CUDA Development for Jetson with NVIDIA Nsight Eclipse Edition

Share: (https://devblogs.nvidia.com/parallelforall/cuda-jetson-nvidia-nsight-eclipse-edition/)

Posted on **March 19, 2017 (https://devblogs.nvidia.com/parallelforall/cuda-jetson-nvidia-nsight-eclipse-edition/)** by **Kudbudeen Jalaludeen (https://devblogs.nvidia.com/parallelforall/author/kjalaludeen/)** and **Satish Salian (https://devblogs.nvidia.com/parallelforall/author/ssalian/)** | **9 Comments (https://devblogs.nvidia.com/parallelforall/cuda-jetson-nvidia-nsight-eclipse-edition/#disqus_thread)**
Tagged **CUDA (https://devblogs.nvidia.com/parallelforall/tag/cuda/)**, **Debugging (https://devblogs.nvidia.com/parallelforall/tag/debugging/)**, **Jetson (https://devblogs.nvidia.com/parallelforall/tag/jetson/)**, **Jetson TX2 (https://devblogs.nvidia.com/parallelforall/tag/jetson-tx2/)**, **NSight Eclipse Edition (https://devblogs.nvidia.com/parallelforall/tag/nsight-eclipse-edition/)**, **Profiling (https://devblogs.nvidia.com/parallelforall/tag/profiling/)**, **Tools (https://devblogs.nvidia.com/parallelforall/tag/tools/)**

Figure 1: NVIDIA Jetson TX2 Developer Kit.

NVIDIA Nsight Eclipse Edition is a full-featured, integrated development environment that lets you easily develop CUDA applications for either your local (x86) system or a remote (x86 or ARM) target. In this post, I will walk you through the process of remote-developing CUDA applications for the NVIDIA Jetson TX2 (https://devblogs.nvidia.com/parallelforall/jetson-tx2-delivers-twice-intelligence-edge/), an ARM-based development kit. Note that this how-to also applies to Jetson TX1 (https://devblogs.nvidia.com/parallelforall/nvidia-jetson-tx1-supercomputer-on-module-drives-next-wave-of-autonomous-machines/) and Jetson TK1 (https://devblogs.nvidia.com/parallelforall/jetson-tk1-mobile-embedded-supercomputer-cuda-everywhere/) (this post is an update to an older post aimed at Jetson TK1 (https://devblogs.nvidia.com/parallelforall/nvidia-nsight-eclipse-edition-for-jetson-tk1/)).

Nsight Eclipse Edition supports two remote development modes: cross-compilation and "synchronize projects" mode. Cross-compiling for ARM on your x86 host system requires that all of the ARM libraries with which you will link your application be present on your host system. In synchronize-projects mode, on the other hand, your source code is synchronized between host and target systems and compiled and linked directly on the remote target, which has the advantage that all your libraries get resolved on the target system and need not be present on the host. Neither of these remote development modes requires an NVIDIA GPU to be present in your host system.

Note: CUDA cross-compilation tools for ARM are available in the Ubuntu installer packages of the CUDA Toolkit (https://developer.nvidia.com/cuda-downloads). If your host system is running a Linux distribution other than Ubuntu, I recommend the synchronize-projects remote development mode.

CUDA Toolkit Setup

The first step involved in cross-compilation is installing the CUDA Toolkit on your host system. You need to install the CUDA 8.0 Toolkit for Jetson TX2 or TX1, or the 6.5 Toolkit for Jetson TK1. You can download the CUDA Toolkit installer for Ubuntu from the CUDA Downloads page (https://developer.nvidia.com/cuda-downloads), or using Jetpack L4T (http://docs.nvidia.com/jetpack-l4t/index.html#developertools/mobile/jetpack/l4t/2.3/jetpack_l4t_install.htm).

I recommend that you use use Jetpack L4T (http://docs.nvidia.com/jetpack-l4t/index.html#developertools/mobile/jetpack/l4t/2.3/jetpack_l4t_install.htm) to install the same version of the CUDA toolkit for both the host and target systems. Once the toolkits are installed on both host and target, download the required Ubuntu DEB package from the CUDA download page (http://developer.nvidia.com/cuda-downloads) to do the cross-platform setup. You can find installation instructions in the CUDA Installation Guide (http://docs.nvidia.com/cuda/cuda-installation-guide-linux/#ubuntu-installation). I will summarize them below.

CUDA CROSS-PLATFORM ENVIRONMENT SETUP

First let's get your host system set up for cross-platform CUDA development. The following target architectures are supported for cross compilation.

- `x86_64`: 64-bit x86 CPU architecture;
- `armhf`: 32-bit ARM CPU architecture, as found on Jetson TK1;
- `aarch64`: 64-bit ARM CPU architecture, found on Jetson TX1 and TX2 and certain Android systems;
- `ppc64le`: 64-bit little-endian IBM POWER8 architecture

1. Enable the foreign architecture. The foreign architecture must be enabled to install the cross-platform toolkit. To enable `armhf`, `arm64 (aarch64), or ppc64le` as a foreign architecture, execute the following commands.

```
# For Jetson TX1/TX2:
$ sudo dpkg --add-architecture arm64
$ sudo apt-get update

# For Jetson TK1: $ sudo dpkg --add-architecture arm64
$ sudo dpkg --add-architecture armhf
$ sudo apt-get update
```

2. Run dpkg to install and update the repo meta-data:

For Jetson TX1/TX2, use CUDA 8:

```
$ sudo dpkg -i cuda-repo-ubuntu1404-8-0-local-ga2_8.0.61-1_amd64.deb
$ sudo apt-get update
```

For Jetson TK1, use CUDA 6.5:

```
$ sudo dpkg – i cuda-repo-ubuntu1404_6.5-37_amd64.deb
$ sudo apt-get update
```

3. Install cuda cross-platform packages. On supported platforms, the `cuda-cross-armhf`, `cuda-cross-aarch64`, and `cuda-cross-ppc64el` packages install all the packages required for cross-platform development to ARMv7, ARMv8, and POWER8, respectively. The libraries and header files of the target architecture's display driver package are also installed to enable the cross compilation of driver applications.

```
$ sudo apt-get install cuda-cross-<arch>
```

For Jetson TK1:

```
$ sudo apt-get install g++-4.6-arm-linux-gnueabihf
```

For Jetson TX1/TX2:

```
$ sudo apt-get install g++-4.8-aarch64-linux-gnu
```

4. OPTIONAL – if you also wish to do native x86 CUDA development and have an NVIDIA GPU in your host system then you can install the full toolchain and driver:

```
$ sudo apt-get install cuda
```

Reboot your system if you installed the driver so that the NVIDIA driver gets loaded. Then update paths to the toolkit install location as follows based on the CUDA Toolkit version installed:

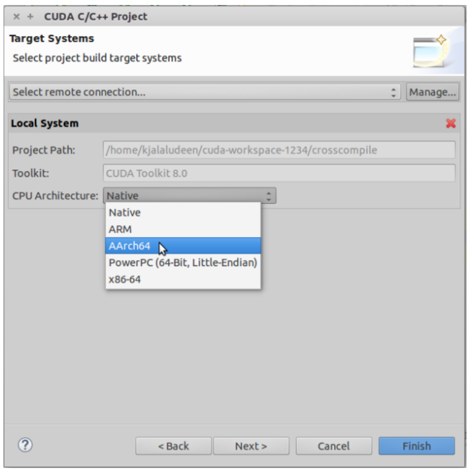
```
$ export PATH=/usr/local/cuda-8.0/bin:$PATH
$ export LD_LIBRARY_PATH=/usr/local/cuda-8.0/lib64:$LD_LIBRARY_PATH
```

At the end of these steps you should see `aarch64-linux` or `armv7-linux-gnueabihf` and the optional `x86_64_linux` folder under /usr/local/cuda-8.0/targets/.

For your cross-development needs, Jetson comes prepopulated with Linux for Tegra (L4T), a modified Ubuntu Linux distribution provided by NVIDIA. NVIDIA provides the board support package and a software stack that includes the CUDA Toolkit and OpenGL drivers. You can download all of these, as well as examples and documentation, from the JetPack page (https://developer.nvidia.com/embedded/jetpack).

Importing a CUDA Sample into Nsight Eclipse Edition

With the CUDA Toolkit installed and the paths setup on the host system, launch Nsight by typing “nsight” (without the quotes) at the command line or by finding the Nsight icon in the Ubuntu dashboard. Once Nsight is loaded, navigate to “File->New->CUDA C/C++ Project” and import an existing CUDA sample to start the Project Creation wizard. For the project name, enter “boxfilter-arm” and select “Import CUDA Sample” in the project type and and select the supported “CUDA Toolkit” in the toolchain. Next, choose the Boxfilter sample which can be found under the Imaging category.



Target Systems
Select project build target systems

Local System
Project Path: /home/kjalaludeen/cuda-workspace-1234/crosscompile
Toolkit: CUDA Toolkit 8.0
CPU Architecture: Native

(https://devblogs.nvidia.com/parallelforall/wp-content/uploads/2017/03/image1.png)

> Figure 2: Selecting Target CPU Architecture in NSight Eclipse Edition.

The remaining options in the wizard let you choose which GPU and CPU architectures to generate code for. First, choose the GPU code that should be generated by the nvcc compiler.

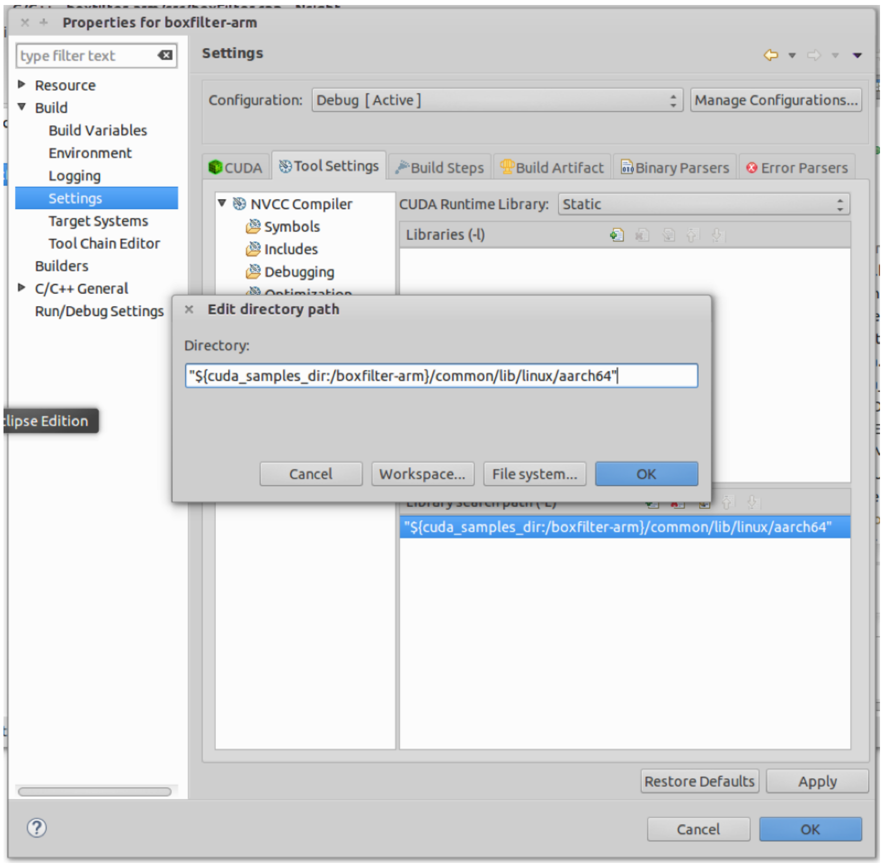
- For Jetson TK1 (NVIDIA Kepler GPU), choose 3.x GPU code and 3.x PTX code.
- For Jetson TX1 (NVIDIA Maxwell GPU), choose 5.x GPU code and 5.x PTX code.
- For Jetson TX2 (NVIDIA Pascal GPU), choose 6.x GPU code and 6.x PTX code.

The next page in the wizard lets you decide if you wish to do native x86 development or cross-compile for an ARM system. Choose “AArch64” for Jetson TX1/TX2 and “ARM” architecture for TK1 in the CPU architecture drop-down box.

Building Your Jetson Application with Nsight Eclipse Edition

CUDA samples can be imported and run on various hardware configurations. For this cross-build exercise you need to resolve the ARM library dependencies used by this application. Here’s how.

1. Right click on the project and navigate to Properties->Build->Settings->Tool Settings->NVCC Linker->Libraries and edit and enter the “Library search path(-L)” to point to `linux/aarch64` (for TX1/TX2) or `linux/armv7l` (for TK1) instead of `linux/x86_64`. This will resolve the `libGLEW` library dependencies. Also remove the entry for `GLU` if present since that library is unused (only for Nsight v6.5).



Configuration: Debug [Active]

CUDA Runtime Library: Static

Directory: "${cuda_samples_dir:/boxfilter-arm}/common/lib/linux/aarch64"

"${cuda_samples_dir:/boxfilter-arm}/common/lib/linux/aarch64"

(https://devblogs.nvidia.com/parallelforall/wp-content/uploads/2017/03/sample_path.png)

1. Click on the Miscellaneous tab and add a new `-Xlinker` option `—unresolved-symbols=ignore-in-shared-libs`. Also select the option “Link with OpenGL libraries” if present.

2. In the terminal window use the scp utility (http://en.wikipedia.org/wiki/Secure_copy) to copy the remaining libraries.

For Jetson TK1:

```
$ sudo scp ubuntu@your.ip.address:/usr/lib/arm-linux-gnueabihf/libglut.so.3 /usr/arm-linux-gnueabihf/lib
$ sudo scp ubuntu@your.ip.address:/usr/lib/arm-linux-gnueabihf/tegra/libGL.so.1 /usr/arm-linux/gnueabihf/lib
$ sudo scp ubuntu@your.ip.address:/usr/lib/arm-linux-gnueabihf/libX11.so.6 /usr/arm-linux-gnueabihf/lib

$ sudo ln -s /usr/arm-linux-gnueabihf/lib/libglut.so.3 /usr/arm-linux-gnueabihf/lib/libglut.so
$ sudo ln -s /usr/arm-linux-gnueabihf/lib/libgl.so.1 /usr/arm-linux-gnueabihf/lib/libgl.so
$ sudo ln -s /usr/arm-linux-gnueabihf/lib/libX11.so.6 /usr/arm-linux-gnueabihf/lib/libX11.so
```

For Jetson TX1/TX2:

```
$ sudo scp ubuntu@your.ip.address:/usr/lib/aarch64-linux-gnu/libglut.so.3  /usr/aarch64-linux-gnu/lib
$ sudo scp ubuntu@your.ip.address:/usr/lib/aarch64-linux-gnu/tegra/libGL.so.1 /usr/aarch64-linux-gnu/lib
$ sudo scp ubuntu@your.ip.address:/usr/lib/aarch64-linux-gnu/libX11.so.6 /usr/aarch64-linux-gnu/lib
$ sudo scp  ubuntu@your.ip.address:/usr/lib/aarch64-linux-gnu/libGLU.so.1  /usr/aarch64-linux-gnu/lib
$ sudo ln -s /usr/aarch64-linux-gnu/lib/libglut.so.3 /usr/aarch64-linux-gnu/lib/libglut.so
$ sudo ln -s /usr/aarch64-linux-gnu/lib/libGL.so.1 /usr/aarch64-linux-gnu/lib/libGL.so
$ sudo ln -s /usr/aarch64-linux-gnu/lib/libX11.so.6 /usr/aarch64-linux-gnu/lib/libX11.so
$ sudo ln -s /usr/aarch64-linux-gnu/lib/libGLU.so.1 /usr/aarch64-linux-gnu/lib/libGLU.so
```

Note: You need to copy these ARM libraries only for the first CUDA sample. You may need additional libraries for other samples.

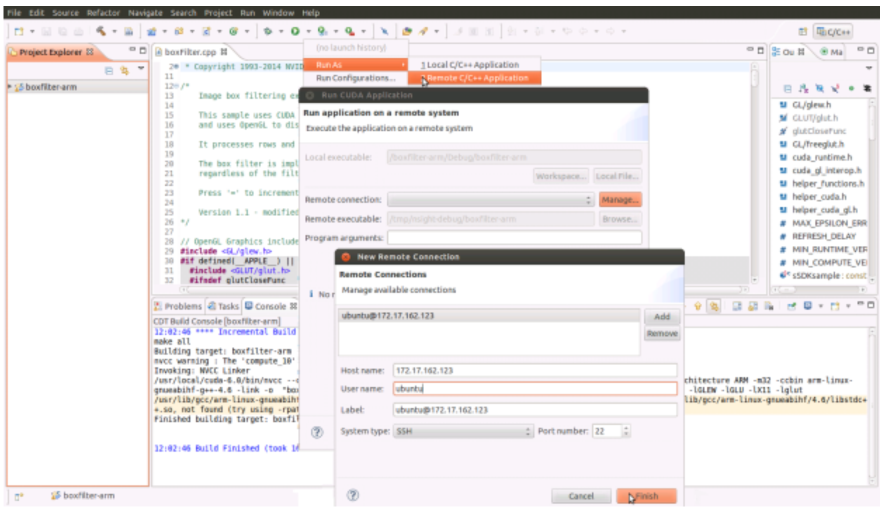
The build process for ARM cross-development is similar to the local build process. Just click on the build “hammer” icon in the toolbar menu to build a debug ARM binary. As part of the compilation process, Nsight will launch nvcc for the GPU code and the cross-compiler for the CPU code as follows:

```
Building file: ../src/boxFilter_kernel.cu
Invoking: NVCC Compiler
/usr/local/cuda-6.0/bin/nvcc -I"/usr/local/cuda-6.0/samples/3_Imaging" -I"/usr/local/cuda-6.0/samples/common/
-I"/home/satish/cuda-workspace_new/boxfilter-arm" -G -g -O0 -ccbin arm-linux-gnueabihf-g++-4.6 -gencode arch=
code=sm_30 -gencode arch=compute_32,code=sm_32 --target-cpu-architecture ARM -m32 -odir "src" -M -o "src/boxF
"../src/boxFilter_kernel.cu"
/usr/local/cuda-6.0/bin/nvcc --compile -G -I"/usr/local/cuda-6.0/samples/3_Imaging" -I"/usr/local/cuda-6.0/sa
-I"/home/satish/cuda-workspace_new/boxfilter-arm" -O0 -g -gencode arch=compute_30,code=compute_30 -gencode ar
code=sm_32 --target-cpu-architecture ARM -m32 -ccbin arm-linux-gnueabihf-g++-4.6  -x cu -o  "src/boxFilter_ke
"../src/boxFilter_kernel.cu"
Finished building: ../src/boxFilter_kernel.cu
```

After the compilation steps, the linker resolves all library references, giving you a ready-to-run `boxfilter-arm` binary.

Running Your Jetson Application from Nsight Eclipse Edition

To run the code on the target Jetson system, click on Run As->Remote C/C++ Application to setup the target system user and host address.



Run CUDA Application

Run application on a remote system

Execute the application on a remote system

Local executable: /boxfilter-arm/Debug/boxfilter-arm

Remote connection:

Remote executable: /tmp/nsight-debug/boxfilter-arm

Program arguments:

New Remote Connection

Remote Connections

Manage available connections

ubuntu@172.17.162.123

Host name: 172.17.162.123

User name: ubuntu

Label: ubuntu@172.17.162.123

System type: SSH Port number: 22

```
CDT Build Console [boxfilter-arm]
12:02:46 **** Incremental Build
make all
Building target: boxfilter-arm
nvcc warning : The 'compute_10'
Invoking: NVCC Linker
/usr/local/cuda-6.0/bin/nvcc --
gnueabihf-g++-4.6 -link -o  "box
/usr/lib/gcc/arm-linux-gnueabihf
+.so, not found (try using -rpat
Finished building target: boxfil

12:02:46 Build Finished (took 1
```

(http://devblogs.nvidia.com/parallelforall/wp-content/uploads/2014/05/nsight_remote_run.png)

Once you finish the remote target system configuration setup, click on the Run icon and you will see a new entry to run the `boxfilter-arm` binary on the Jetson.

Note: the box filter application relies on data files that reside in the `data/` subfolder of the application, which you will need to copy to the target system. Use the `scp` utility to copy those files into the `/tmp/nsight-debug/data/` folder on your Jetson.

Next, edit the `boxfilter.cpp` file as follows:

1. To ensure that the application runs on the correct display device, add this line to the top of the main function if not already there. This is not required on CUDA toolkit 8.0 samples.

```
setenv("DISPLAY", ":0", 0);
```

1. Add the following lines to the top of the display function so that the app auto-terminates after a few seconds. This is required to gather deterministic execution data across multiple runs of the application, which you will need later in the profiling section:

```
static int icnt = 120;
while(!icnt--)
{
    cudaDeviceReset();
    exit(EXIT_SUCCESS);
}
```

3. Also comment out the call to `freeTextures()` in the `cleanup()` function since it might cause runtime error.

Click on Run to execute the modified Box Filter application on your Jetson.

Debugging Your Jetson Application in Nsight Eclipse Edition

The remote target system configuration that you set up in Nsight earlier will also be visible under the debugger icon in the toolbar.

Before you launch the debugger, note that by default Jetson doesn't allow any application to solely occupy the GPU 100% of the time. In order to run the debugger on Jetson TK1, you need to fix this using the following command (note: this is not required on Jetson TX1/TX2).

```
root@tegra-ubuntu:/home/ubuntu# sudo echo N > sys/kernel/debug/gk20a.0/timeouts_enabled
```

Now you can launch the debugger using the debug icon back on the host system. Nsight will switch to its debugger perspective and break on the first instruction in the CPU code. You can single-step a bit there to see the execution on the CPU and watch the variables and registers as they are updated.

To break on any CUDA kernel executing on the GPU, , go to the CUDA view in the top-right pane of Nsight and click on the cube icon dropdown. Then select the "break on application kernel launches" feature to break on the first instruction of a CUDA kernel launch. You can now resume the application, which will run until the first breakpoint is hit in the CUDA kernel. From here, you can browse the CPU and GPU call stack in the top-left pane. You can also view the variables, registers and HW state in the top-right pane. In addition, you can see that the Jetson GPU is executing 16 blocks of 64 threads each running on its Streaming Multiprocessors (SMs).

You can also switch to disassembly view and watch the register values being updated by clicking on the **i->** icon in the debug view to do GPU instruction-level single-stepping.

(http://devblogs.nvidia.com/parallelforall/wp-content/uploads/2014/05/nsight_debug_view.png)

To "pin" (focus on) specific GPU threads, double click the thread(s) of interest in the CUDA tab in the top-right pane. The pinned CUDA threads will appear in the top-left pane, allowing you to select and single-step just those threads. (Keep in mind, however, that single-stepping a given thread causes the remaining threads of the same warp (http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#differences-between-host-and-device) to step as well, since they share a program counter.) You can experiment and watch this by pinning threads that belong to different warps.

There are more useful debug features that you will find by going into the debug configuration settings from the debug icon drop down, such as enabling cuda-memcheck and attaching to a running process (on the host system only).

To quit the application you are debugging, click the red stop button in the debugger perspective.

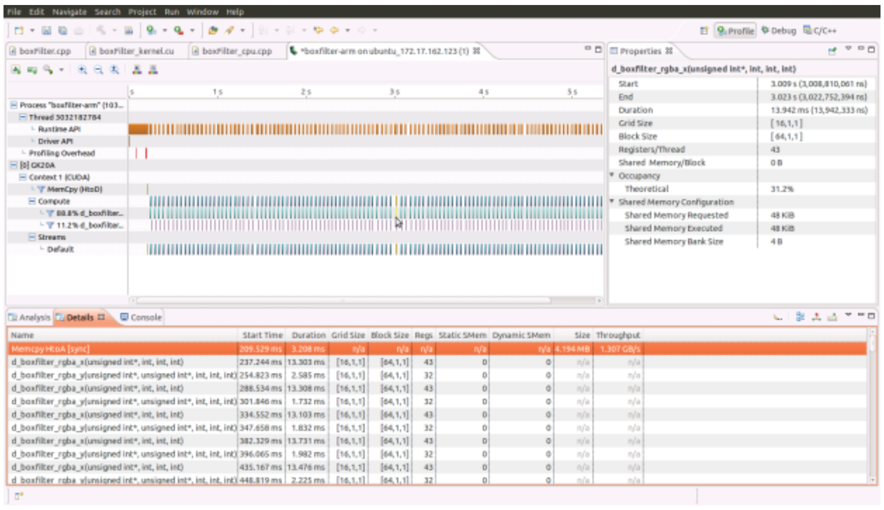
Profiling Your Jetson Application in Nsight Eclipse Edition

Switch back to the C++ project editor view to start the profiler run. The remote target system configuration you set up in Nsight earlier will also be visible to you under the profiler icon in the toolbar.

Before you launch the profiler, note that you need to create a release build with `-lineinfo` included in the compile options. This tells the compiler to generate information on source-to-instruction correlation. To do this, first go to the project settings by right-clicking on the project in the left pane. Then navigate to Properties->Build->Settings->Tool Settings->Debugging and check the box that says "Generate line-number…" and click Apply.

Back in the main window, click on the build hammer dropdown menu to create a release build. Resolve any build issues as you did during the first run above, then click on the Run As->Remote C/C++ Application to run the release build of the application. At this point Nsight will overwrite the Jetson TK1 system with the release binary you want to profile and run it once.

Next click on the profile icon dropdown and choose Profile Configurations where you must select "Profile Remote Application" since the binary is already on the Jetson TK1. Nsight will then switch you to the profiler perspective while it runs the application to gather an execution timeline view of all the CUDA Runtime and Driver API calls and of the kernels that executed on the GPU. The properties tab displays details of any event you select from this timeline; the details of the events can also be viewed in text form in the Details tab in the lower pane.



d_boxfilter_rgba_x(unsigned int*, int, int, int)

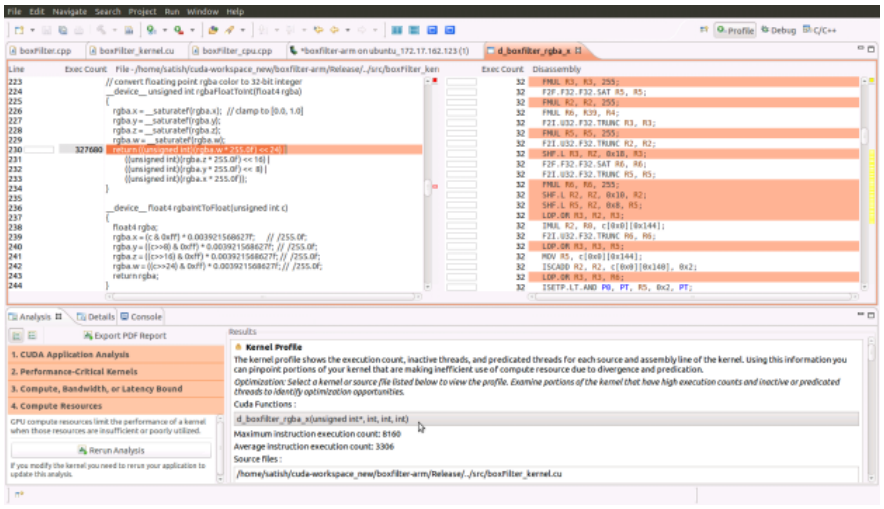
Start	3.009 s (3,008,810,061 ns)
End	3.023 s (3,022,752,394 ns)
Duration	13.942 ms (13,942,333 ns)
Grid Size	[16,1,1]
Block Size	[64,1,1]
Registers/Thread	43
Shared Memory/Block	0 B
Occupancy	
Theoretical	31.2%
Shared Memory Configuration	
Shared Memory Requested	48 KiB
Shared Memory Executed	48 KiB
Shared Memory Bank Size	4 B

Name	Start Time	Duration	Grid Size	Block Size	Regs	Static SMem	Dynamic SMem	Size	Throughput
Memcpy HtoA [sync]	209.529 ms	3.208 ms	n/a	n/a	n/a	n/a	n/a	4.194 MB	1.307 GB/s
d_boxfilter_rgba_x(unsigned int*, int, int, int)	237.244 ms	13.303 ms	[16,1,1]	[64,1,1]	43	0	0	n/a	n/a
d_boxfilter_rgba_y(unsigned int*, unsigned int*, int, int, int)	254.823 ms	2.585 ms	[16,1,1]	[64,1,1]	32	0	0	n/a	n/a
d_boxfilter_rgba_x(unsigned int*, int, int, int)	288.534 ms	13.308 ms	[16,1,1]	[64,1,1]	43	0	0	n/a	n/a
d_boxfilter_rgba_y(unsigned int*, unsigned int*, int, int, int)	301.846 ms	1.732 ms	[16,1,1]	[64,1,1]	32	0	0	n/a	n/a
d_boxfilter_rgba_x(unsigned int*, int, int, int)	334.552 ms	13.103 ms	[16,1,1]	[64,1,1]	43	0	0	n/a	n/a
d_boxfilter_rgba_y(unsigned int*, unsigned int*, int, int, int)	347.658 ms	1.832 ms	[16,1,1]	[64,1,1]	32	0	0	n/a	n/a
d_boxfilter_rgba_x(unsigned int*, int, int, int)	382.329 ms	13.731 ms	[16,1,1]	[64,1,1]	43	0	0	n/a	n/a
d_boxfilter_rgba_y(unsigned int*, unsigned int*, int, int, int)	396.065 ms	1.982 ms	[16,1,1]	[64,1,1]	32	0	0	n/a	n/a
d_boxfilter_rgba_x(unsigned int*, int, int, int)	435.167 ms	13.476 ms	[16,1,1]	[64,1,1]	43	0	0	n/a	n/a
d_boxfilter_rgba_y(unsigned int*, unsigned int*, int, int, int)	448.819 ms	2.225 ms	[16,1,1]	[64,1,1]	32	0	0	n/a	n/a

(http://devblogs.nvidia.com/parallelforall/wp-content/uploads/2014/05/nsight_profile_view.png)

Below the timeline view in the lower pane, there is also an Analysis tab that is very useful for performance tuning. It guides you through a step-by-step approach (http://devblogs.nvidia.com/parallelforall/cudacasts-episode-19-cuda-6-guided-performance-analysis-visual-profiler/) on resolving performance bottlenecks in your application. You can switch between guided and unguided analysis by clicking on their icons under the Analysis tab.

You can also get a source-to-instruction correlation view, with hot spots (where the instructions-executed count was particularly high) identified in red as the figure below shows. You get this view from within the guided analysis mode by first clicking on “Examine Individual Kernels” and selecting the highest ranked (100) kernel from the list of examined kernels, then clicking “Perform Kernel Analysis” followed by “Perform Compute Analysis.” From there, clicking “Show Kernel Profile” shows the `d_boxfilter_rgba_a` kernel in the right pane. Double-click on the kernel name to see the source-to-instruction view. Clicking on a given line of source code highlights the corresponding GPU instructions.



```
Line  Exec Count  File - /home/satish/cuda-workspace_new/boxfilter-arm/Release/../src/boxFilter_ker
223               // convert floating point rgba color to 32-bit integer
224               __device__ unsigned int rgbaFloatToInt(float4 rgba)
225               {
226                 rgba.x = __saturatef(rgba.x);  // clamp to [0.0, 1.0]
227                 rgba.y = __saturatef(rgba.y);
228                 rgba.z = __saturatef(rgba.z);
229                 rgba.w = __saturatef(rgba.w);
230   327680        return ((unsigned int)(rgba.w * 255.0f) << 24) |
231                   ((unsigned int)(rgba.z * 255.0f) << 16) |
232                   ((unsigned int)(rgba.y * 255.0f) << 8) |
233                   ((unsigned int)(rgba.x * 255.0f));
234               }
235
236               __device__ float4 rgbaIntToFloat(unsigned int c)
237               {
238                 float4 rgba;
239                 rgba.x = (c & 0xff) * 0.003921568627f;      // /255.0f;
240                 rgba.y = ((c>>8) & 0xff) * 0.003921568627f;  // /255.0f;
241                 rgba.z = ((c>>16) & 0xff) * 0.003921568627f; // /255.0f;
242                 rgba.w = ((c>>24) & 0xff) * 0.003921568627f; // /255.0f;
243                 return rgba;
244               }
```

```
Exec Count  Disassembly
32  FMUL R3, R3, 255;
32  F2F.F32.F32.SAT R5, R5;
32  FMUL R2, R2, 255;
32  FMUL R6, R39, R4;
32  F2I.U32.F32.TRUNC R3, R3;
32  FMUL R5, R5, 255;
32  F2I.U32.F32.TRUNC R2, R2;
32  SHF.L R3, R2, 0x18, R3;
32  F2F.F32.F32.SAT R6, R6;
32  F2I.U32.F32.TRUNC R5, R5;
32  FMUL R6, R6, 255;
32  SHF.L R2, R2, 0x10, R2;
32  SHF.L R5, R2, 0x8, R5;
32  LOP.OR R3, R2, R3;
32  IMUL R2, R0, c[0x0][0x144];
32  F2I.U32.F32.TRUNC R6, R6;
32  LOP.OR R3, R3, R5;
32  MOV R5, c[0x0][0x144];
32  ISCADD R2, R2, c[0x0][0x140], 0x2;
32  LOP.OR R3, R3, R6;
32  ISETP.LT.AND P0, PT, R5, 0x2, PT;
```

1. CUDA Application Analysis
2. Performance-Critical Kernels
3. Compute, Bandwidth, or Latency Bound
4. Compute Resources

GPU compute resources limit the performance of a kernel when those resources are insufficient or poorly utilized.

If you modify the kernel you need to rerun your application to update this analysis.

Kernel Profile

The kernel profile shows the execution count, inactive threads, and predicated threads for each source and assembly line of the kernel. Using this information you can pinpoint portions of your kernel that are making inefficient use of compute resource due to divergence and predication.

Optimization: Select a kernel or source file listed below to view the profile. Examine portions of the kernel that have high execution counts and inactive or predicated threads to identify optimization opportunities.

Cuda Functions :

d_boxfilter_rgba_x(unsigned int*, int, int, int)

Maximum instruction execution count: 8160

Average instruction execution count: 3306

Source files :

/home/satish/cuda-workspace_new/boxfilter-arm/Release/../src/boxFilter_kernel.cu

(http://devblogs.nvidia.com/parallelforall/wp-content/uploads/2014/05/nsight_rc_to_sass.png)

As you can see, whether you are new to NVIDIA Nsight Eclipse Edition or an avid Nsight user, Nsight makes it just as easy and straightforward to create CUDA applications for the Jetson TK1 platform as for all your CUDA-enabled GPUs.

Get Started with CUDA Today

Start by downloading the CUDA Toolkit (https://developer.nvidia.com/cuda-toolkit). If you are new to CUDA, check out this tutorial to get you started with CUDA programming (https://devblogs.nvidia.com/parallelforall/even-easier-introduction-cuda/). To learn more about Jetson TX2, NVIDIA’s latest embedded developer platform, read “NVIDIA Jetson TX2 Delivers Twice the Intelligence to the Edge (https://devblogs.nvidia.com/parallelforall/jetson-tx2-delivers-twice-intelligence-edge/)“.

RELATED POSTS

NVIDIA Nsight Eclipse Edition for Jetson TK1 (https://devblogs.nvidia.com/parallelforall/nvidia-nsight-eclipse-edition-for-jetson-tk1/)

Remote application development using NVIDIA® Nsight™ Eclipse Edition (https://devblogs.nvidia.com/parallelforall/remote-application-development-nvidia-nsight-eclipse-edition/)

NVIDIA Jetson TX2 Delivers Twice the Intelligence to the Edge (https://devblogs.nvidia.com/parallelforall/jetson-tx2-delivers-twice-intelligence-edge/)

CUDACasts Episode 13: Clock, Power, and Thermal Profiling with Nsight Eclipse Edition (https://devblogs.nvidia.com/parallelforall/cudacasts-episode-13-clock-power-thermal-profiling-nsight-eclipse-edition/)

// ∀

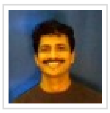
Share:

About Kudbudeen Jalaludeen

Kudbuc Jalaluc

Kudbudeen Jalaludeen is a senior software engineer at NVIDIA currently working on CUDA developer tools. He has a Bachelor's degree in Electronics and Communication Engineering and experience developing tools and IDEs based on eclipse and other UI frameworks.
View all posts by Kudbudeen Jalaludeen → (https://devblogs.nvidia.com/parallelforall/author/kjalaludeen/)

About Satish Salian



Satish Salian is a Software Technical Lead at NVIDIA currently involved with extending the VRWorks SDK. In the recent past Satish has been involved with building the SW stack of the world fastest Deep Learning Systems @ NVIDIA called the DGX-1 and DIGITS DEVBOX. Prior projects also include creating CUDA developer tools, display control UI tools and NVAPI SDKs at NVIDIA. He has a Bachelor's degree in Computer Engineering from University of Pune, India.
View all posts by Satish Salian → (https://devblogs.nvidia.com/parallelforall/author/ssalian/)

- Pingback: CUDA Development for Jetson with NVIDIA Nsight Eclipse Edition | PCHardwareNews (http://pchardwarenews.com/cuda-development-for-jetson-with-nvidia-nsight-eclipse-edition)()

- *Borislav Karaivanov*

  Is it true that if one has fully installed JetPack 3.0 on the host machine connected to the Jetson TX2, then the section “CUDA Cross-Platform Environment Setup” can be skipped (i.e., its content took place during the JetPack 3.0 installation)?

  - *Kudbudeen*

    Yes, cross compile libs and compiler will be installed before JetPack installing ‘Compile CUDA Samples’.

- *Ram Kumar Koppu*

  Hi, Run and Debug setup information in your blog is not enough to setup my TX2 dev env with my TX2 dev kit. Can you provide these steps in details with screen shots pl ease?

  - *Kudbudeen*

    Can you provide more details on which part you are having issues? The remote connection dialog should look the same as shown in the screenshot.

- *Dourado*

  I’m having an issue to run the example on TX1. I get the following error: “CUDA driver version is insufficient for CUDA runtime version”. My dev machine has Cuda 8.0.61 while the TX1 has 8.0.33. I checked my GPU and PTX for 5.3 generation and still the same problem. I can compile and the samples normally directly on TX1, but the cross compile isnt working.
  Can you help me?

  https://uploads.disquscdn.com/images/4132471fca060413b1fa57b36375b84ed38322836a0797db536de46fdb3b6f1e.png (https://uploads.disquscdn.com/images/4132471fca060413b1fa57b36375b84ed38322836a0797db536de46fdb3b6f1e.png)

  - *Kudbudeen*

    You can have multiple versions of CUDA installed on your host machine. You don’t have to install the driver on the host machine when installing CUDA toolkit on host for Cross compiling. You can download Jetpack to install compatible host and target CUDA toolkits for TX1.

- *Nathan*

Hi I am having difficulty getting the boxFilter sample to display on the host machine. I have Cuda Toolkit 8.0 install and went through the entire walk through and everything works with compiling and running the example yet it does not display. I checked to make sure the line setenv(“DISPLAY”, “:0”, 0); is present in the main method in boxFilter.cpp and it is. If you could please help me I would greatly appreciate it.

- *Kudbudeen*

  Hi, The boxFilter sample will run and display on the target system(Jetson) if you have followed the instructions to setup the remote connection. You need to check it in the monitor attached to Jetson and not on the host machine. If you are already checking on the target system then please post the output from Nsight EE console view.