THE STORY

Your temporary identity

- Your company sells a camera system
- Expanding into intelligent video analytics
  - Capture video
  - Recognition objects
  - Take action in real-time
  - Visualize
- You heard of Jetson TX2
  - Powerful hardware accelerated image inference
  - Comprehensive tools to debug and profile
THE STORY

Your application

- Prototype extends video playback software
- Accelerated video decode (V4L2)
- Prepare images for TensorRT (inference)
- TensorRT for accelerated recognition of objects in the video
- OpenGL ES to render the video and highlight the recognized objects
DEVELOPMENT FOR THE JETSON TX2

The Setup

x86_64 Ubuntu 16.04
CUDA 8.0 cross development toolkit

Jetson TX2
ARMv8 Ubuntu 16.04
CUDA 8.0 armv8-linux-gnueabihf toolkit

Ethernet/SSH
DEVELOPMENT FOR THE JETSON TX2

The Tools

- Setup
- CUDA GDB
  - CUDA Memcheck
  - Tegra Graphics Debugger
- Debug
- Tegra System Profiler
- Optimize Application Structure
- Visual Profiler
- Tune CUDA Kernels & OpenGL Shaders
- Tegra Graphics Debugger
NVIDIA® NSIGHT™ ECLIPSE EDITION
Integrated CUDA Development Environment

Edit, build and debug CUDA applications
CUDA aware source code editor
CUDA-GDB debugger for the CPU and GPU
Support for cross-platform development
NVIDIA® NSIGHT™ ECLIPSE EDITION
CROSS COMPILE SETTINGS

Verify the compiler and linker settings from project property page (Right click on the project->Properties)

➢ Target CPU architecture must be set to AArch64
➢ Add Library search path(-L) to the location of target libraries in the CUDA toolkit.
➢ You can choose to install the cross compile packages when installing CUDA toolkit (Jetpack or Desktop installer).

Build the project using 🛠 button in the main toolbar.
Verify the remote debug settings in the launch configuration.

Go to Run-> Debug configuration...-> C/C++ Remote application-> Step1_Debug

- Remote Tab - Enter the remote connection details
- Local Tab - Verify the project and binary details
NVIDIA® NSIGHT™ ECLIPSE EDITION
Remote Debug Configuration

Arguments Tab - Verify the program arguments.

Environment Tab - Make sure that the DISPLAY variable is set to :0
CUDA-GDB Debugger Settings:

- Stop at main is unselected (to continue quicker in this demo)
- “Download shared libraries from target” and “Load shared symbols automatically” options are selected, to get target library debugging symbols
NVIDIA® NSIGHT™ ECLIPSE EDITION
CUDA GDB Debug Session

Launch debug session by clicking on “Debug” button

Verify that the debugger stops at an error in the kernel

Check the debug views to inspect the values
  ➢ View program variables and registers across several CUDA threads
  ➢ View, Navigate and filter to selectively track execution across threads
  ➢ Set breakpoints and single-step execution at both source-code and assembly levels
  ➢ View memory and disassembly
Check the debug views to inspect the values

➢ Note that pDevPtr is NULL (CudaKernel.cu: Line 190)

➢ This is the pointer to the input data, so we cause a NULL-ptr dereference here

➢ Fix the launch site of the kernel to pass in the correct variable

➢ Re-build the app

➢ Run the app again to verify that the error is fixed and doesn’t crash anymore
CUDA MEMCHECK
Tools for CUDA Correctness Checking

The app does not crash anymore, but we should still check for “silent” memory errors causing undefined/wrong results.

$ cuda-memcheck --tool <TOOL> <app>

Memcheck

  Out-of-bounds/misaligned memory accesses, memory leaks

  Integrated into cuda-gdb (set cuda memcheck on)

Synccheck - Divergence checker for synchronization primitives

Racecheck - Shared memory data access hazard detector

Initcheck - Uninitialized device global memory access detector
CUDA MEMCHECK
Tools for CUDA Correctness Checking

Verify that Debug Configuration has “Enable CUDA Memcheck” enabled in Debugger tab.

Debug the program again with memcheck enabled.
CUDA MEMCHECK
Tools for CUDA Correctness Checking

Verify that Debug Configuration has “Enable CUDA Memcheck” enabled in Debugger tab.

Debug the program again with memcheck enabled

- Stops at the offending source code line
- Found invalid access to constant buffer by all thread
- Array index off by one
- Due to limited time, note that the program runs only a few frames in this exercise
IT WORKS! But how does it perform?

Performance analysis
- Low overhead sampling and tracing
- CPU and GPU activity
- Rich trace events
- Call-stack sampling
- Avoid false positives on “GPU-bound”
SYSTEM PROFILER

Exercise: Annotating tensorrt_thread with ranges

- Open Step3 > main_tensorrt.cpp
- Line 64 - near top of `tensorrt_thread()`
  - `nvtxRangePush("hello world");
    do that something
    `nvtxRangePop();`
  - Change the messages to whatever you would like
  - Run it
    - We will use this in the next step
- NVTX works across all the profilers in this lab
- Learn about more about NVTX features online
SYSTEM PROFILER

Hands-on Introduction

- Open the System Profiler

- Select *Project 1*
  - Options for profiling
  - Tab may already be open
SYSTEM PROFILER

Main parts

• Device connection in toolbar
• Top half is timeline
• Bottom half is a function table with call-stack statistics
SYSTEM PROFILER
Timeline Row Hierarchy

- CPU > Cores > Utilization Avg(black) & Max(gray)
- Threads (most are hidden)
  - Utilization (time given by kernel)
  - CPU Core Occupancy
  - Thread State (running, blocked)
  - Event Trace
    - Various APIs (CUDA, OpenGL, NVTX user annotations)
- CUDA GPU
  - Open that up
  - See all kernels & when running
  - See how busy the GPU is
  - But there is a big empty block of opportunity to fill
SYSTEM PROFILER

Timeline

- Lets now zoom in to see the details
  - CTRL + MOUSE WHEEL
- See the frame pattern on CUDA API and GPU kernels
- On the threads:
  - Without work you get
    - Blocked state backtrace
    - API trace (CUDA, GL)
  - With a little work
    - User annotations with NVTX API
What our threads are doing:

- Dec_feed_loop thread is doing V4L2 decode
- Conv0_capture is V4L2 pitch linear conversion
- Render_thread
  - Mostly waiting, renders when data is available
- TensorRT_thread
  - CUDA, synchronizing, and copying data
Exercise 1: Introduction

- Function Statistics Table
  - Fast 10khz!
  - Made with call-stack sampling.
  - read_decoder_input_nalu
  - nanosleep
  - Explore algorithm cost (flat view - stack%)
SYSTEM PROFILER

Timeline: Conclusions

- Good pipelining
  - Parallelism with video decode, TensorRT, and render
  - Lots of waiting threads, might be nice to have a job system instead
  - Could the long pole thread be made faster? Or split into more parallel stages?

- GPU opportunities
  - Fill bubbles with better CPU-side GPU API usage
  - Use more efficient memory reused and transfer techniques
  - Merge and optimize CUDA kernels
  - You’ll see NVVP and TGD in a few minutes to dive in deeper
Program optimization is often an iterative process on multiple levels.

After optimizing the overall program performance, we can look into fine-tuning individual kernels.
VISUAL PROFILER
Tuning CUDA with Automated Performance Analysis

Setup Release build for performance analysis

Launch the application in Visual Profiler

Inspect timeline and individual kernels, memory copies, NVTX, ...

Use automated analysis to detect performance problems

Optimize kernels and verify results
VISUAL PROFILER
Tuning CUDA with Automated Performance Analysis
VISUAL PROFILER

Optimization Area
VISUAL PROFILER
Tuning CUDA with Automated Performance Analysis

Kernel converts image format and reduces resolution by half by averaging 4 pixels
Currently loops over individual channels and input pixels (2*2*3 accesses per thread)
Results in very inefficient global memory access pattern
Better: Load two pixels (with 3 channels each) at a time per thread by reading a `uint64_t`

Use bitmasks/shifts to select individual channel values from the `uint64_t`

Enable by switching to `prepareForTensorRtKernelRGB_fast` kernel
Verify global memory access issues are fixed

Re-build application in NsightEE and run once to deploy to Jetson

Re-run **Kernel Performance** in Visual Profiler and **Global Memory Access Pattern** analysis on this kernel

Results in perfectly coalesced global memory accesses and one transaction per access
TEGRA GRAPHICS DEBUGGER

Debug and Profile OpenGL Applications

Real time experiments and hardware performance monitoring
Debug OpenGL state via live frame capture
Frame serialization via C++ source code generation
Dynamic shader editing
Automated GPU bottleneck analysis
Advanced Range Profiler for finding performance problems
Shader performance statistics
TEGRA GRAPHICS DEBUGGER

Exercise 1: Missing Edge Detection
TEGRA GRAPHICS DEBUGGER

Exercise 1: Missing Edge Detection

1) Run the demo
   1) In Eclipse, click the down arrow next to the larger green circle
   2) Select “Step 5 Graphics Debugger”
   3) Double click TGD on the desktop (have already running?)
   4) Attach

2) We are using a blur + edge detect to try and find the outlines of the vehicles

3) Note that you don’t see the edge detection on the cars

4) Capture the frame
TEGRA GRAPHICS DEBUGGER

Exercise 1: Missing Edge Detection

1) Use the scrubber to scrub through the rendering
2) Note the first “Render Target Range” & scrub to that draw call
3) Open API Inspector to inspect the state for that draw call (explain API Inspector briefly)
4) Select the “FB”/Frame Buffer section from the pipeline navigation
5) Click on the name link for GL_COLORATTACHMENT0 to bring that up in the Resources View (explain Resources View briefly)
6) Note that the resource is all brown, so our blur pass is not working properly
TEGRA GRAPHICS DEBUGGER

Exercise 1: Missing Edge Detection

1) Back to the API Inspector, select the “FS” page so we can inspect the shader

2) Edit the shader in main to not do the blur but to just output “vec4(texCoord, 0, 1);”, compile the shader, and note the output shows a single texture coordinate being displayed.

3) So, the bug must be somewhere else in the setup
TEGRA GRAPHICS DEBUGGER

Exercise 1: Missing Edge Detection

1) Go to the VS page to see if there is a problem with our vertex setup

2) Note in the Program Input section, there are 2 attributes, “in_pos” for position at location 0 and “in_tc” for texture coordinates at location 1
TEGRA GRAPHICS DEBUGGER

Exercise 1: Missing Edge Detection

1) Now, go to the Vtx Spec section and let’s see if we are feeding these inputs correctly

2) Note that index/location 0 for position is enabled, but index/location 1 for texture coords is not

3) This points to a bug in our code. Go to Eclipse and open the project to get access to NvEglRenderer.cpp
1) This points to a bug in our code. Go to Eclipse and open the project to get access to NvEglRenderer.cpp

2) Search for the “renderInternal” function around line 440 and start looking through that code.

3) You see that we get to the loop for the calculation and set get the attribute location for in_pos and the vertex attrib array for pos_location is enabled.

4) Then, we get the attribute location for in_tc, but the vertex attrib array is not enabled for tc_location.

5) Uncomment that line “glEnableVertexAttribArray(tc_location);”
TEGRA GRAPHICS DEBUGGER

Exercise 2: Tuning the Detection Kernel
1) Let’s start by some performance profiling of our current Kernel

2) Run “Step 6 Graphics Profiler”

3) Capture a frame from the application

4) Right click on the scrubber and select “Profile Frame”

5) Explain briefly the different Range Profiler sections

6) Select the that first render target range we were looking at before
1) In the middle section, you will note that the shader and texture inefficiency values are pretty high, 56% & 47% respectively.

2) Inefficiency is a mix of Speed of Light (or SOL) and Busy.
   1) SOL = % of the maximum amount of work that could have been done on the GPU during the experiment
   2) Busy = how active the unit was during the experiment
   3) Low SOL + low busy, not as big of a deal, pipeline might not even have been filled
   4) High SOL + high busy = try and reduce the amount of work being done
   5) Med/low SOL + high busy = try and make the work more efficient
TEGRA GRAPHICS DEBUGGER

Exercise 2: Tuning the Detection Kernel

Since we know we are doing a blur, we can see that grabbing all of those taps is costly, especially in the texture and shader units.

<table>
<thead>
<tr>
<th>Pipeline Overview</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Shader Inefficiency</td>
<td>52.57%</td>
</tr>
<tr>
<td>Texture Inefficiency</td>
<td>51.74%</td>
</tr>
</tbody>
</table>
TEGRA GRAPHICS DEBUGGER

Exercise 2: Tuning the Detection Kernel

1) Scrub to for draw event
2) Open the API Inspector and select the “FS” section
3) Click on the shader to open the editor
4) Inside of blur_bilateral, look at the KSize. Change it from 20 to 4 and refresh the profiler.
5) Note the tex & shader inefficiency go down
WRAP UP

The Tools

- Easy cross-platform development
- Debug kernels, check API errors and memory accesses
- Optimize application-level performance
- Optimize CUDA kernels and GL shaders

**Setup**
- CUDA GDB
- CUDA Memcheck
- Tegra Graphics Debugger

**Debug**
- Visual Profiler
- Tegra Graphics Debugger

**Optimize Application Structure**
- Tegra System Profiler

**Tune CUDA Kernels & OpenGL Shaders**
GETTING STARTED...

JetPack
For Linux Ubuntu 16.04

Jump starts developing on Jetson/DrivePX/DriveCX
Installs Linux ARM cross-compilation tool chain
Installs tools!!!, CUDA, libraries,...
Flashes devkit with OS Image
Reference documentation and samples
Compiles code samples, pushes them to devkit
And Runs a sample...as proof
WORLD-CLASS DEVELOPER TOOLS

Build
Debug
Profile

Getting started

Libraries

IDE integration

Standalone tools

Hardware support
CPU and GPU debugging & profiling

JetPack

OpenGL, CUDA, OpenGLES, Android, GNU C++

OpenCV, NVIDIA VISIONWORKS, cuDNN, NVTX, NVIDIA Tools eXtension

Visual Studio, eclipse

Windows, Linux, macOS

CPU and GPU debugging
THANKS! QUESTIONS?

Learn more
https://developer.nvidia.com/embedded/develop/tools

Download Jetpack
https://developer.nvidia.com/embedded/jetpack

Contact us by email or forums
devtools-support@nvidia.com