetPack SDK
DeepStream SDK



JETSON AGX XAVIER

World's First Autonomous Machine Platform

Computer Vision Engines
Vision Accelerator
Stereo & Optical Flow Engine
HDR ISP



Carmel ARM V8.2 CPU
8 Cores 4x2MB L2, 4MB L3
10-wide Superscalar
Cache coherent CPU complex

Volta Tensor Core GPU
FP32/FP16/INT8 Multi-Precision
512 CUDA Cores
6 TFLOPS (FP16)
22.6 Tensor Core DL TOPS

DLA – Designed for Inference
5.7 TFLOPS FP16
11.4 TOPS INT8

Multimedia Engines
Encode, Decode,
Video Image Compositor
H.264, H.265, VP9
HDMI and DP Display support

Boot, Power & Security
Boot and power management processor
TEE + ARM TrustZone
AES, RSA, SHA

Industry Standard IO
Always on Sensor processor engine (AON/SPE)
CAN, DMIC, GPIO, I^2C, I^2S, PMC, SPI, UART

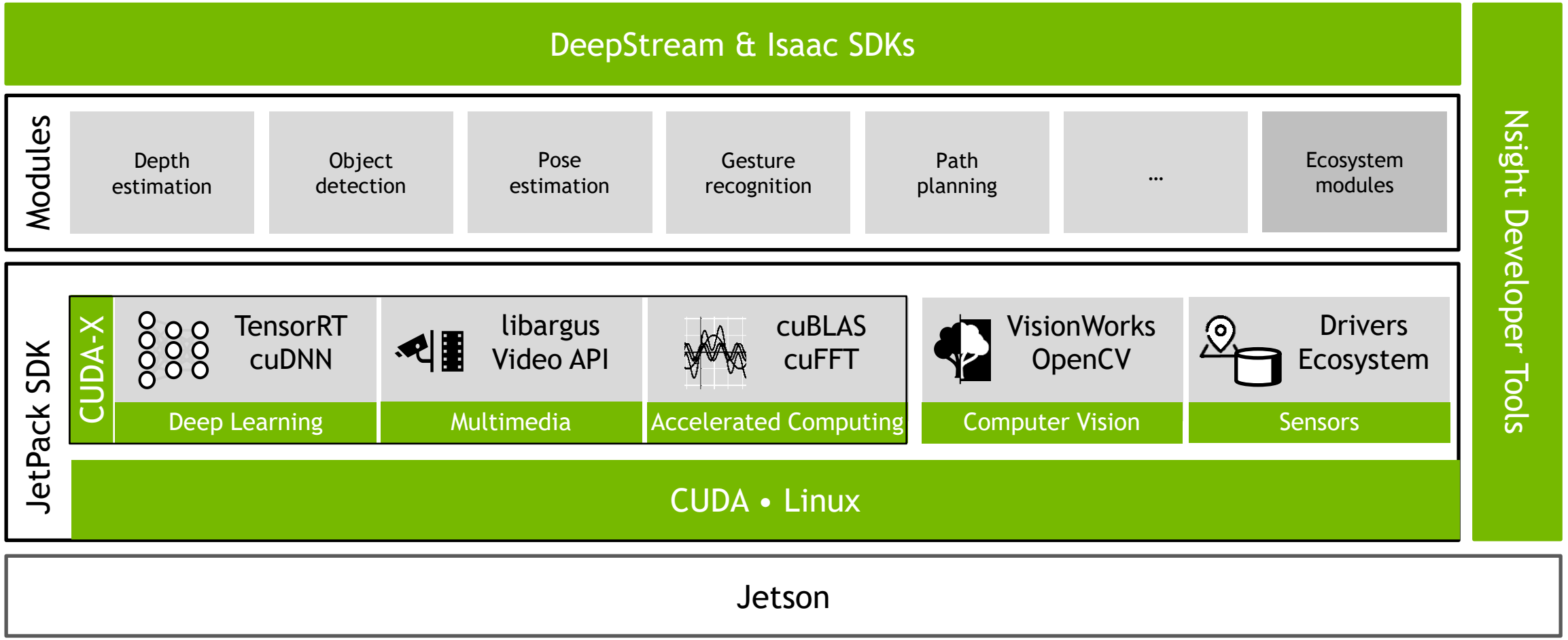
Industry Standard High-Speed IO
PCIe Gen4 Root and Endpoint
16 lanes MIPI CSI-2 | 8 lanes SLVS-EC
Support C-Phy and D-Phy
RGMII Ethernet

USB 3.1 and 2.0
USB 3.1 Gen2 Host and Device

Memory
256-Bit LPDDR4X
32GB, 137 GB/s

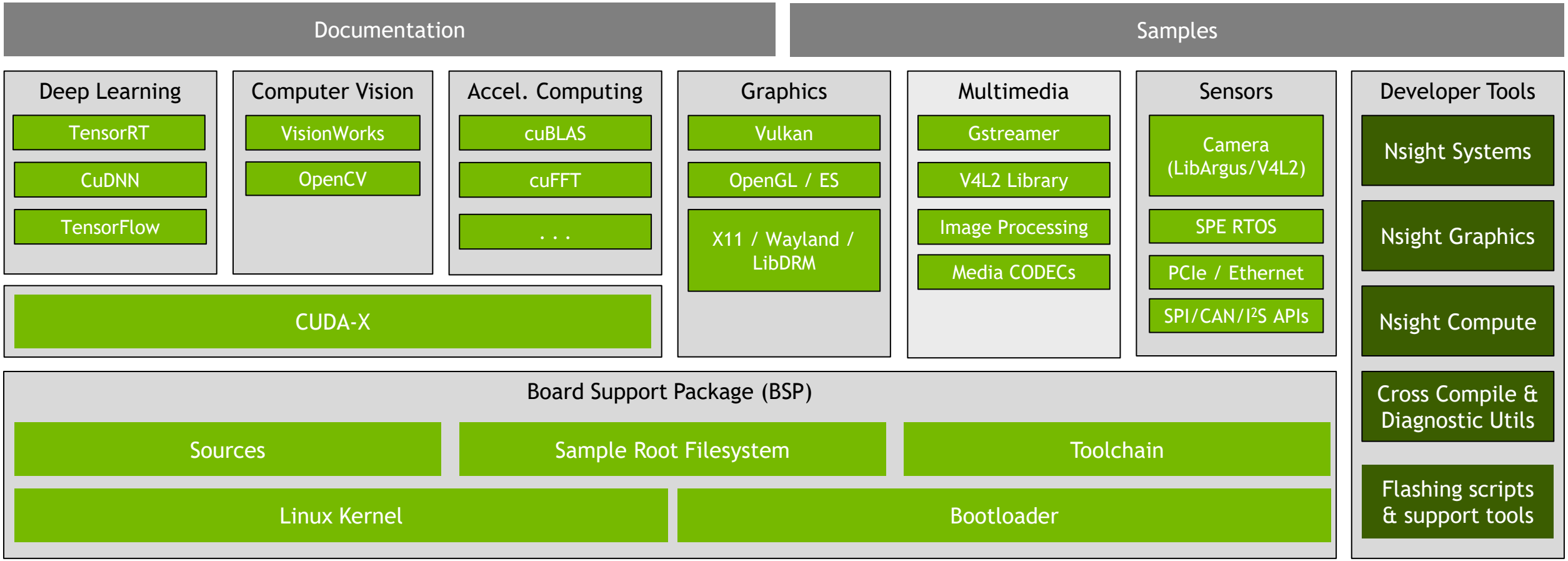
MIGRATING ANALYSIS FOR SOFTWARE STACK

JETSON SOFTWARE



Jetson software: developer.nvidia.com/jetson

JETPACK SDK ARCHITECTURE



JETSON LINUX

NVIDIA® Jetson™ Linux Driver Package (L4T)

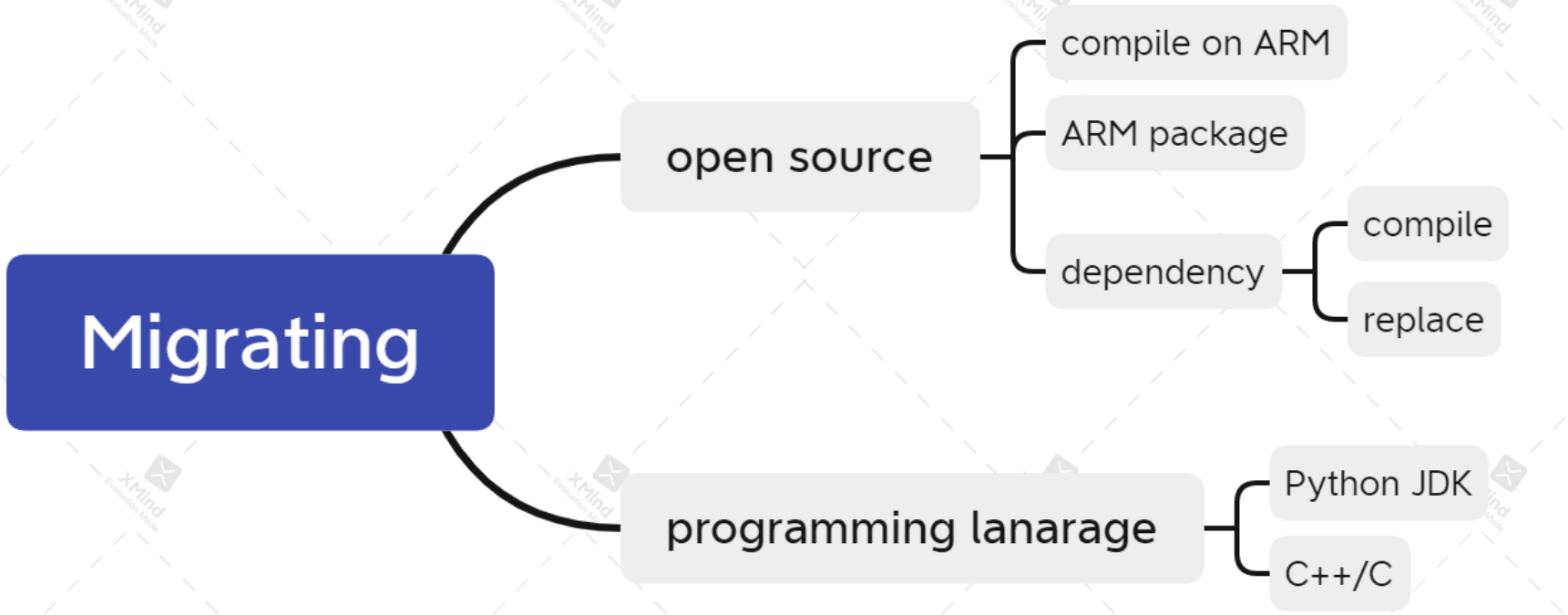
Linux Kernel 4.9

Ubuntu 18.04

	JETSON NANO	JETSON TX2	JETSON XAVIER NX	JETSON AGX XAVIER
GPU	128 Core Maxwell 0.5 TFLOPs (FP16)	256 Core Pascal 1.3 TFLOPS (FP16)	384 Core Volta 21TOPs (INT8)	512 Core Volta 32TOPs (INT8)
CPU	4 core ARM A57	6 core Denver and A57	6 core Carmel ARM v8.2 64-bit CPU	8 core Carmel ARM v8.2 64-bit CPU
CPU Max Freq	1.43 GHz	6 core Denver2&A57: 1.4GHz | 4 core A57: 2 GHz	4/6 core @ 1.4 GHz | 2 core @ 1.9 GHz	8 core @ 2.265 GHz

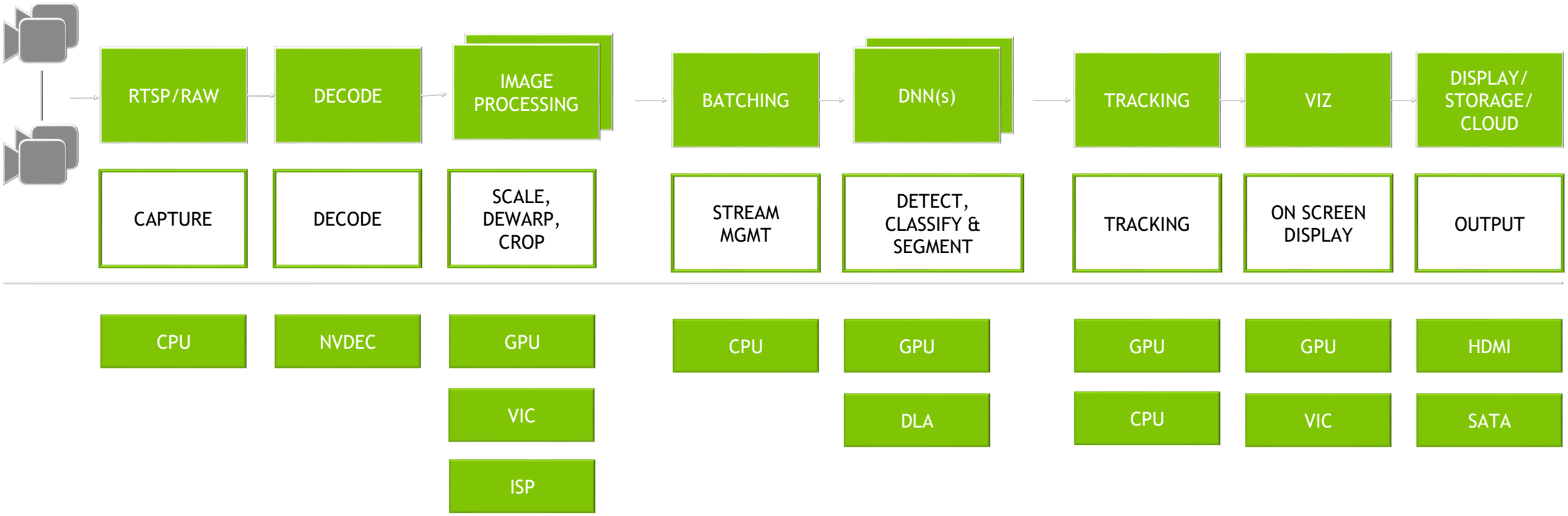
COMMON MIGRATING ANALYSIS

Cross compile tools chain



https://docs.nvidia.com/jetson/l4t/index.html#page/Tegra%20Linux%20Driver%20Package%20Development%20Guide/xavier_toolchain.html

JETSON MIGRATING :DEEPSTREAM



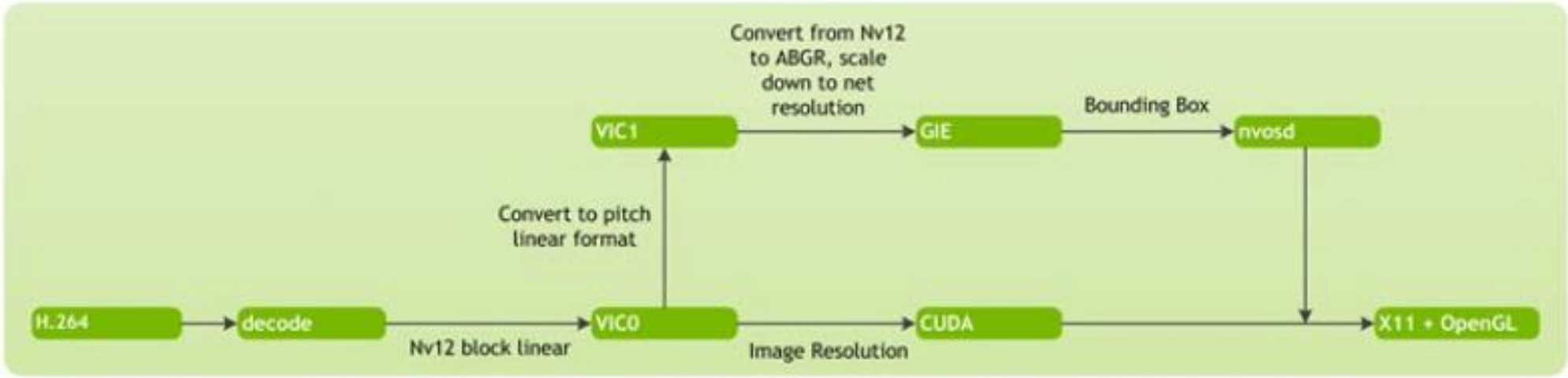
JETSON MIGRATING :MULTIMEDIA API

Directory Location Relative to ll_samples/samples	Description
00_video_decode (video decode)	Decodes H.264, H.265, VP8, VP9, MPEG4, and MPEG2 video from a local file and then shares the YUV buffer with egl renderer.
01_video_encode (video encode)	Encodes YUV bitstream from local file and then write elementary H.264/H.265 into file.
02_video_dec_cuda (CUDA processing with decode)	Decodes H.264/H.265 video from a local file and then shares the YUV buffer with CUDA to draw a black box in the left corner.
03_video_cuda_enc (CUDA processing with encode)	Use CUDA to draw a black box in the YUV buffer and then feeds it to video encoder to generate an H.264/H.265 video file.
04_video_dec_trt (TensorRT video decode)	Uses simple TensorRT calls to save the bounding box info to a file.
05_jpeg_encode (JPEG encode)	Uses `libjpeg-8b` APIs to encode JPEG images from software-allocated buffers.
06_jpeg_decode (JPEG decode)	Uses `libjpeg-8b` APIs to decode a JPEG image from software-allocated buffers.
07_video_convert (NvBuffer conversion)	Uses `v4l2` APIs to do video format conversion and video scaling.
08_video_dec_drm (Direct Rendering Manager)	Uses the NVIDIA® Tegra® Direct Rendering Manager (DRM) to render video stream or UI.
09_camera_jpeg_capture (libargus & libjpeg-8b)	Simultaneously uses Libargus API to preview camera stream and libjpeg-8b APIs to encode JPEG images.
10_camera_recording (libargus capture)	Gets the real-time camera stream from the Libargus API and feeds it into the video encoder to generate H.264/H.265 video files.
12_camera_v4l2_cuda (camera capture CUDA processing)	Captures images from a V4L2 camera and shares the stream with CUDA engines to draw a black box in the upper left corner.
13_multi_camera (multi image capture & composite)	Captures multiple cameras and composites them to one frame.
14_multivideo_decode (multi video decode)	Decodes multiple H.264, H.265, VP8, VP9, MPEG4, and MPEG2 videos from local files and writes YUV buffer into corresponding files.
15_multivideo_encode (multi video encode)	Encodes multiple YUV bitstreams from local files and writes elementary H.264/H.265/VP8/VP9 into corresponding files.
16_multivideo_encode (multi video transcode)	Transcodes multiple bitstreams from local files and writes elementary H.264/H.265/VP8/VP9 into corresponding files.
unittest_samples/camera_unit_sample (capture with libv4l2_nvargus)	Unit level sample; uses libv4l2_nvargus to preview camera stream.
unittest_samples/decoder_unit_sample (video decode unit sample)	Unit level sample; decodes H.264 video from a local file and dumps the raw YUV buffer.
unittest_samples/encoder_unit_sample (video encode unit samples)	Unit level sample; encodes YUV bitstream from a local file and writes elementary H.264 bitstream into file.
unittest_samples/transform_unit_sample (nvbuf_utils pixel format conversion)	Unit level sample; uses nvbuf_utils utility to convert one colorspace YUV bitstream to another.
backend (video analytics)	Performs intelligent video analytics on four concurrent video streams going through a decoding process using the on chip decoders, vi GPU compute.
frontend (TensorRT multichannel video capture)	Performs independent processing on four different resolutions of video capture coming directly from camera.
v4l2cuda (CUDA format conversion)	Uses V4L2 image capturing with CUDA format conversion.

https://docs.nvidia.com/jetson/l4t-multimedia/group__l4t__mm__test__group.html

MIGRATING : BACKEND (VIDEO ANALYTICS)

nsorRT.



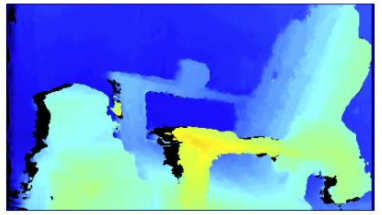
JETSON MIGRATING :VPI(VISION PROGRAMMING INTERFACE)

Uniform interface for seamless access Supports multiple compute engines:

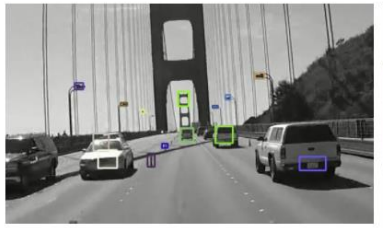
- Implements Computer Vision / Image Processing algorithms on GPU, CPU, PVA & VIC

Highly optimized

- Increase performance replacing non-performant OpenCV or VisionWorks



Stereo Disparity Estimator

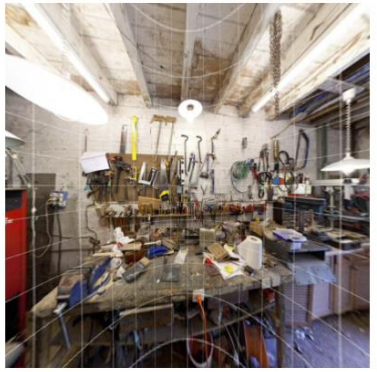


KLT Feature Tracker

Perspective Warp

Remap

Pyramidal LK Optical Flow



Lens Distortion Correction

Dense Optical Flow

Background Subtractor

[Vision Programming Interface (VPI) | NVIDIA Developer]

MIGRATING : VPI 1.1

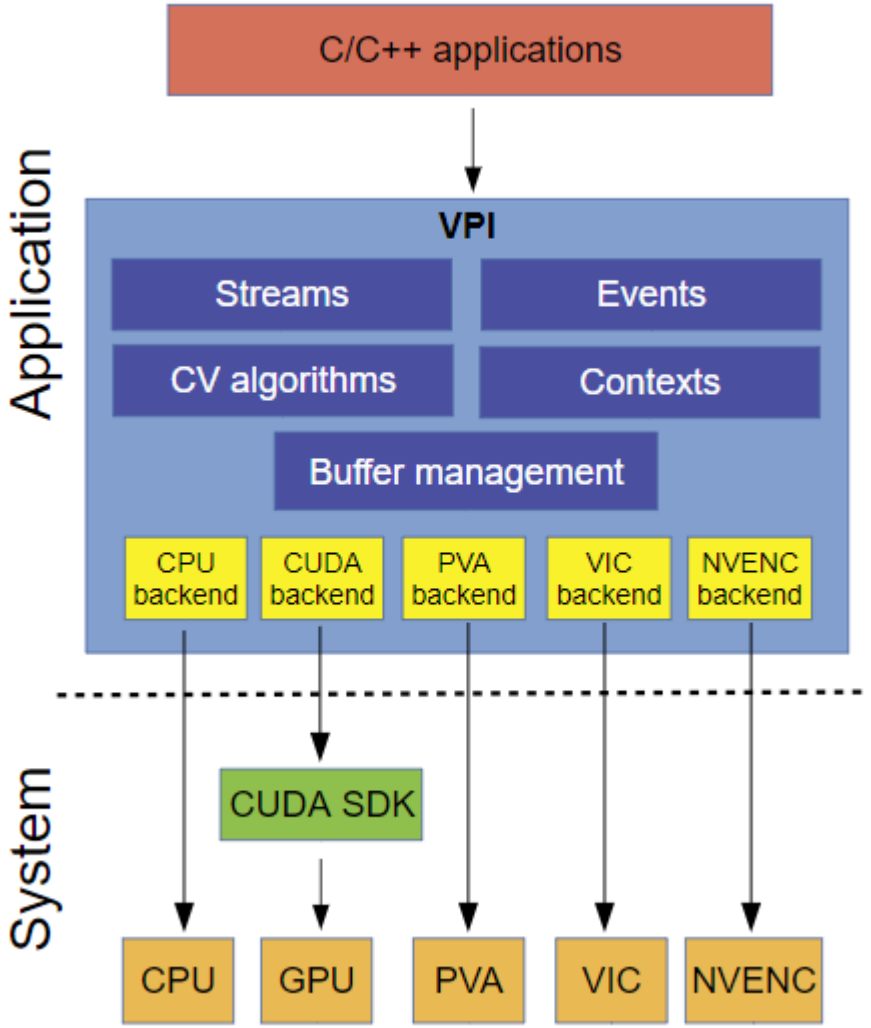
- Developer preview of Python binding
  All algorithms are exposed, except for KLT Template Tracker.

- New algorithms
  Pyramidal LK Optical Flow (NVENC) / Laplacian / Image Histogram / Histogram Equalization / Background Subtraction

- Added NVENC backend
  exposing some functionality provided by NVENC processor in Tegra devices

- New image formats
  Y8 Y16 NV24

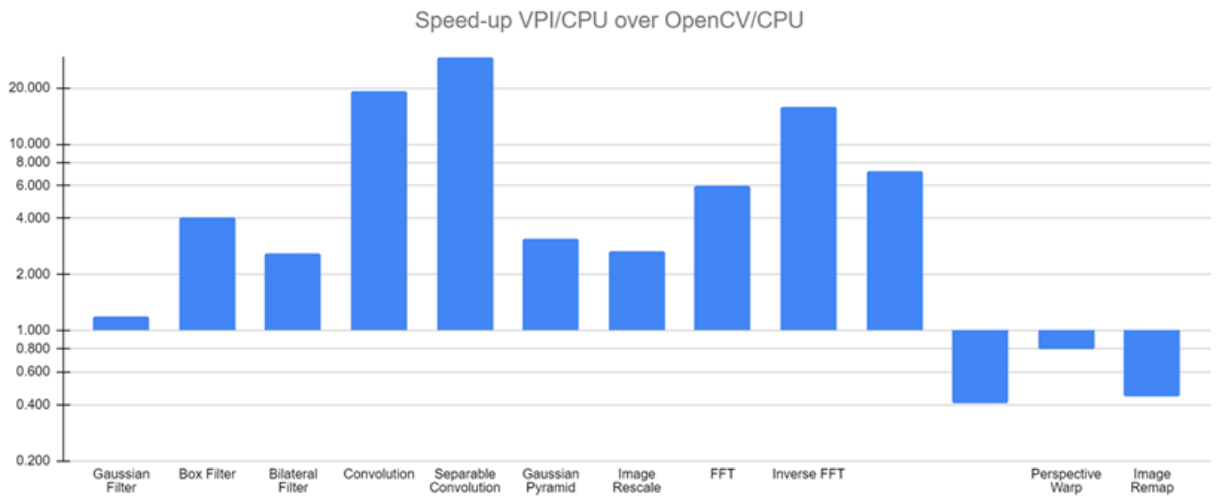
- **Rescale** (VIC), **Remap** (VIC), **Convert Image Format** (VIC, CUDA and CPU) algorithms now supports NV24 image format
- **Remap** on VIC now accepts U8 and U16 formats.



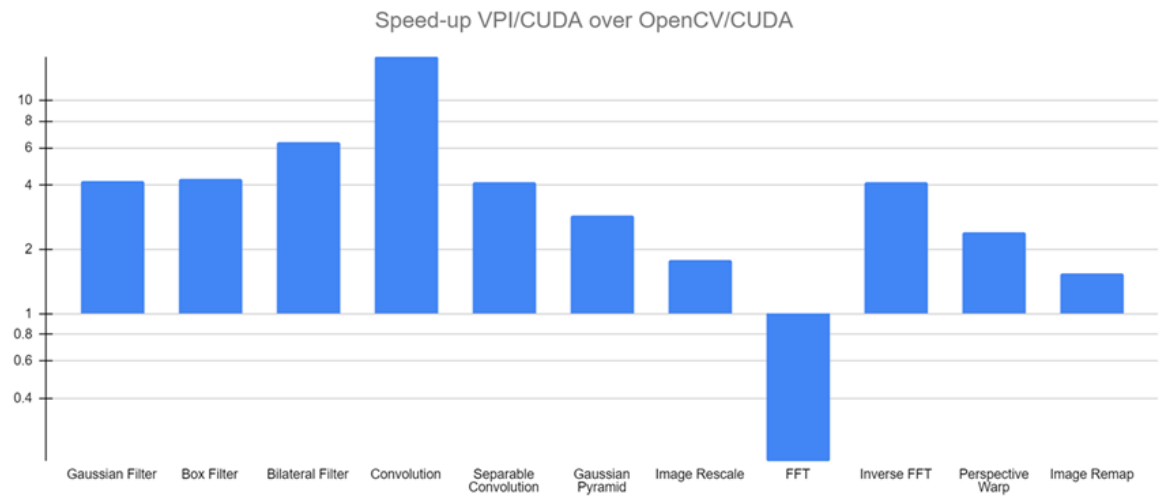
VPI BENCHMARKS

Significant Speed Up Compared to OpenCV

Up to 30X faster than OpenCV on CPU



Up to 15X faster than OpenCV on GPU



Graph above compares VPI 1.0 performance with OpenCV 4.1.1on Jetson AGX Xavier

JETSON MIGRATING : OPENCV

```
import sys
import cv2

def read_cam():
    cap = cv2.VideoCapture("rtspsrc
location=rtsp://wowzaec2demo.streamlock.net/vod/mp4:BigBuckBunny_115k.mov ! rtph264depay !
h264parse ! nvv4l2decoder ! nvvidconv ! video/x-raw, format=(string)BGRx ! videoconvert ! video/x-
raw,format=BGR ! appsink ")
    if cap.isOpened():
        cv2.namedWindow("demo", cv2.WINDOW_AUTOSIZE)
        while True:
            ret_val, img = cap.read();
            cv2.imshow('demo',img)
            cv2.waitKey(10)

    else:
     print "rtsp open failed"

    cv2.destroyAllWindows()


if __name__ == '__main__':
    read_cam()
```

MIGRATING ANALYSIS FOR DL MODEL

Jetson Migrating : TensorRT

SDK for High-Performance Deep Learning Inference

Optimize and Deploy neural networks in production

Maximize throughput for latency-critical apps with compiler & runtime

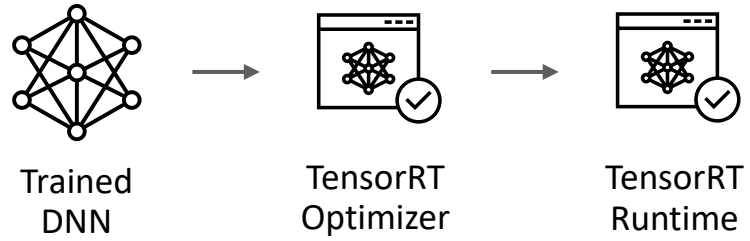
Deploy responsive and memory efficient apps

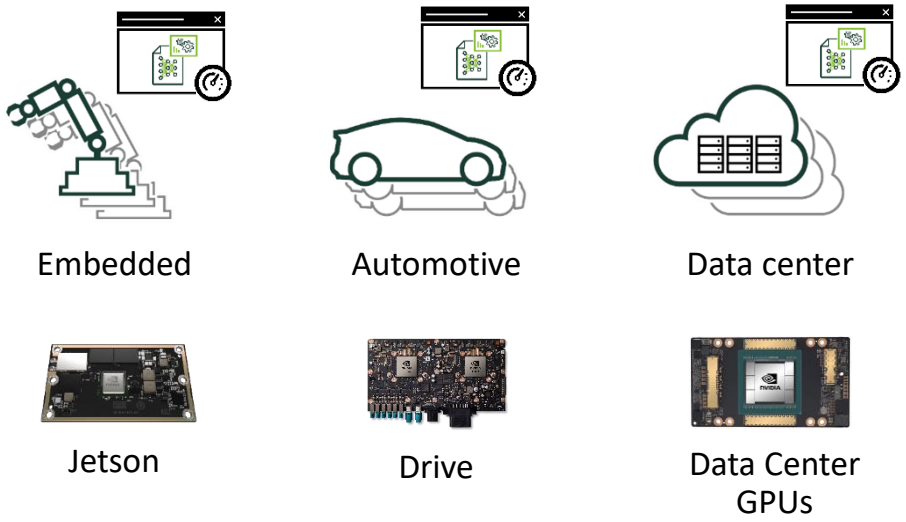
FP32, TF32, FP16 & INT8

Optimize every network including CNNs, RNNs and Transformers

Accelerate every framework - ONNX support, TensorFlow integration

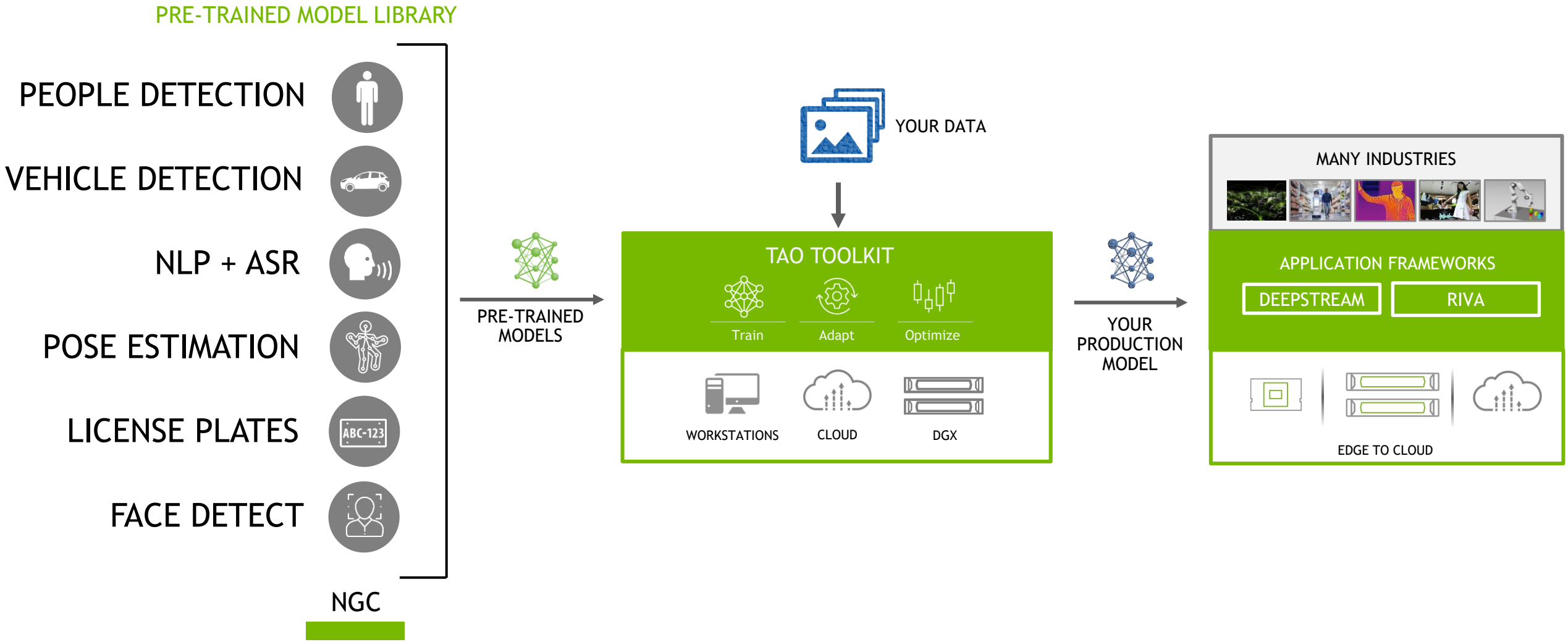
Run multiple models on a node with containerized inference server





developer.nvidia.com/tensorrt

JETSON MIGRATING : TAO



AFTER JETSON MIGRATING : TRTEXEC

- It’s useful for *benchmarking networks* on random or user-provided input data.
- It’s useful for *generating serialized engines* from models.

```
[10/28/2021-23:44:31] [I] Average on 10 runs - GPU latency: 0.384033 ms - Host latency: 0.398389 ms (end to end 0.410693 ms, enqueue 0.297998 ms)
[10/28/2021-23:44:31] [I] Average on 10 runs - GPU latency: 0.371533 ms - Host latency: 0.380811 ms (end to end 0.394434 ms, enqueue 0.290308 ms)
[10/28/2021-23:44:31] [I] Average on 10 runs - GPU latency: 0.378052 ms - Host latency: 0.39165 ms (end to end 0.404321 ms, enqueue 0.293994 ms)
[10/28/2021-23:44:31] [I] Average on 10 runs - GPU latency: 0.374902 ms - Host latency: 0.384131 ms (end to end 0.399316 ms, enqueue 0.295801 ms)
[10/28/2021-23:44:31] [I] Average on 10 runs - GPU latency: 0.374805 ms - Host latency: 0.390356 ms (end to end 0.40332 ms, enqueue 0.294043 ms)
[10/28/2021-23:44:31] [I] Average on 10 runs - GPU latency: 0.374854 ms - Host latency: 0.390527 ms (end to end 0.403003 ms, enqueue 0.289136 ms)
[10/28/2021-23:44:31] [I] Average on 10 runs - GPU latency: 0.372754 ms - Host latency: 0.382178 ms (end to end 0.394971 ms, enqueue 0.289819 ms)
[10/28/2021-23:44:31] [I] Average on 10 runs - GPU latency: 0.378882 ms - Host latency: 0.388403 ms (end to end 0.402539 ms, enqueue 0.292432 ms)
[10/28/2021-23:44:31] [I] Average on 10 runs - GPU latency: 0.379224 ms - Host latency: 0.388623 ms (end to end 0.401123 ms, enqueue 0.295972 ms)
[10/28/2021-23:44:31] [I] Average on 10 runs - GPU latency: 0.376245 ms - Host latency: 0.386133 ms (end to end 0.399048 ms, enqueue 0.295312 ms)
[10/28/2021-23:44:31] [I] Average on 10 runs - GPU latency: 0.384497 ms - Host latency: 0.394141 ms (end to end 0.406372 ms, enqueue 0.298315 ms)
[10/28/2021-23:44:31] [I]
[10/28/2021-23:44:31] [I] === Performance summary ===
[10/28/2021-23:44:31] [I] Throughput: 2305.45 qps
[10/28/2021-23:44:31] [I] Latency: min = 0.350342 ms, max = 0.681244 ms, mean = 0.41922 ms, median = 0.420349 ms, percentile(99%) = 0.509949 ms
[10/28/2021-23:44:31] [I] End-to-End Host Latency: min = 0.361816 ms, max = 0.699432 ms, mean = 0.432425 ms, median = 0.433105 ms, percentile(99%) = 0.523621
ms
[10/28/2021-23:44:31] [I] Enqueue Time: min = 0.260254 ms, max = 0.591461 ms, mean = 0.318292 ms, median = 0.313721 ms, percentile(99%) = 0.41626 ms
[10/28/2021-23:44:31] [I] H2D Latency: min = 0.00585938 ms, max = 0.140259 ms, mean = 0.0110233 ms, median = 0.00610352 ms, percentile(99%) = 0.0441895 ms
[10/28/2021-23:44:31] [I] GPU Compute Time: min = 0.341064 ms, max = 0.663513 ms, mean = 0.404844 ms, median = 0.405518 ms, percentile(99%) = 0.490509 ms
[10/28/2021-23:44:31] [I] D2H Latency: min = 0.00292969 ms, max = 0.0924072 ms, mean = 0.00335495 ms, median = 0.0032959 ms, percentile(99%) = 0.00369263 ms
[10/28/2021-23:44:31] [I] Total Host Walltime: 3.00071 s
[10/28/2021-23:44:31] [I] Total GPU Compute Time: 2.80071 s
[10/28/2021-23:44:31] [I] Explanations of the performance metrics are printed in the verbose logs.
[10/28/2021-23:44:31] [I]
```

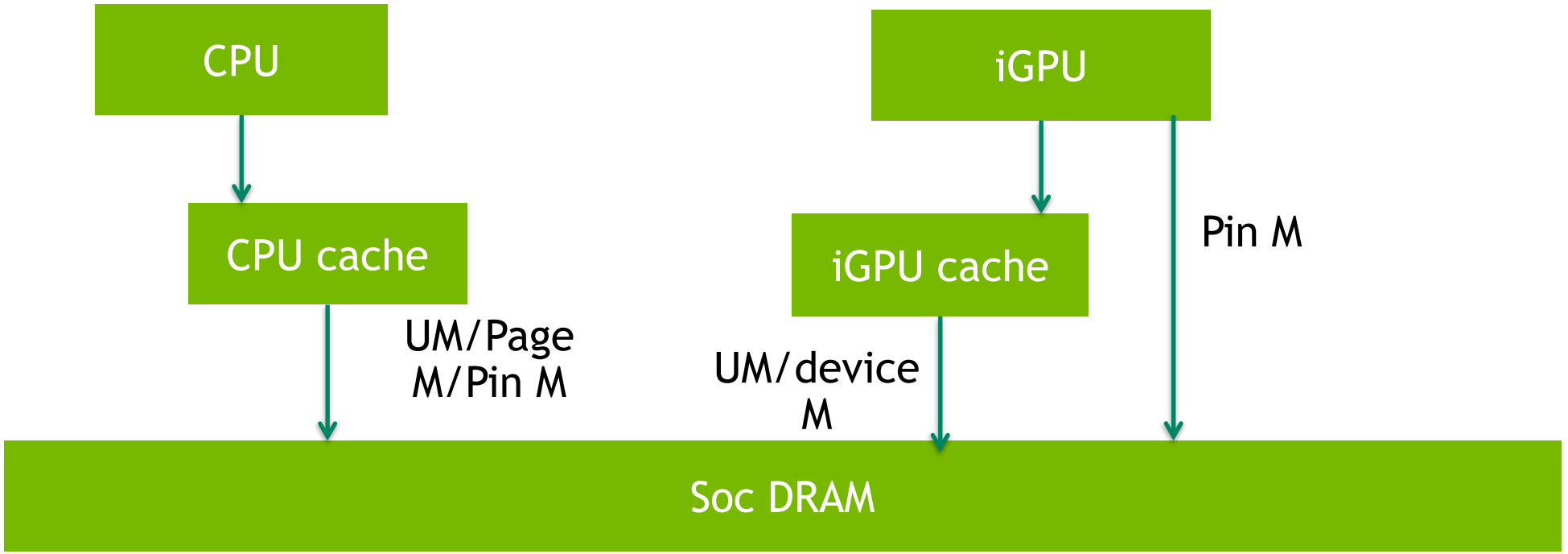
https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html#trtexec


MIGRATING ANALYSIS FOR CUDA

MEMORY MANAGEMENT FOR TEGRA

CPU (Host) and the iGPU share SoC DRAM memory.

Page mem, Pin mem, unify mem



CUDA FOR TEGRA

Memory： device mem/pinned mem/unified mem

I/O coherency is supported on Tegra devices starting with Xavier SOC

Table 1. Characteristics of Different Memory Types in a Tegra System

Memory Type	CPU	iGPU	Tegra®-connected dGPU
Device memory	Not directly accessible	Cached	Cached
Pageable host memory	Cached	Not directly accessible	Not directly accessible
Pinned host memory	Uncached where compute capability is less than 7.2. Cached where compute capability is greater than or equal to 7.2.	Uncached	Uncached
Unified memory	Cached	Cached	Not supported

https://docs.nvidia.com/cuda/cuda-for-tegra-appnote/index.html

PORTING CONSIDERATIONS

- Memory Selection

Device Memory / Pageable Host Memory / Pinned Memory(small buffers**) / Unified Memory**

Memory Type	Recommended	Not Recommended
Conventional Memory	For large buffers with no or minimal data exchange between CPU and GPUs.	Frequent migration of data between CPU and GPU
Pinned Memory	For frequent data access from both CPU and GPU specifically for small buffers with less repetitive data access. Can also be used for large buffers when GPU only needs to access data once in a coalescing manner.	Repeated data access
Unified Memory	For frequent data access from both CPU and GPU with highly repetitive access (to offset the cost of cache coherency maintenance)	For applications with high predictability needs and low latency.

SIMPLEZEROCOPY

CUDA/samples/0_Simple/simpleZeroCopy(small data)

1： pinned-memory ->zerocopy， no DtoH/HtoD

2：add paged-memory in demo for compared with Pin memory，add H2D/D2H ， --use_generic_memory

```
==16976== Profiling application: ./simpleZeroCopy --use_generic_memory
==16976== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   62.25%  12.022ms        10  1.2022ms  1.1973ms  1.2057ms  vectorAddGPU(float*, float*, float*, int)
                   19.55%  3.7753ms         1  3.7753ms  3.7753ms  3.7753ms  [CUDA memcpy DtoH]
                   18.20%  3.5158ms         2  1.7579ms  1.3471ms  2.1687ms  [CUDA memcpy HtoD]
      API calls:   90.66%  238.30ms         3  79.432ms  370.47us  236.79ms  cudaMalloc
                    4.93%  12.951ms        10  1.2951ms  1.2568ms  1.4799ms  cudaDeviceSynchronize
                    3.64%  9.5787ms         3  3.1929ms  1.6317ms  6.0785ms  cudaMemcpy
```

Paged-memory

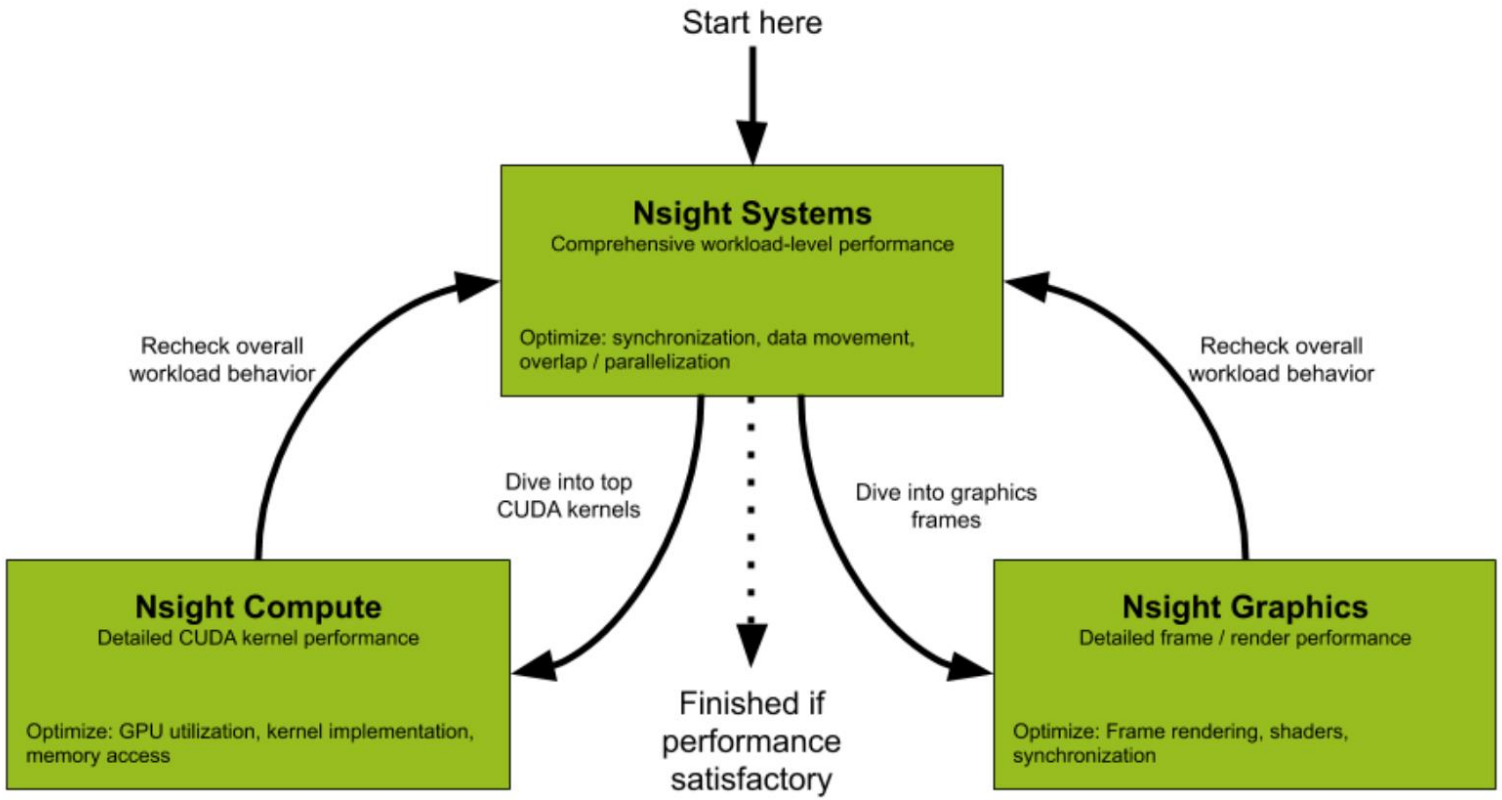
1：CUDA mem DtoH 3ms
2：CUDA mem HtoD 3ms
3： vectorAddGPU 1.2ms(paged)
->1.09ms(pinned)

```
==16422== Profiling application: ./simpleZeroCopy
==16422== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  10.910ms        10  1.0910ms  1.0886ms  1.0995ms  vectorAddGPU(float*, float*, float*, int)
      API calls:   94.11%  254.36ms         3  84.788ms  554.24us  252.38ms  cudaHostAlloc
                    4.68%  12.659ms        10  1.2659ms  1.1453ms  2.1060ms  cudaDeviceSynchronize
                    0.70%  1.8936ms         3  631.20us  527.08us  747.21us  cudaFreeHost
                    0.36%  982.57us        10  98.256us  32.576us  156.64us  cudaLaunchKernel
                    0.07%  188.55us        97  1.9430us     992ns  35.585us  cuDeviceGetAttribute
```

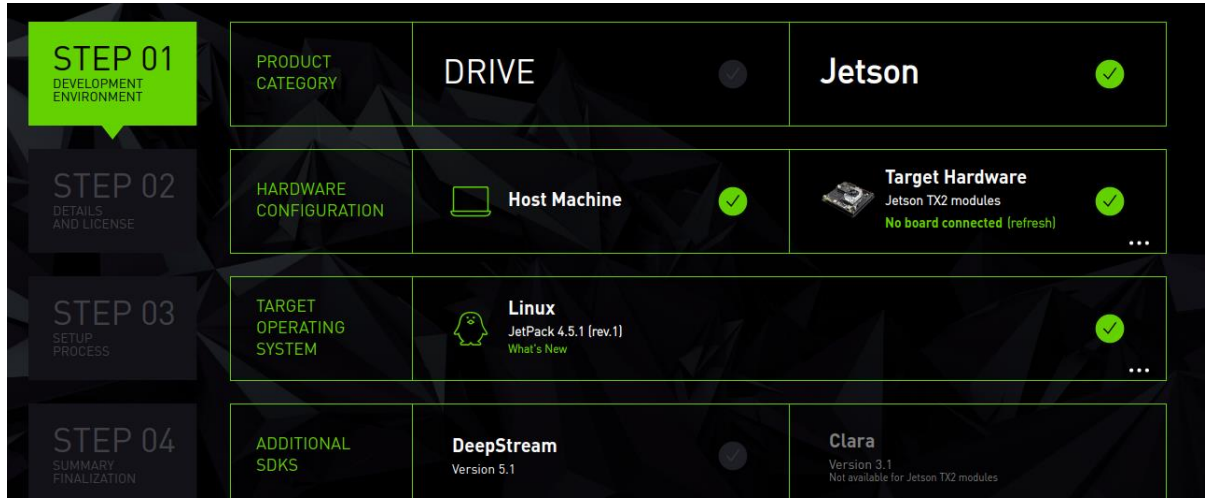
Pinned-memory

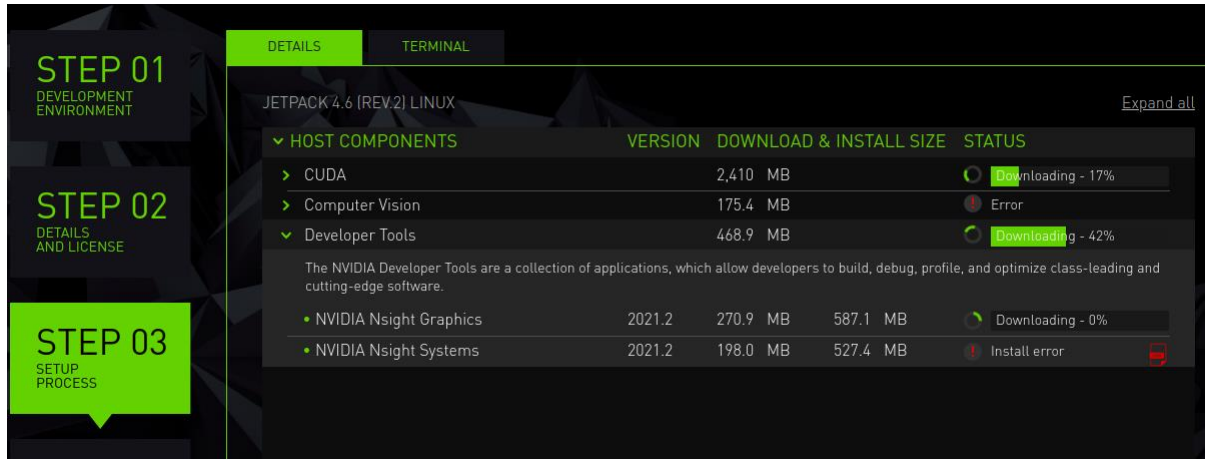
PERF PROFILING

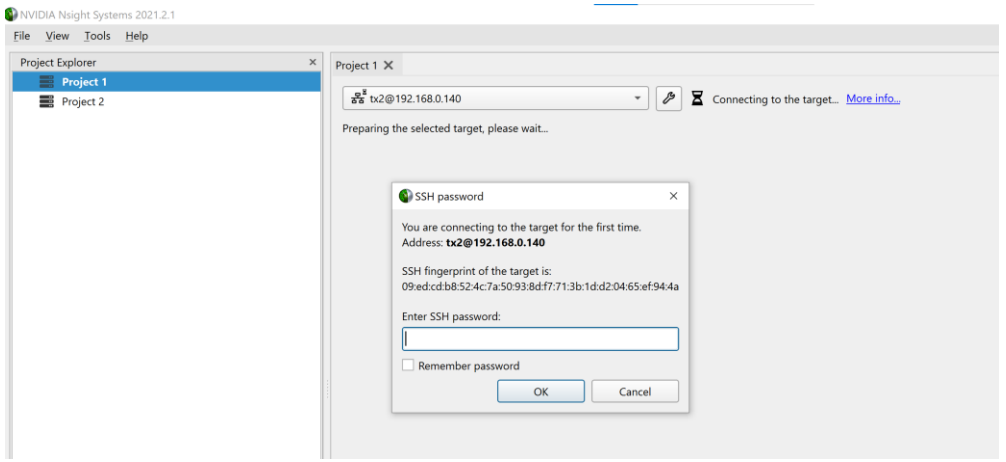
NSIGHT TOOLS



JETSON MIGRATING：NSIGHT SYSTEM

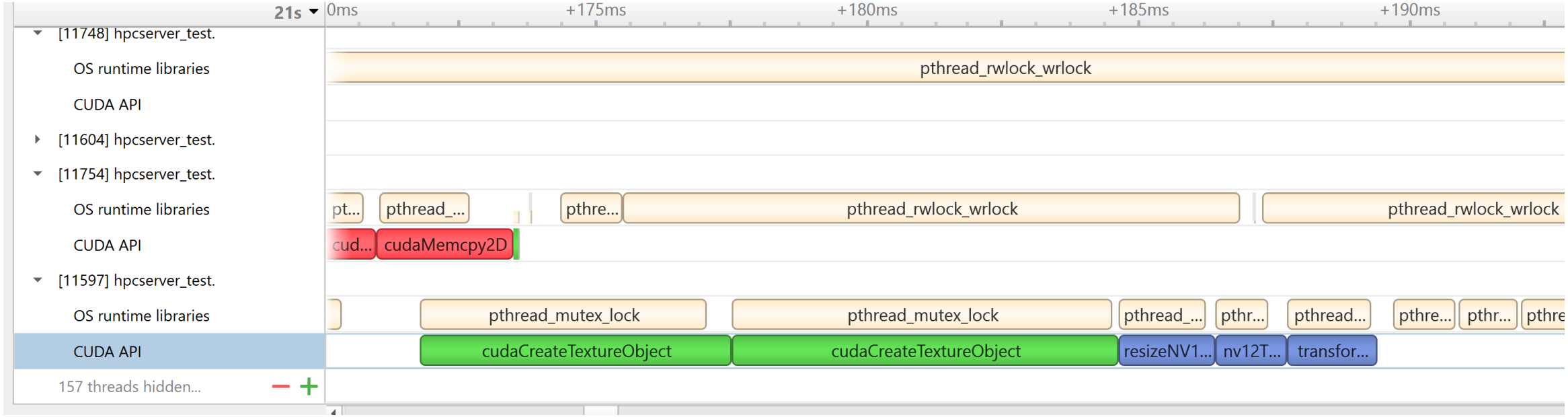






https://developer.nvidia.com/nsight-systems

JETSON MIGRATING : NSIGHT SYSTEM SAMPLE



NVPROF

profiling data from the command-line

```
$ nvprof matrixMul
[Matrix Multiply Using CUDA] - Starting...
==27694== NVPROF is profiling process 27694, command: matrixMul
GPU Device 0: "GeForce GT 640M LE" with compute capability 3.0

MatrixA(320,320), MatrixB(640,320)
Computing result using CUDA Kernel...
done
Performance= 35.35 GFlop/s, Time= 3.708 msec, Size= 131072000 Ops, WorkgroupSize= 1024 threads/block
Checking computed result for correctness: OK

Note: For peak performance, please refer to the matrixMulCUBLAS example.
==27694== Profiling application: matrixMul
==27694== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 99.94%  1.11524s       301  3.7051ms  3.6928ms  3.7174ms  void matrixMulCUDA<int=32>(float*, float*, float*, int, int)
  0.04%  406.30us         2  203.15us  136.13us  270.18us  [CUDA memcpy HtoD]
  0.02%  248.29us         1  248.29us  248.29us  248.29us  [CUDA memcpy DtoH]

==27964== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 49.81%  285.17ms         3  95.055ms  153.32us  284.86ms  cudaMalloc
 25.95%  148.57ms         1  148.57ms  148.57ms  148.57ms  cudaEventSynchronize
 22.23%  127.28ms         1  127.28ms  127.28ms  127.28ms  cudaDeviceReset
  1.33%  7.6314ms       301  25.353us  23.551us  143.98us  cudaLaunch
  0.25%  1.4343ms         3  478.09us  155.84us  984.38us  cudaMemcpy
  0.11%  601.45us         1  601.45us  601.45us  601.45us  cudaDeviceSynchronize
```

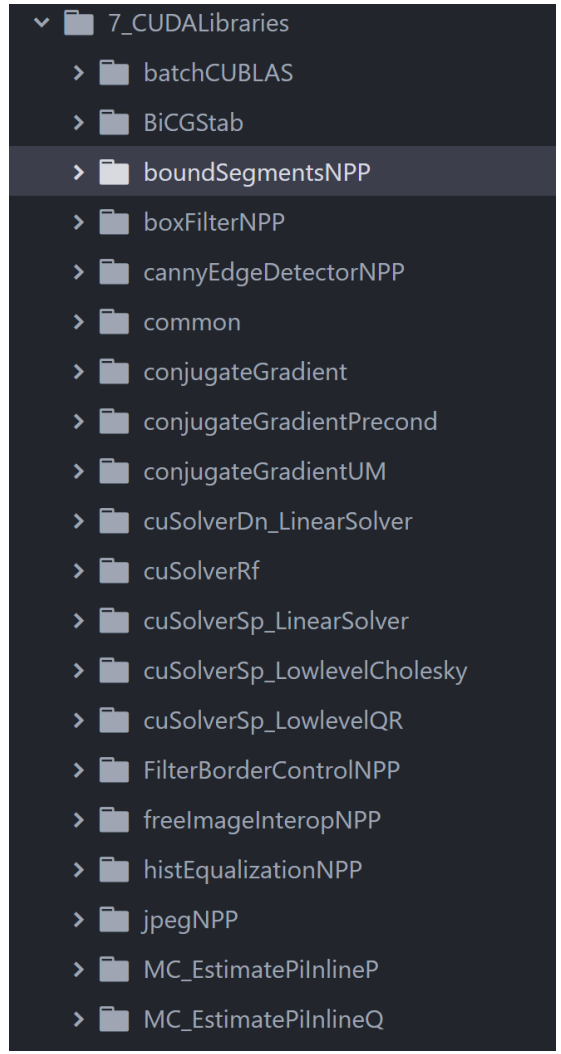
/usr/local/cuda/bin/nvprof --metrics achieved_occupancy --metrics gld_throughput ../../bin/aarch64/linux/release/simpleZeroCopy

NPP:NVIDIA 2D IMAGE AND SIGNAL PERFORMANCE PRIMITIVES

What is NPP?
NVIDIA NPP is a library of functions for performing CUDA accelerated 2D image and signal processing

NPP can be used in one of two ways:
•A stand-alone library for adding GPU acceleration to an application with minimal effort. Using this route allows developers to add GPU acceleration to their applications in a matter of hours.
•A cooperative library for interoperating with a developer's GPU code efficiently.

•boundSegmentsNPP
•cannyEdgeDetectorNPP
•FilterBorderControlNPP



RESOURCES

https://developer.nvidia.com/blog/migrating-nvidia-nsight-tools-nvvp-nvprof/

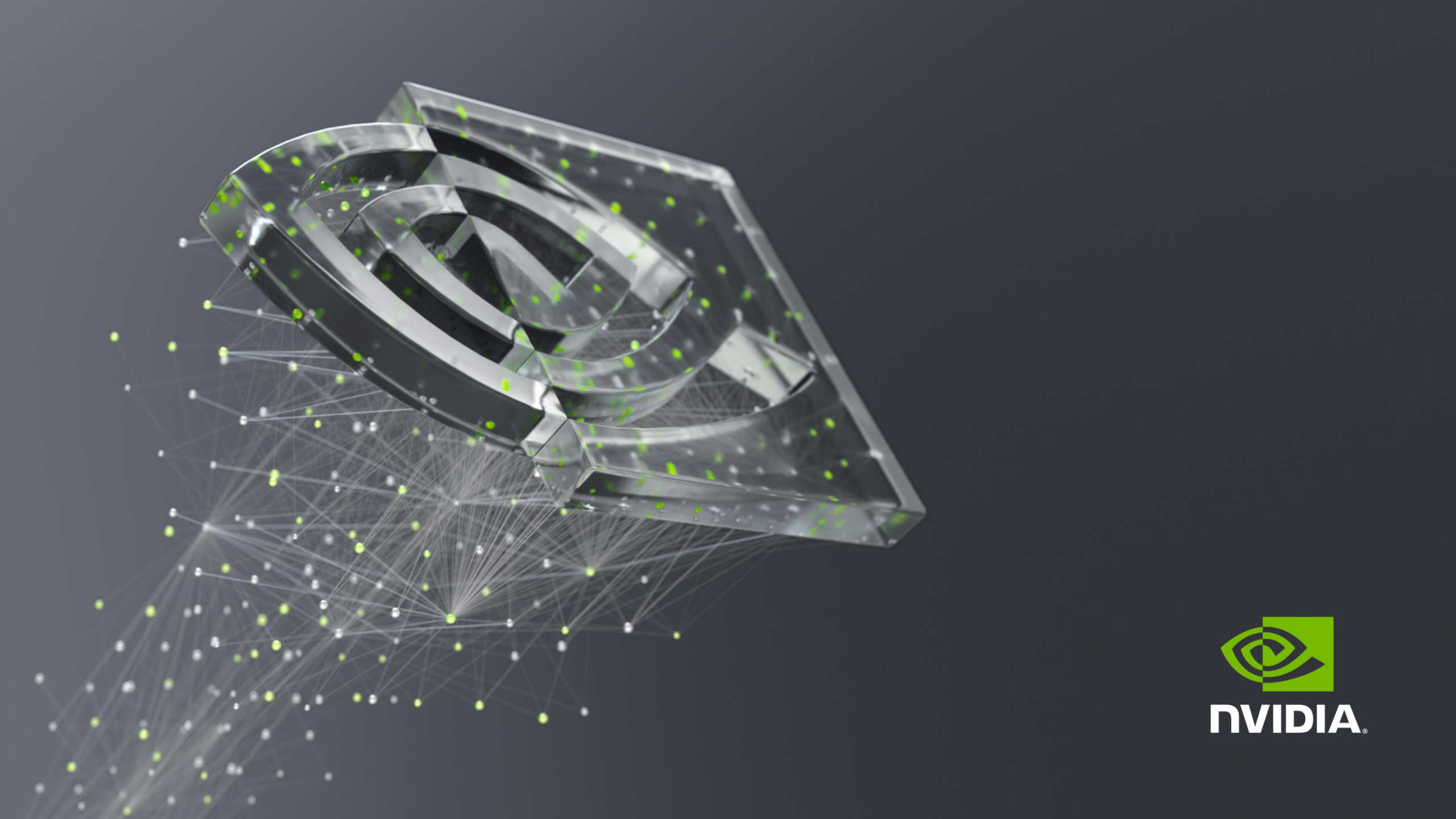
https://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-overview

CUDA C 语言编程指南： http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

CUDA C 语言最佳实践指南： http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

适用于 Tegra 的 CUDA： https://docs.nvidia.com/cuda/cuda-for-tegra-appnote/index.html

https://docs.nvidia.com/cuda/npp/index.html


NVIDIA