Debugging and Profiling

Dr. Alptekin Temizel

Introduction

- Parallel Nsight is a development environment for CUDA and graphics applications running on NVIDIA GPUs. It consists of:
  - CUDA Debugger
  - Graphics Debugger
  - Analysis and Profiling Tools

Introduction

- Supports Visual Studio.
- Eclipse version also available
- Also supports applications that use Direct3D 12 API
- Documentation available online:

http://docs.nvidia.com/gameworks/index.html#developertools/desktop/nvidia_nsight.htm

CUDA Debugger

- The CUDA Debugger helps you debug your CUDA applications.
  - You can set breakpoints in CUDA source code
  - inspect memory
  - view the values of local variables
  - perform memory checks
- You can use the CUDA Debugger with applications built with the CUDA Runtime (CUDART) API or with the CUDA Driver API.

Graphics Debugger

- The Graphics Debugger allows you to debug frame by each draw call. You can
  - debug vertex shaders
  - debug pixel shaders
  - view pipeline states

Analysis and Profiling Tools

- The Analysis tools help you to understand how workloads are distributed across your application.
- You can see -along a visual timeline-
  - API calls (CUDA C, OpenCL, DirectX, Microsoft DirectCompute, OpenGL, and Cg)
  - memory copies
  - kernel executions
  - CPU core and thread events
  - Custom user events
- You can gather and analyze kernel-level performance information

Target and Host Setup

Parallel Nsight allows you to debug your applications in two different ways:

- **Local** debugging, in which the host and target are on the same machine.
- **Remote** debugging, in which the host and target are on two different machines
  -

Target and Host Setup

- **Remote** debugging:
  - The **host** machine will run Visual Studio to build your project, as well as to launch debugging sessions.
  - A separate computer is configured as the **target** machine. The target will
    - run the Parallel Nsight Monitor. The monitor detects incoming requests from the host to execute an application on the target machine.
    - run the application to be debugged or analyzed. Parallel Nsight software manages the synchronization of files between the host and target machines.

Local vs. Remote Debugging

- V2.2 onwards: Local **CUDA debugging on a single GPU system**. Note: Local debugging might adversely impact the performance of your application when debugging certain types of applications (such as most graphics applications).

- In remote debugging, Visual Studio environment will continue to run on the host machine, even if the target machine has to be rebooted because of an application crash.

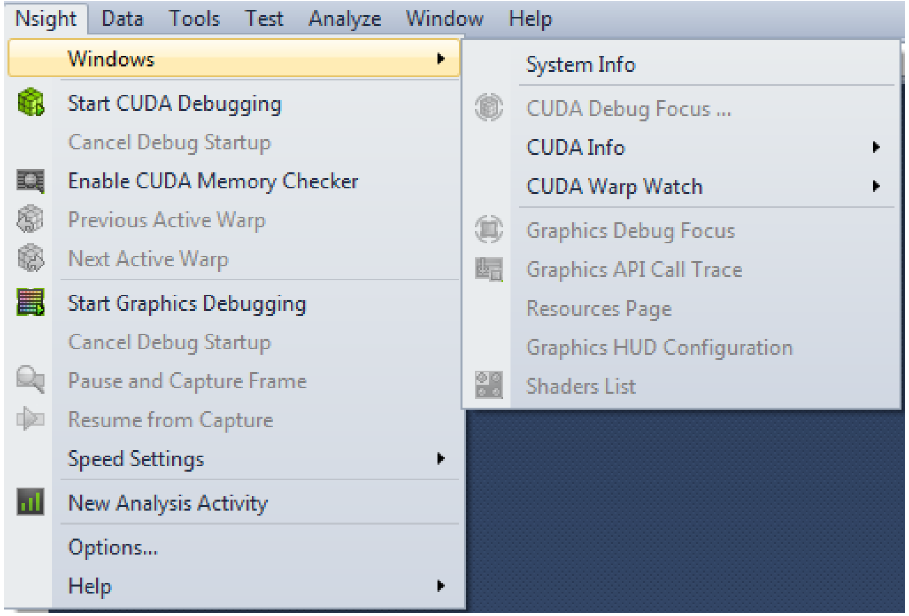
- Some components of Parallel Nsight, such as shader debugging, will not work at all with a local debugging configuration.

Target Setup

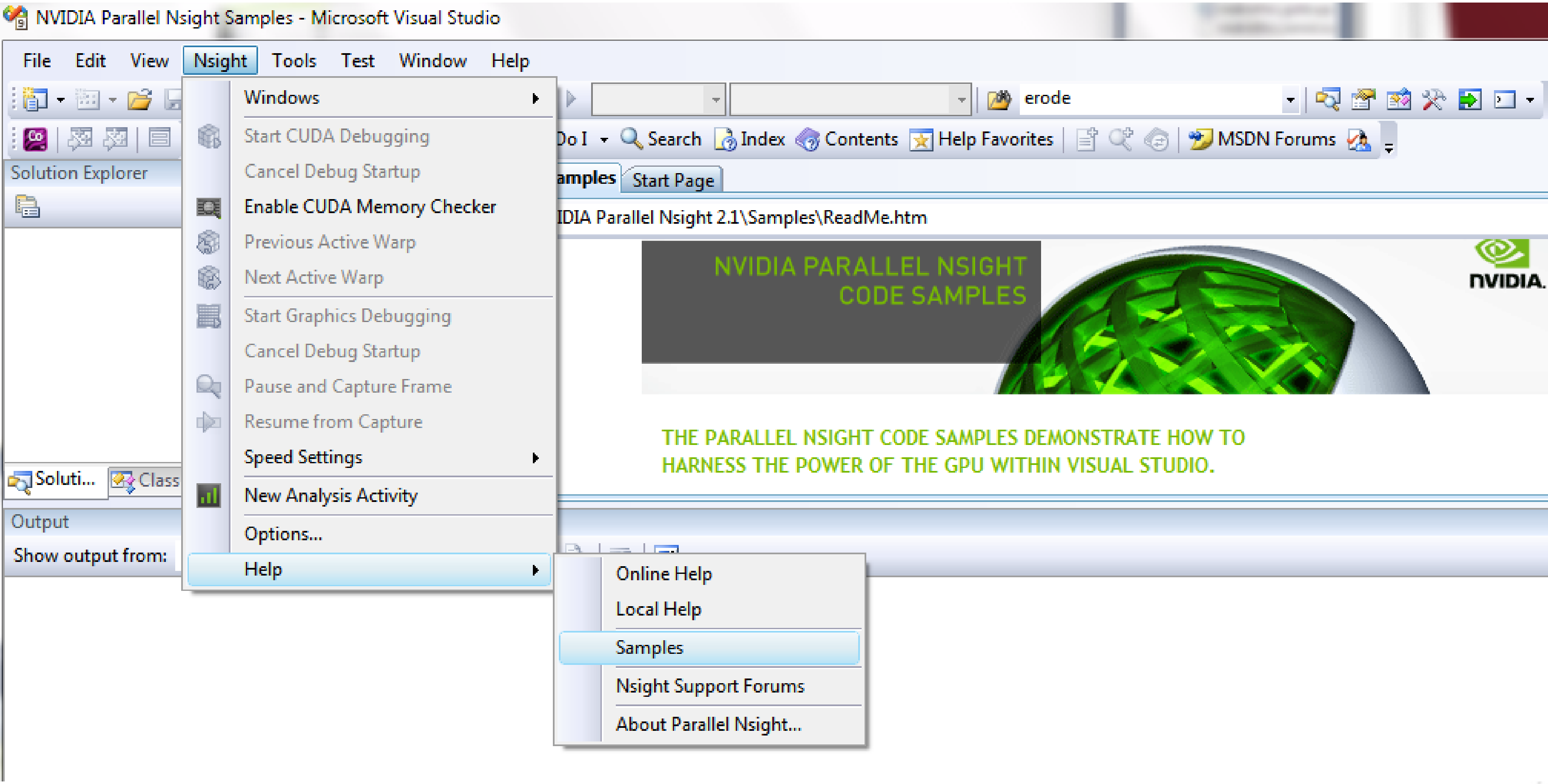
- The target computer does not need to be running Visual Studio.
- You need to install the **Parallel Nsight Monitor** on the target computer.
- See the guide for monitor setup.

Host Setup

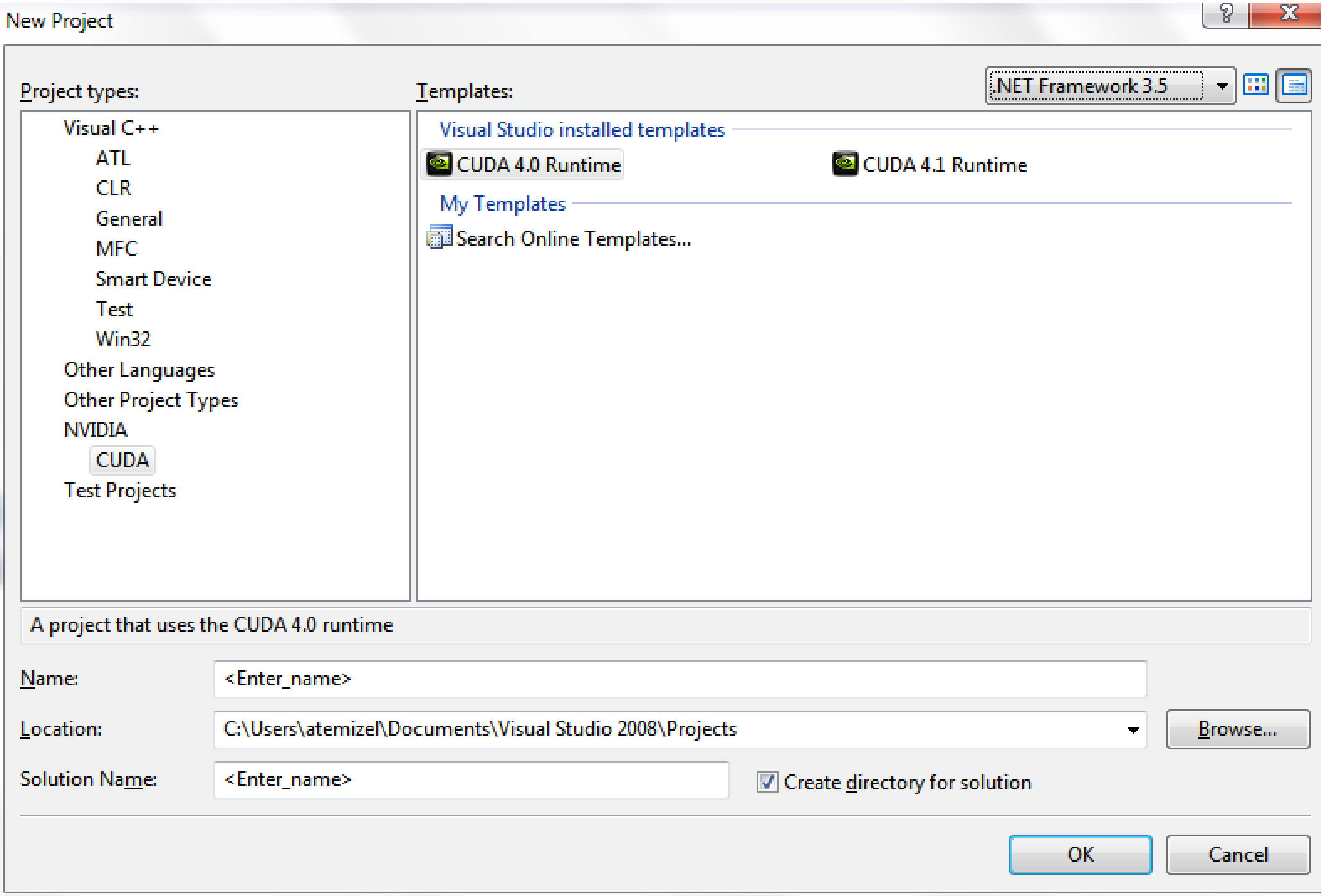
- **Host** is the computer that is running Visual Studio, where your application is being built and debugged.
- After you install Parallel Nsight on your computer, you will see a new menu called Nsight on your Visual Studio taskbar.
- See the documentation for installation.



Samples


NVIDIA Parallel Nsight Samples - Microsoft Visual Studio
File
Edit
View
Nsight
Tools
Test
Window
Help
Windows
Start CUDA Debugging
Cancel Debug Startup
Enable CUDA Memory Checker
Previous Active Warp
Next Active Warp
Start Graphics Debugging
Cancel Debug Startup
Pause and Capture Frame
Resume from Capture
Speed Settings
New Analysis Activity
Options...
Help
Online Help
Local Help
Samples
Nsight Support Forums
About Parallel Nsight...
erode
Search
Index
Contents
Help Favorites
MSDN Forums
Solution Explorer
Start Page
IDIA Parallel Nsight 2.1\Samples\ReadMe.htm
NVIDIA PARALLEL NSIGHT
CODE SAMPLES
NVIDIA.
THE PARALLEL NSIGHT CODE SAMPLES DEMONSTRATE HOW TO
HARNESS THE POWER OF THE GPU WITHIN VISUAL STUDIO.
Output
Show output from:

Templates

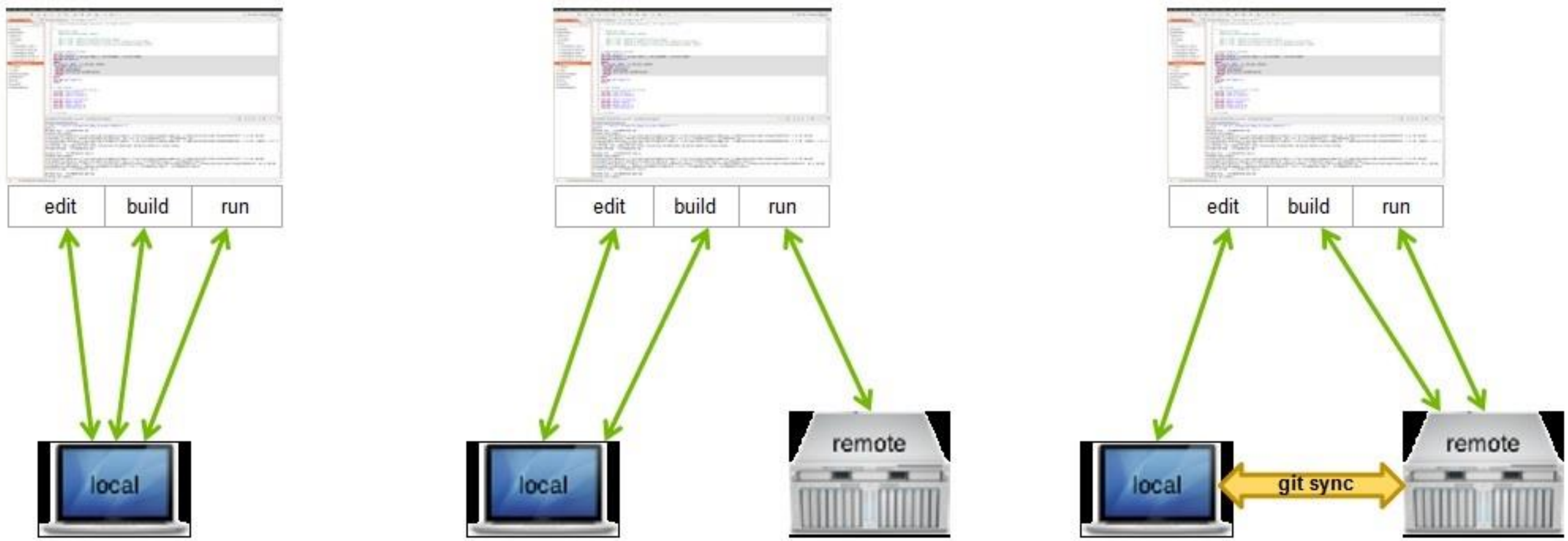

New Project
Project types:
Templates:
.NET Framework 3.5
Visual C++
ATL
CLR
General
MFC
Smart Device
Test
Win32
Other Languages
Other Project Types
NVIDIA
CUDA
Test Projects
Visual Studio installed templates
CUDA 4.0 Runtime
CUDA 4.1 Runtime
My Templates
Search Online Templates...
A project that uses the CUDA 4.0 runtime
Name:
<Enter_name>
Location:
C:\Users\atemizel\Documents\Visual Studio 2008\Projects
Browse...
Solution Name:
<Enter_name>
Create directory for solution
OK
Cancel

NVIDIA Nsight Eclipse Edition

NVIDIA Nsight Eclipse Edition is a full-featured, integrated development environment that lets you develop CUDA applications for either your local (x86) system or a remote (x86 or ARM) target.

ARM target is typically a Jetson board

NVIDIA Nsight Eclipse Edition

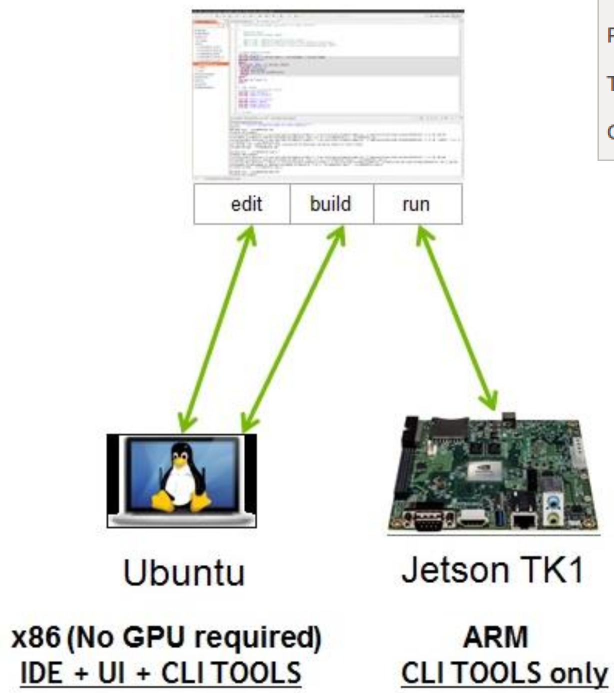


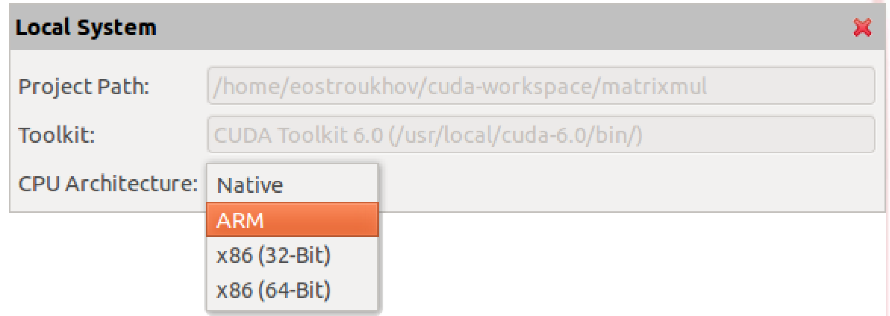
Create and run native builds on host

Cross compile on host and run on target

Sync projects to create remote builds on targets

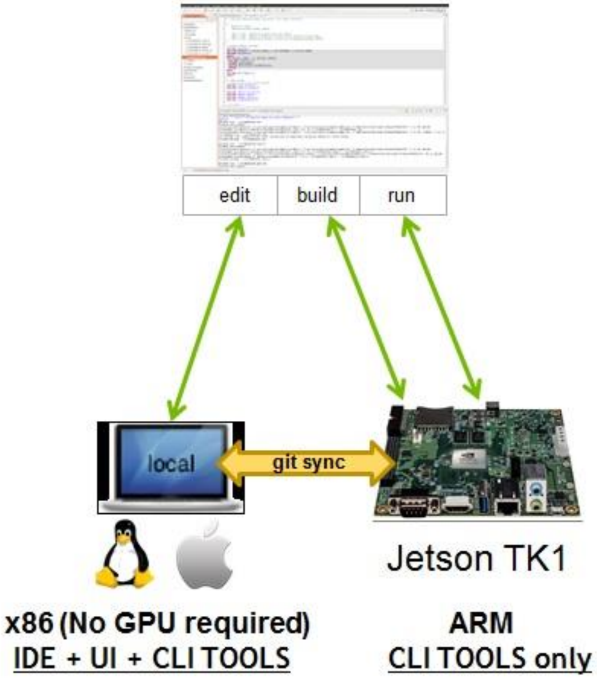
NVIDIA Nsight Eclipse Edition





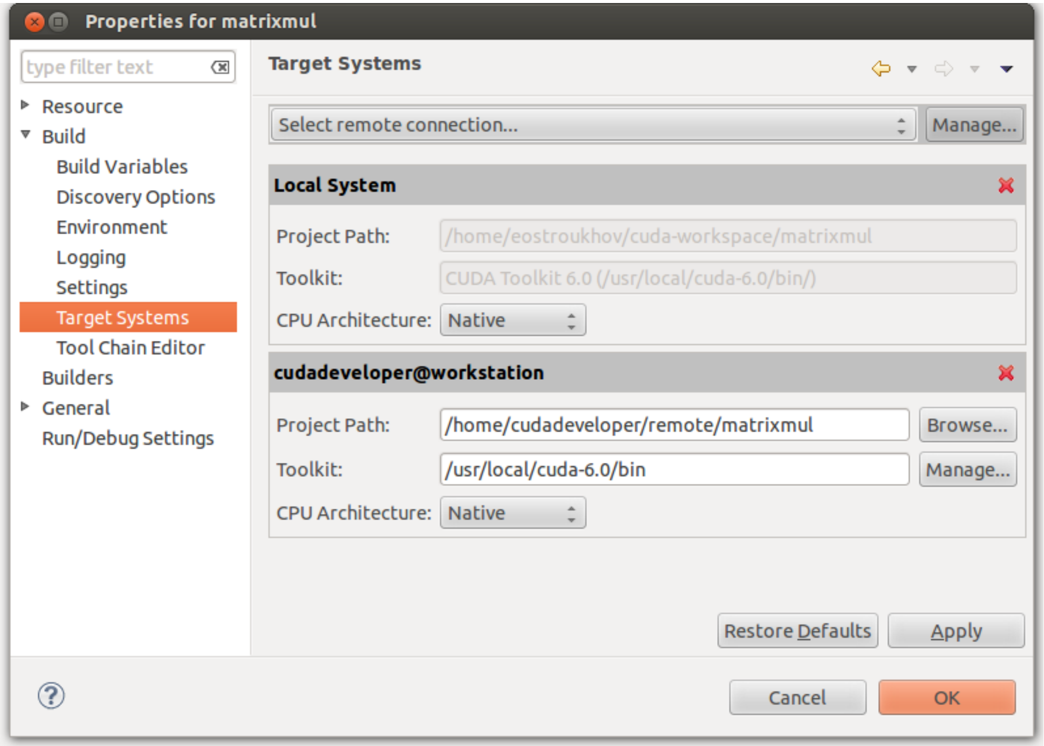
In the **cross compilation mode** the project resides on the host system and the cross compilation is also done on the host system. The cross compilation mode is only supported on an Ubuntu x86 host system.

NVIDIA Nsight Eclipse Edition



In the **remote synchronized project mode** the project resides on the host system and gets synchronized with the remote target system. The compilation gets done natively on the target system. The remote synchronized project mode is supported on Mac OSX, Linux x86 and Linux POWER systems.

NVIDIA Nsight Eclipse Edition



To create native remote build using remote synchronized project mode click on Manage... button to add a remote system, then select the project path, toolkit and the CPU architecture of the target system:
To synchronize projects between the host and target system, install and configure git on both the local and remote systems:
git config --global user.name <anyname>
git config --global user.email <anyemail>

NVIDIA Nsight Eclipse Edition

Nsight Eclipse Edition supports two remote development modes. Neither of these remote development modes requires an NVIDIA GPU to be present in your host system:

- 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.

- Synchronize-projects mode: 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.

NVIDIA Nsight Eclipse Edition

Jetpack L4T installs the same version of the CUDA toolkit for both the host and target systems.

Get your host system set up for cross-platform CUDA development: https://devblogs.nvidia.com/cuda-jetson-nvidia-nsight-eclipse-edition/

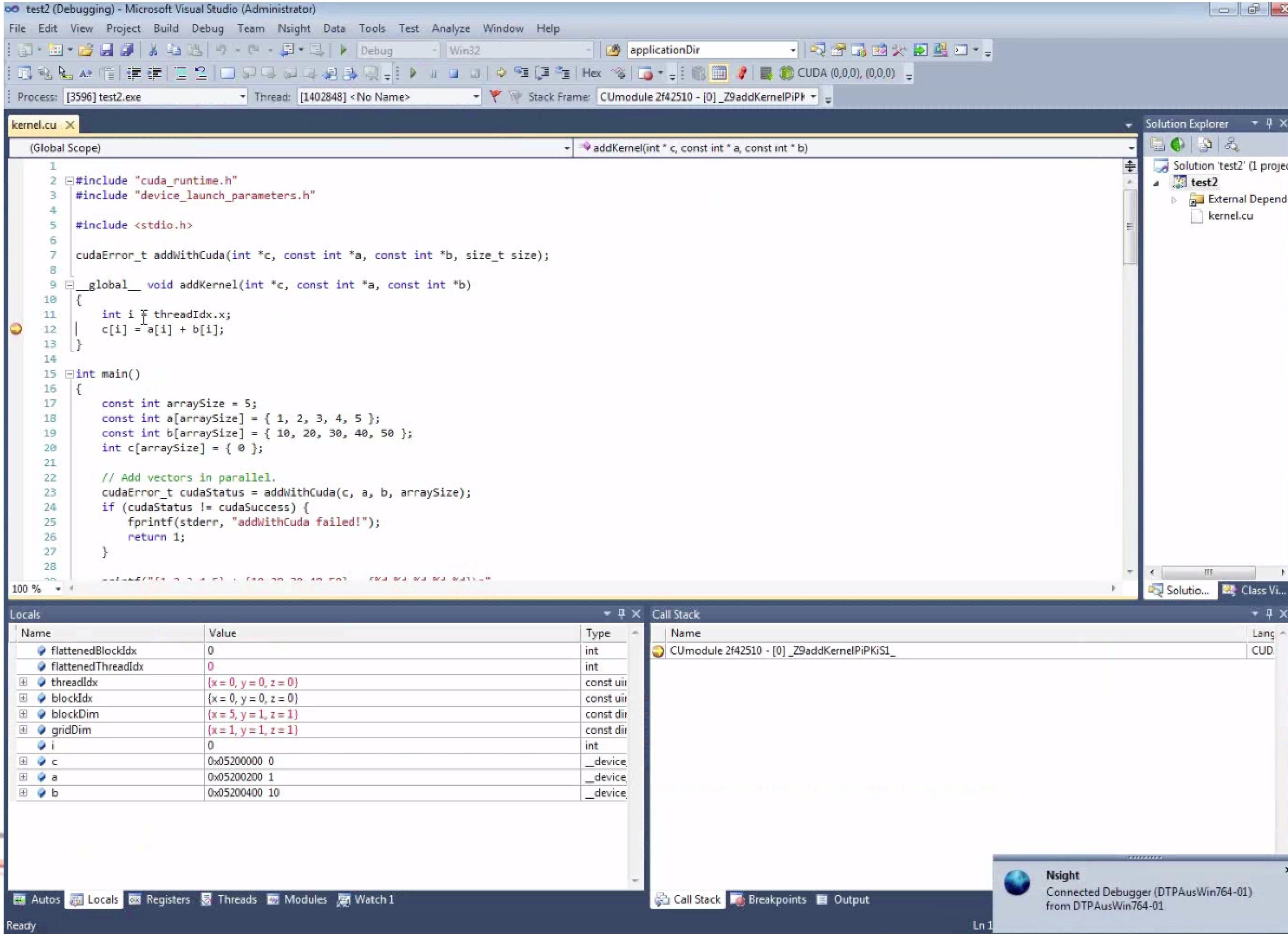
The following target architectures are supported for cross compilation.

- 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

NVIDIA Nsight Eclipse Edition

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

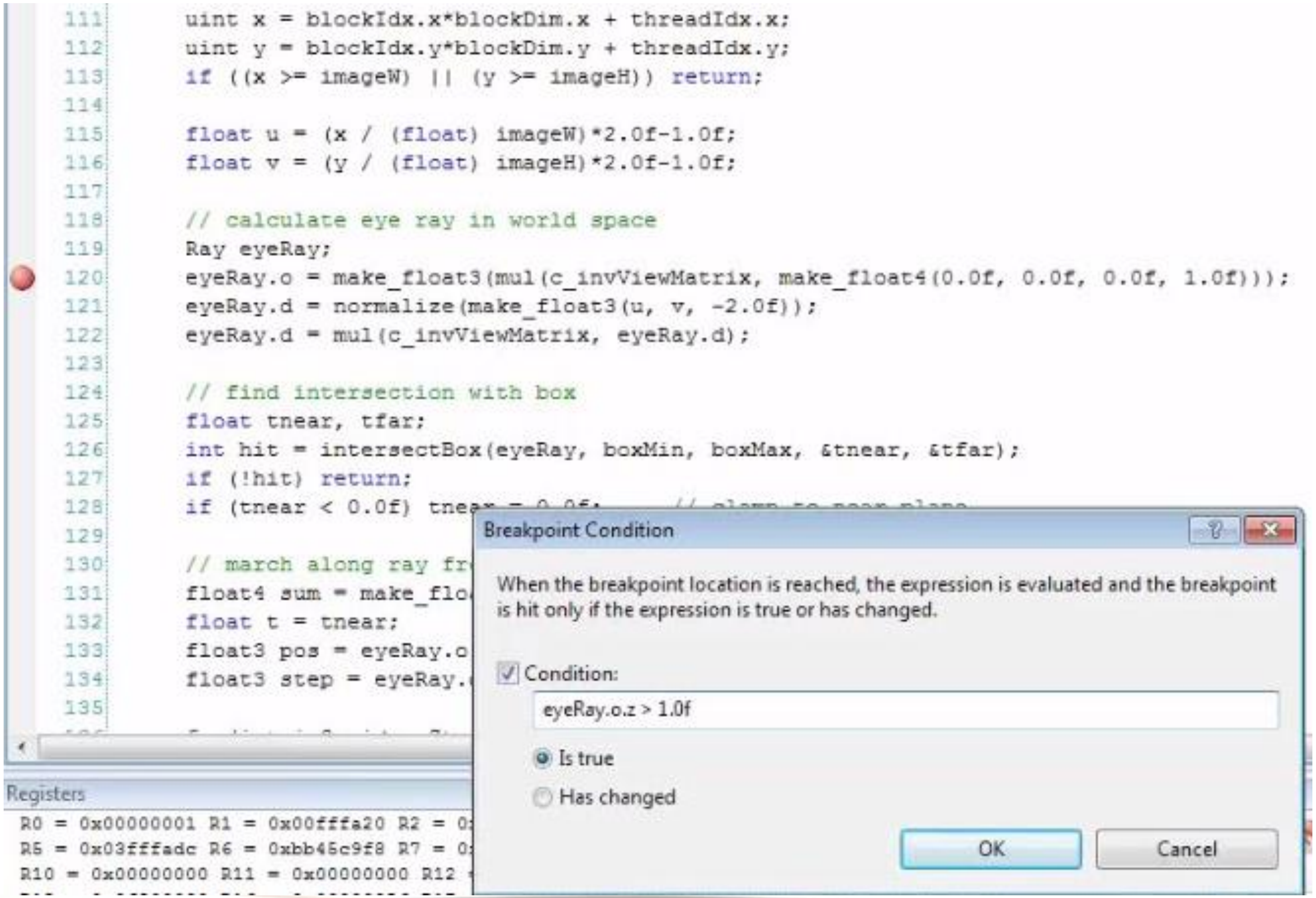
Debugging



test2 (Debugging) - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Team Nsight Data Tools Test Analyze Window Help
Process: [3596] test2.exe
Thread: [1402848] <No Name>
Stack Frame: CUmodule 2f42510 - [0] _Z9addKernelPiPK
CUDA (0,0,0), (0,0,0)
kernel.cu
(Global Scope)
addKernel(int * c, const int * a, const int * b)
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
cudaError_t addWithCuda(int *c, const int *a, const int *b, size_t size);
__global__ void addKernel(int *c, const int *a, const int *b)
{
int i = threadIdx.x;
c[i] = a[i] + b[i];
}
int main()
{
const int arraySize = 5;
const int a[arraySize] = { 1, 2, 3, 4, 5 };
const int b[arraySize] = { 10, 20, 30, 40, 50 };
int c[arraySize] = { 0 };
// Add vectors in parallel.
cudaError_t cudaStatus = addWithCuda(c, a, b, arraySize);
if (cudaStatus != cudaSuccess) {
fprintf(stderr, "addWithCuda failed!");
return 1;
}
Solution Explorer
Solution 'test2' (1 projec
test2
External Depende
kernel.cu
Locals
Name Value Type
flattenedBlockIdx 0 int
flattenedThreadIdx 0 int
threadIdx {x = 0, y = 0, z = 0} const uii
blockIdx {x = 0, y = 0, z = 0} const uii
blockDim {x = 5, y = 1, z = 1} const dir
gridDim {x = 1, y = 1, z = 1} const dir
i 0 int
c 0x05200000 0 __device
a 0x05200200 1 __device
b 0x05200400 10 __device
Call Stack
CUmodule 2f42510 - [0] _Z9addKernelPiPKiS1_
Autos Locals Registers Threads Modules Watch 1
Call Stack Breakpoints Output
Nsight
Connected Debugger (DTPAusWin764-01) from DTPAusWin764-01
Ready

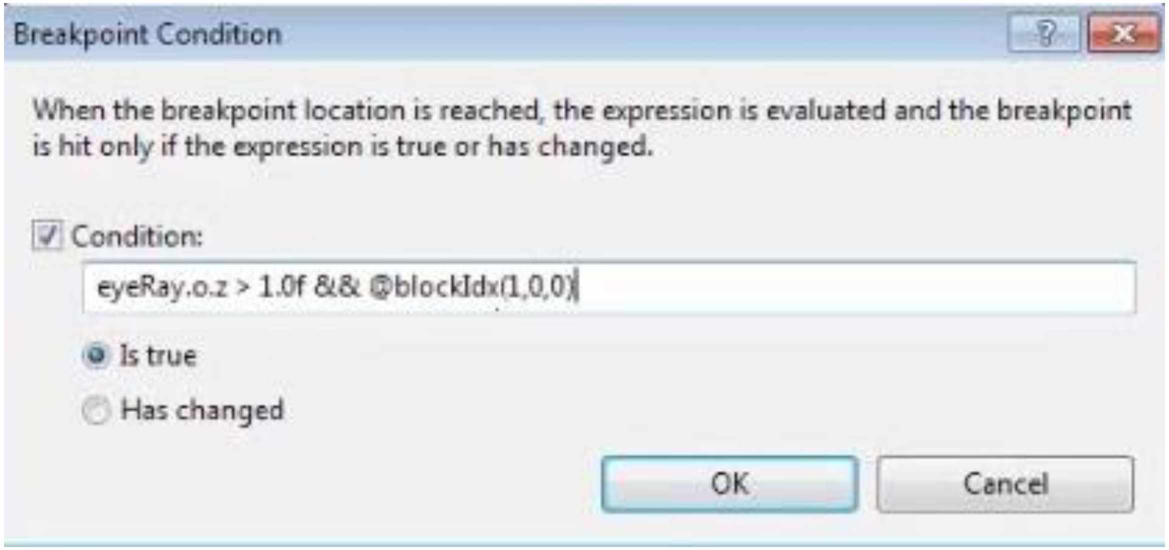
Conditional Breakpoint

- Right click on a breakpoint to set a condition.

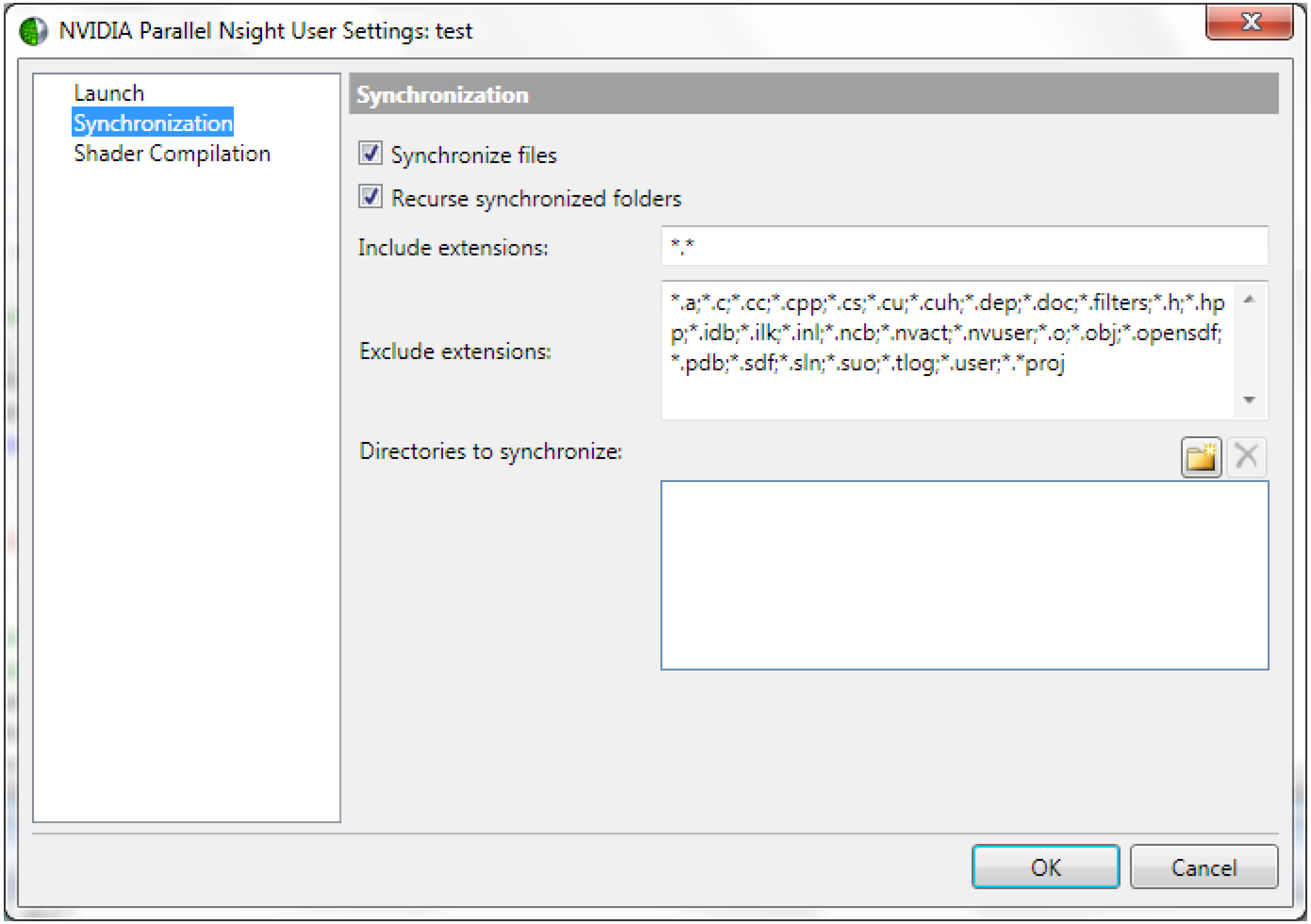


Conditional Breakpoint

- You can select the block as well. In this example, it will only hit the breakpoint if the block id is (1,0,0)

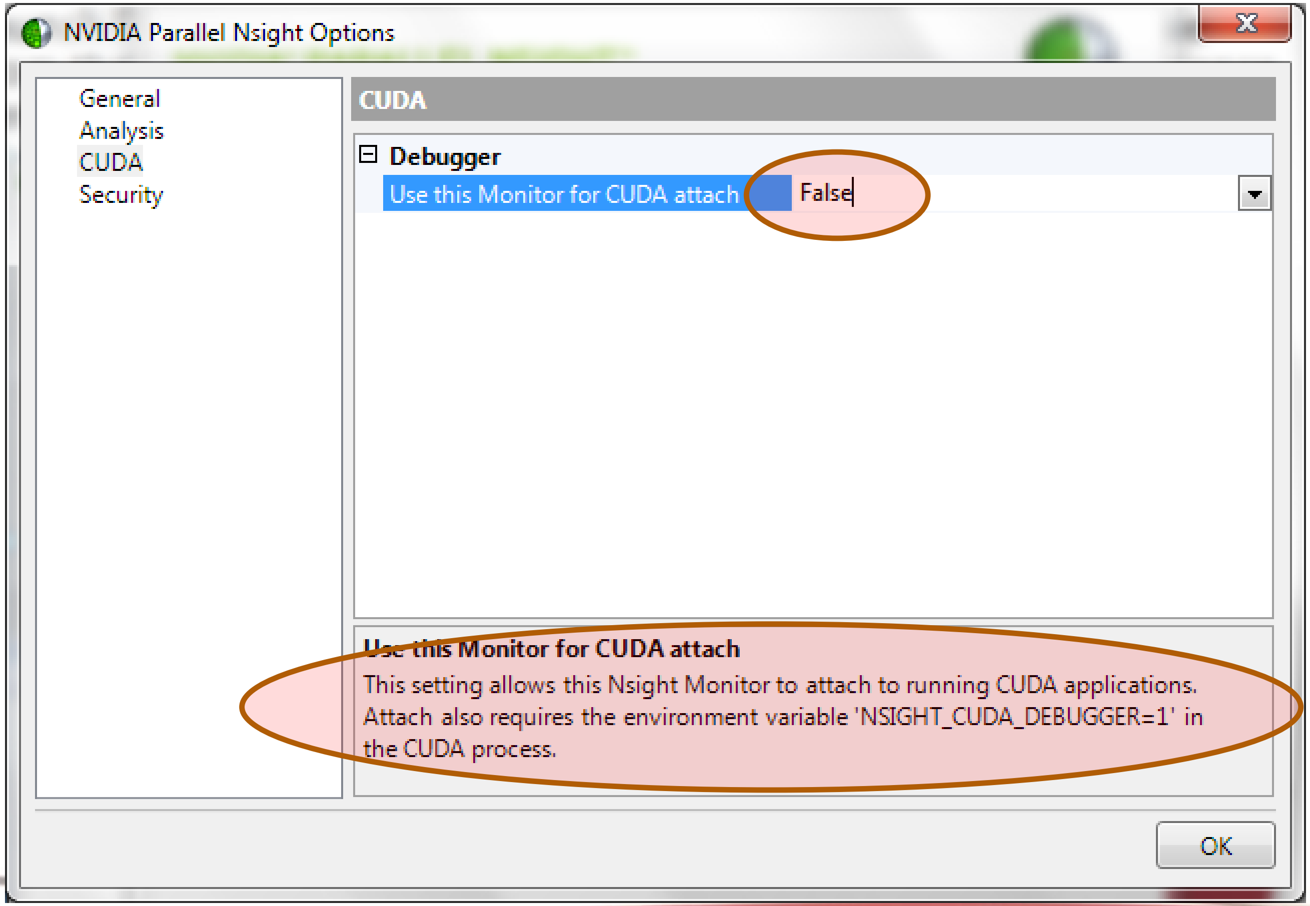


Remote Debugging- Synchronization



Attaching a Process

- Launch Parallel Nsight Monitor and set the following option to **true** from the options. This is set to false by default to prevent performance penalty when attaching to a process is not required.



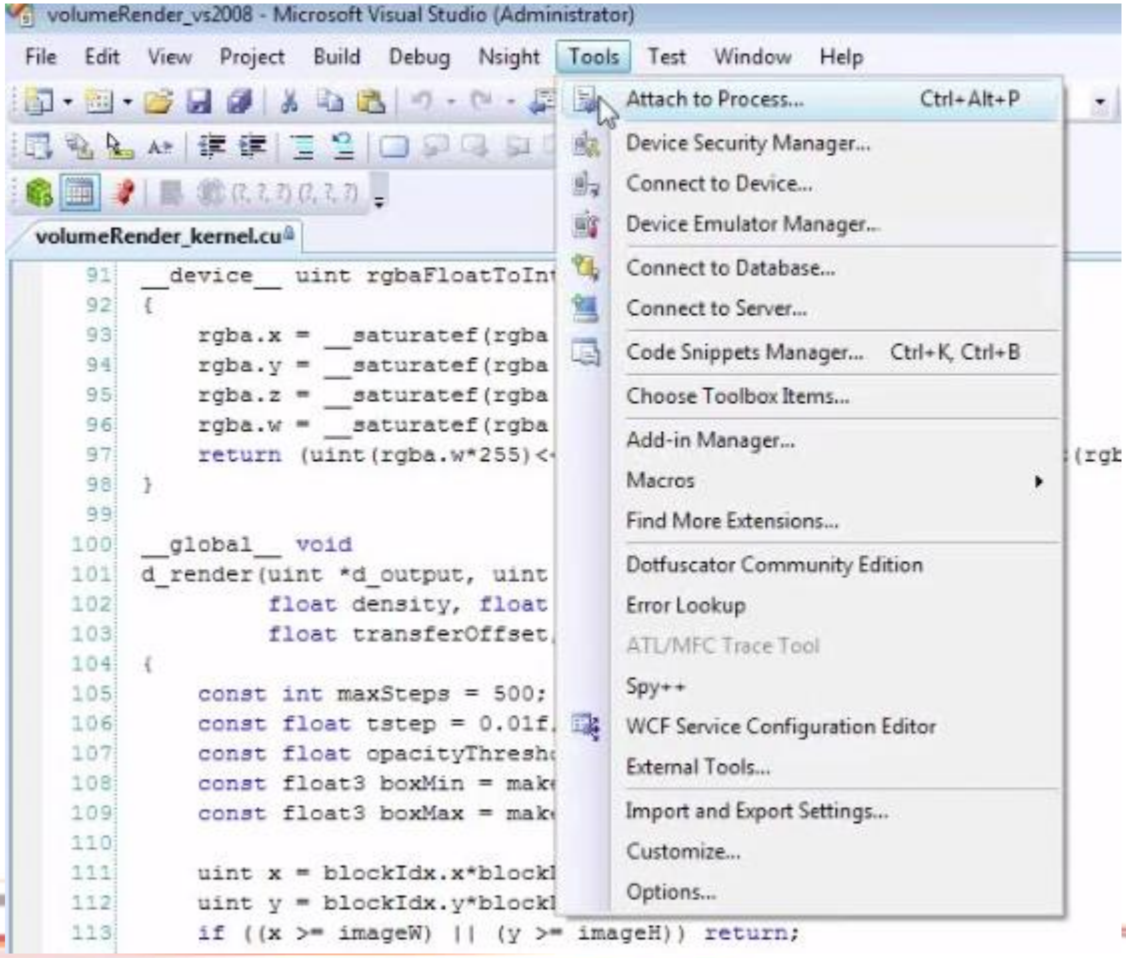
Attaching a Process

- Set the environment variable and launch the process. Here we run an SDK sample called volumeRender

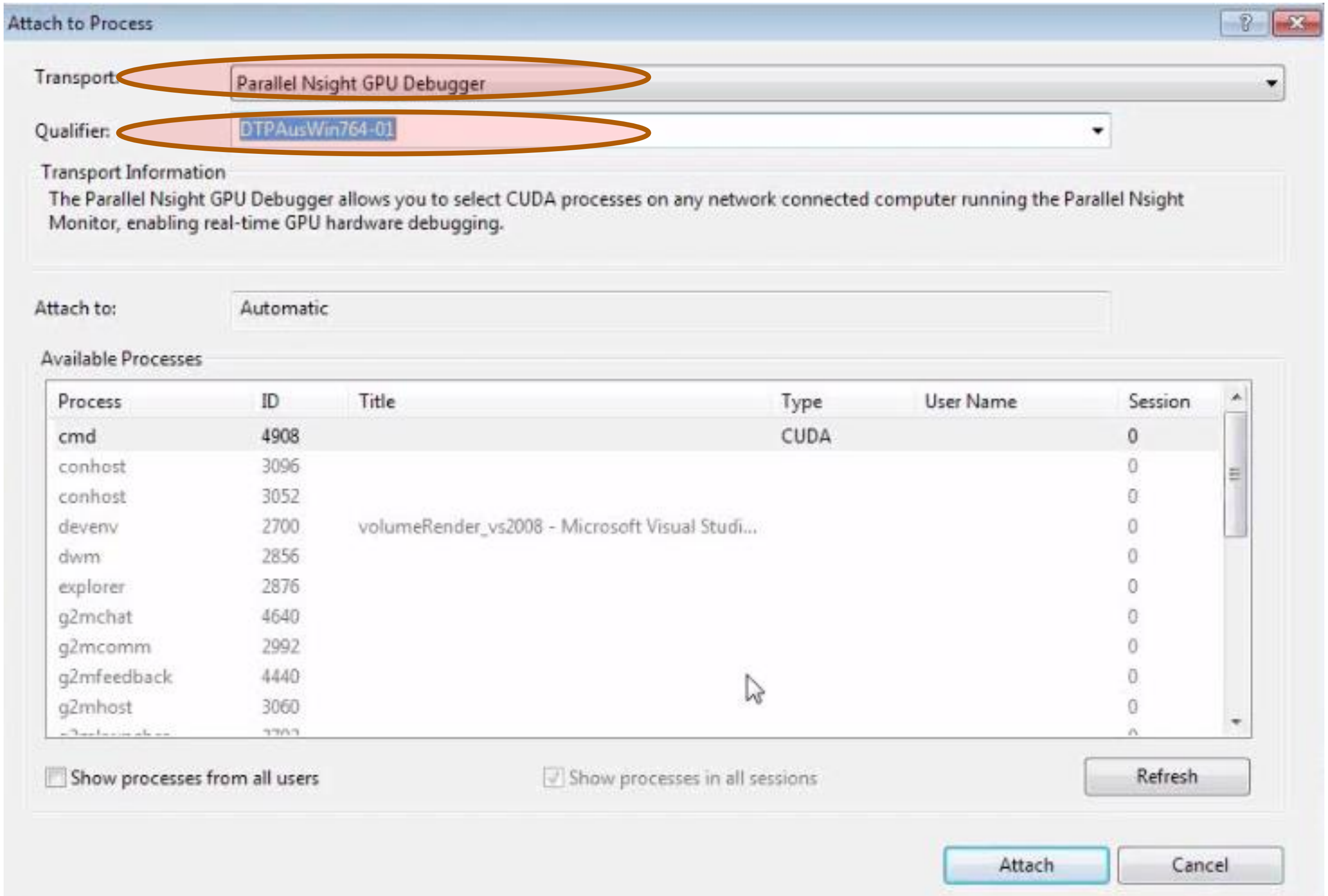
```
Administrator: Visual Studio Command Prompt (2010)
C:\SDKs\Compute\C\bin\win32\Debug>cmd.exe /c volumeRender.exe
[volumeRender.exe] starting...
volumeRender.exe Starting...

Read '../../../src/volumeRender/data/Bucky.raw', 32768 bytes
Press '+' and '-' to change density (0.01 increments)
      ']' and '[' to change brightness
      ';' and ''' to modify transfer function offset
      '.' and ',' to modify transfer function scale


C:\SDKs\Compute\C\bin\win32\Debug>launch volumeRender.exe

C:\SDKs\Compute\C\bin\win32\Debug>SET NSIGHT_CUDA_DEBUGGER=1

C:\SDKs\Compute\C\bin\win32\Debug>cmd.exe /c volumeRender.exe
[volumeRender.exe] starting...
volumeRender.exe Starting...

Read '../../../src/volumeRender/data/Bucky.raw', 32768 bytes
Press '+' and '-' to change density (0.01 increments)
      ']' and '[' to change brightness
      ';' and ''' to modify transfer function offset
      '.' and ',' to modify transfer function scale


C:\SDKs\Compute\C\bin\win32\Debug>
```

Attaching a Process

- Nsight Monitor will watch the processes that is running on the machine.
- Open the project source code in Visual Studio



Attaching a Process

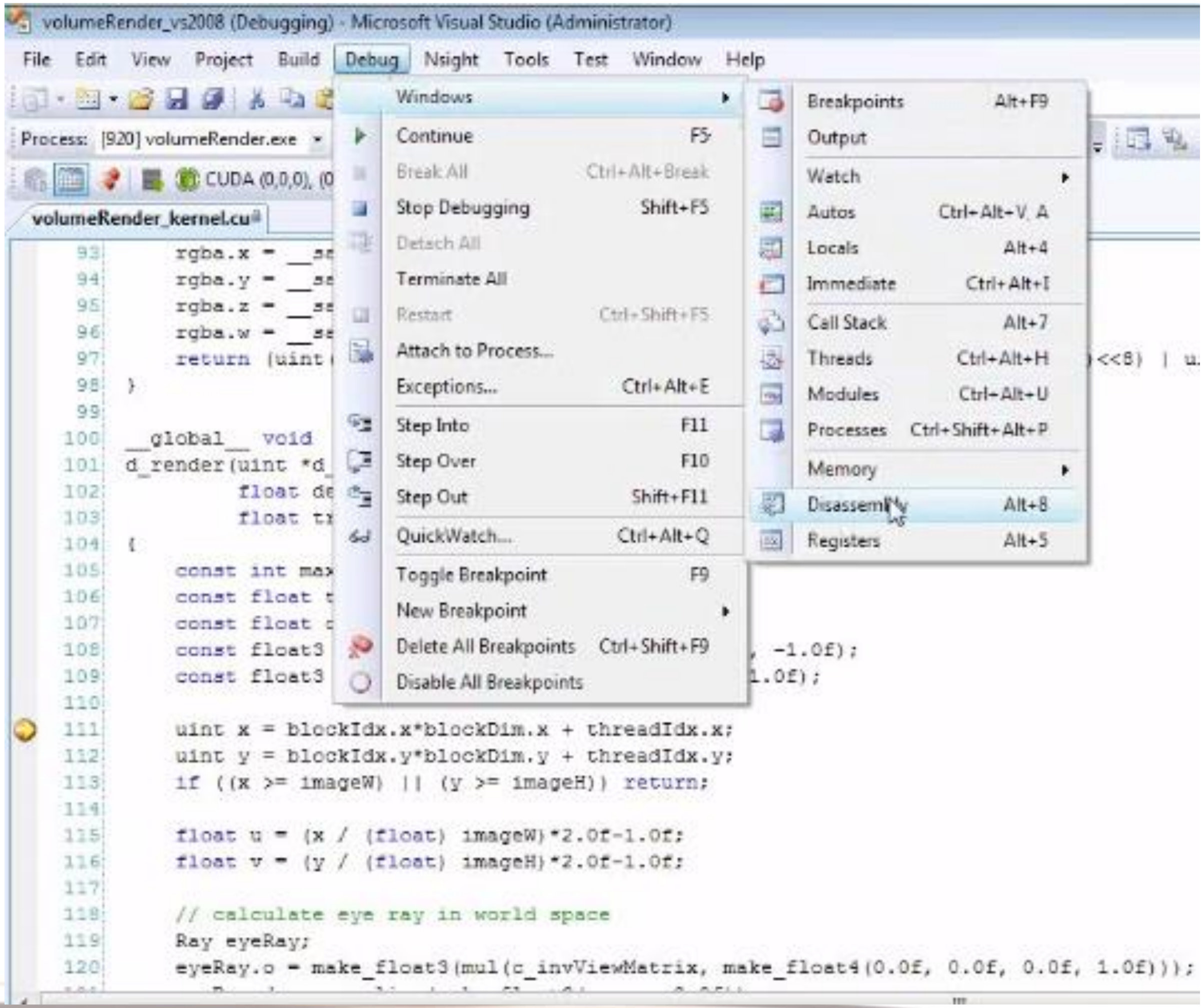


Assembly Debugging

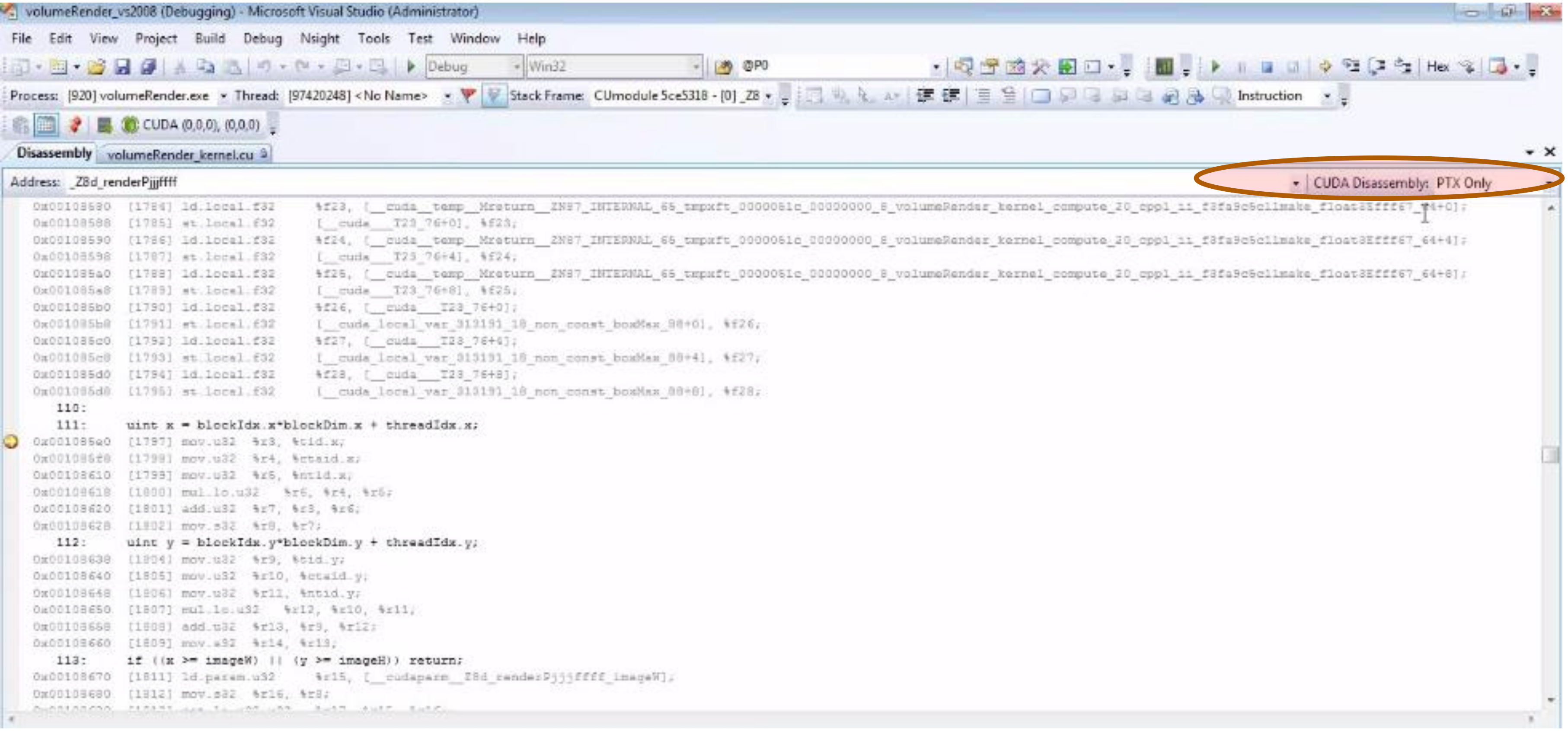
- The assembly on a GPU is different from that of a CPU
- **PTX** is a machine-independent assembly language
- PTX is compiled down to SASS
- **SASS** is the code executed on a particular GPU family
- Most CUDA runtime applications use PTX, since this enables them to run on GPUs released after the original application

PTX/SASS Debugging

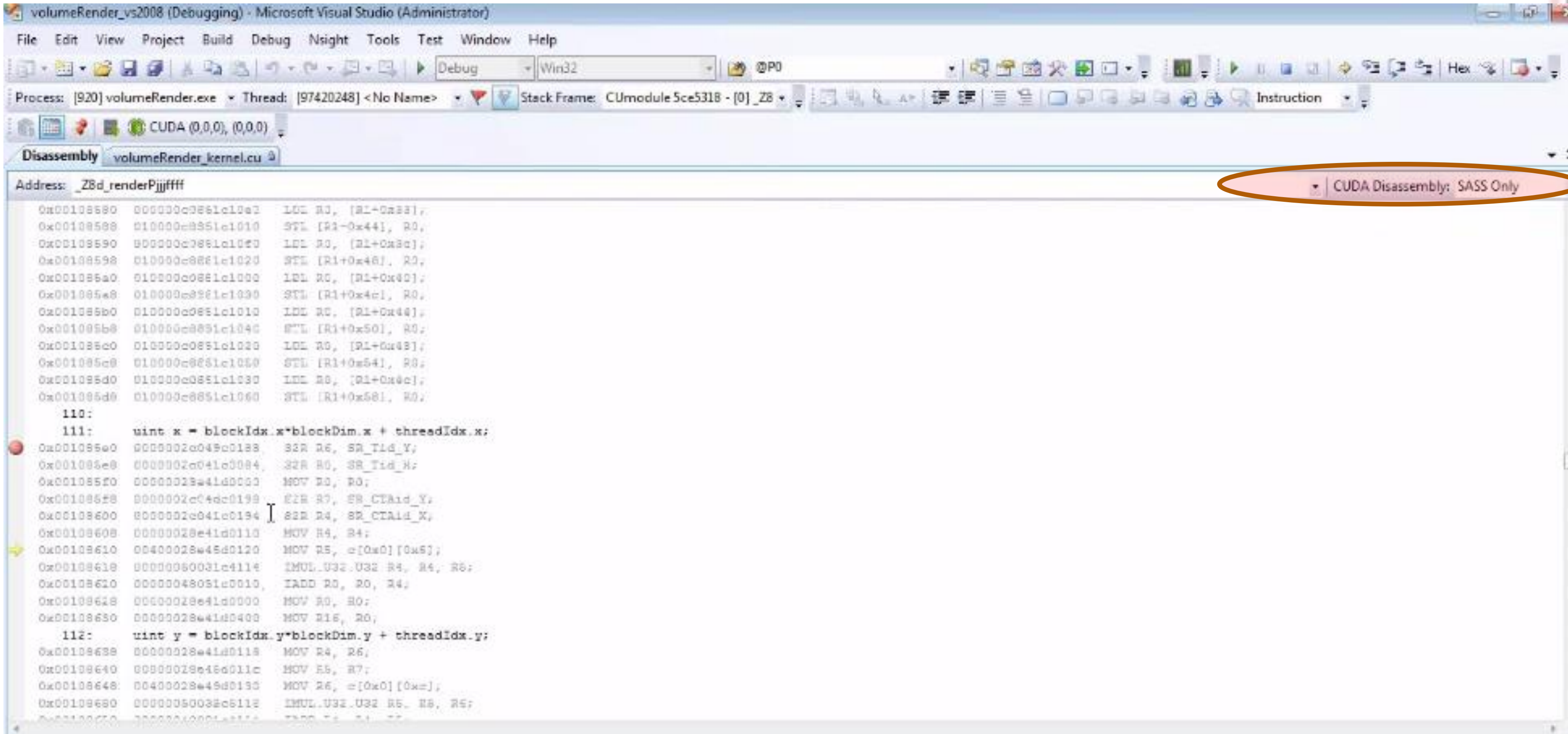
- Set a breakpoint and start debugging



PTX Debugging



SASS Debugging



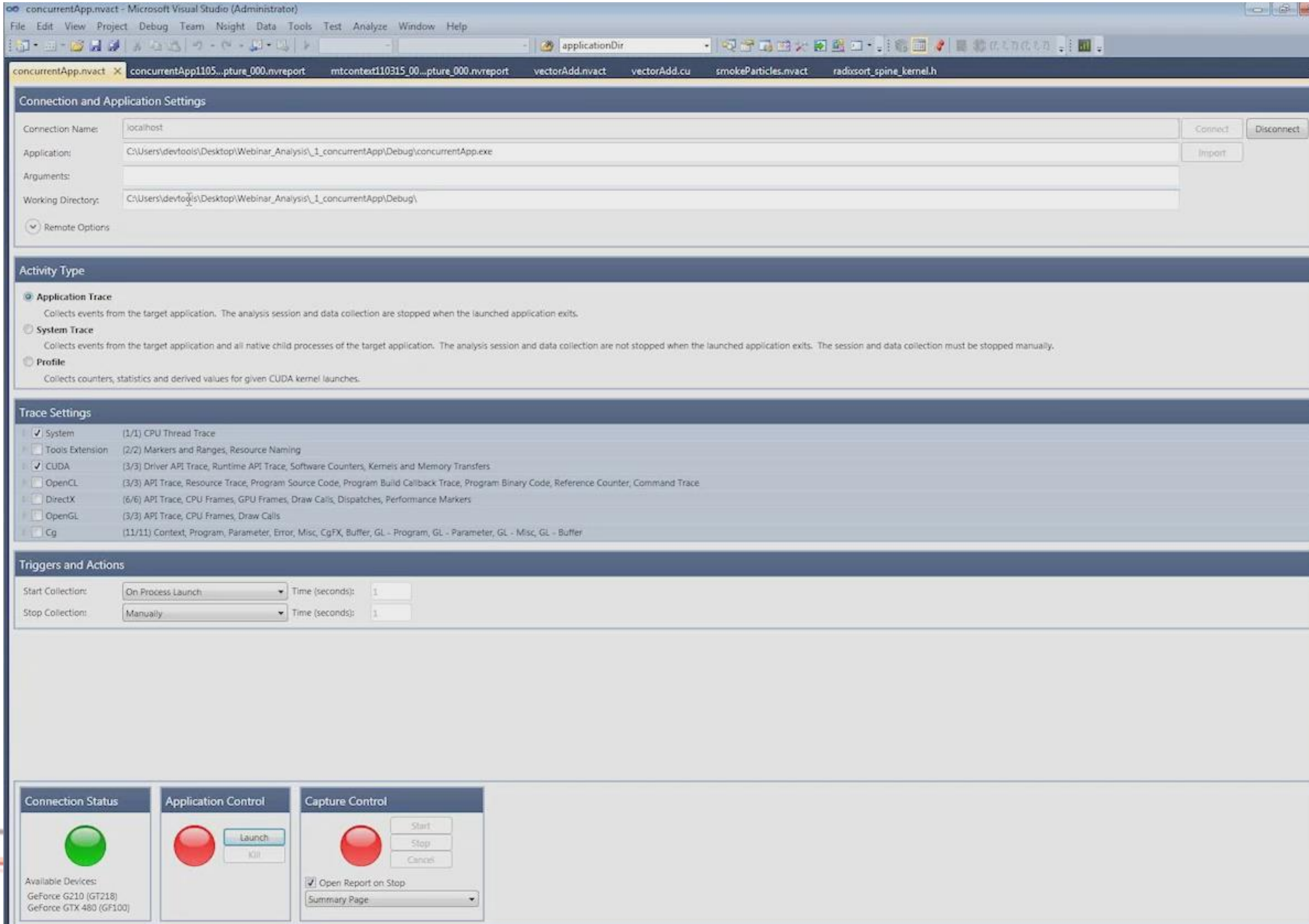
PTX and SASS

```
   111:      uint x = blockIdx.x*blockDim.x + threadIdx.x;
0x001085e0  [1797] mov.u32  %r3, %tid.x;
0x001085e0                0000002c049c0188    S2R R6, SR_Tid_Y;
0x001085e8                0000002c041c0084    S2R R0, SR_Tid_X;
0x001085f0                00000028e41d0000    MOV R0, R0;
0x001085f8  [1798] mov.u32  %r4, %ctaid.x;
0x001085f8                0000002c04dc0198    S2R R7, SR_CTAid_Y;
0x00108600                0000002c041c0194    S2R R4, SR_CTAid_X;
0x00108608                00000028e41d0110    MOV R4, R4;
0x00108610  [1799] mov.u32  %r5, %ntid.x;
0x00108610                00400028e45d0120    MOV R5, c[0x0][0x8];
0x00108618  [1800] mul.lo.u32   %r6, %r4, %r5;
0x00108618                00000050031c4114    IMUL.U32.U32 R4, R4, R5;
0x00108620  [1801] add.u32  %r7, %r3, %r6;
0x00108620                00000048031c0010    IADD R0, R0, R4;
0x00108628  [1802] mov.s32  %r8, %r7;
0x00108628                00000028e41d0000    MOV R0, R0;
0x00108630                00000028e41d0400    MOV R16, R0;
```

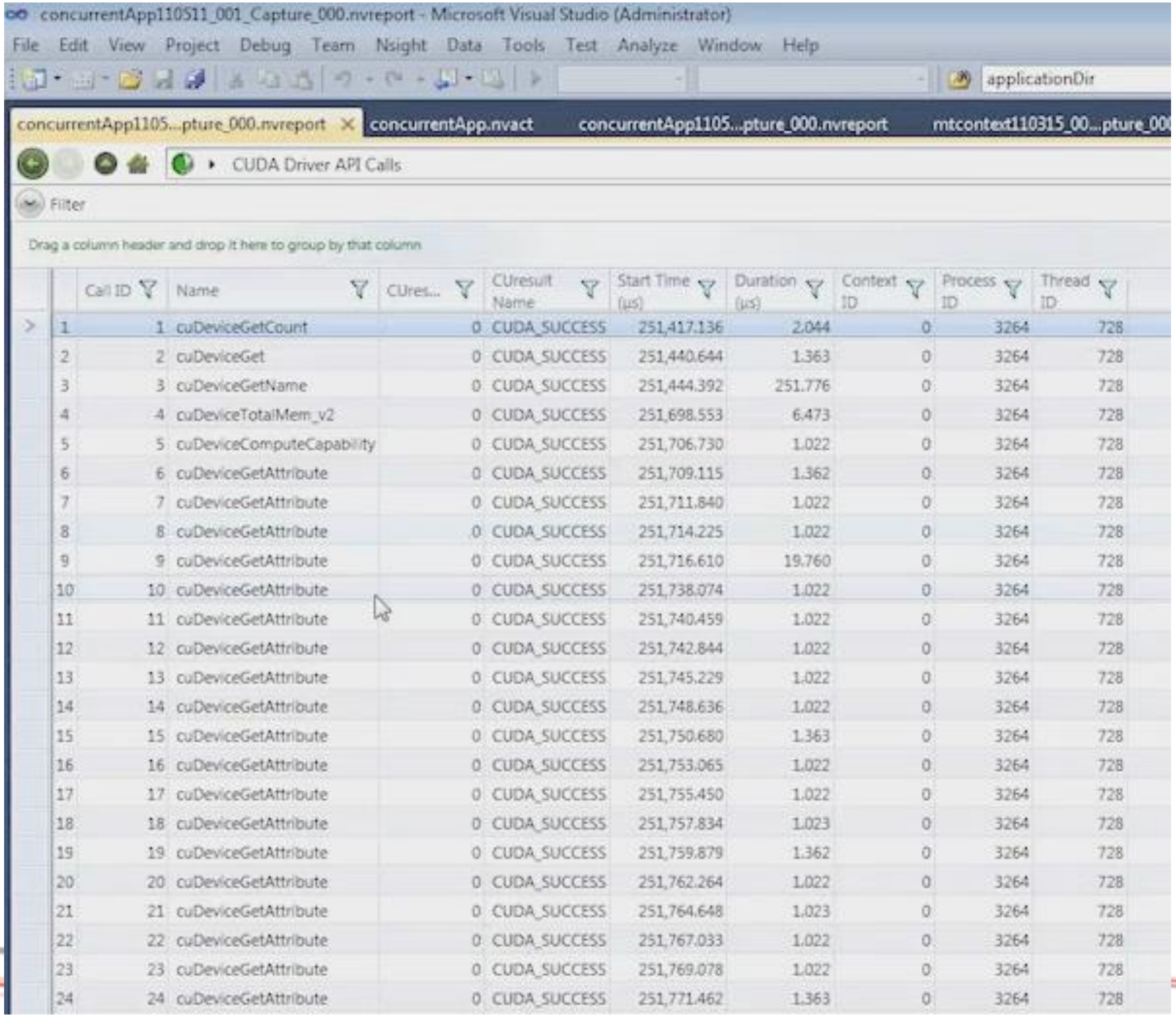
Performance Analysis-Profiling

- We'll run concurrentApp.exe example.
- The example has 4 kernels setup in 4 different streams so they can run concurrently.

Performance Analysis-Profiling

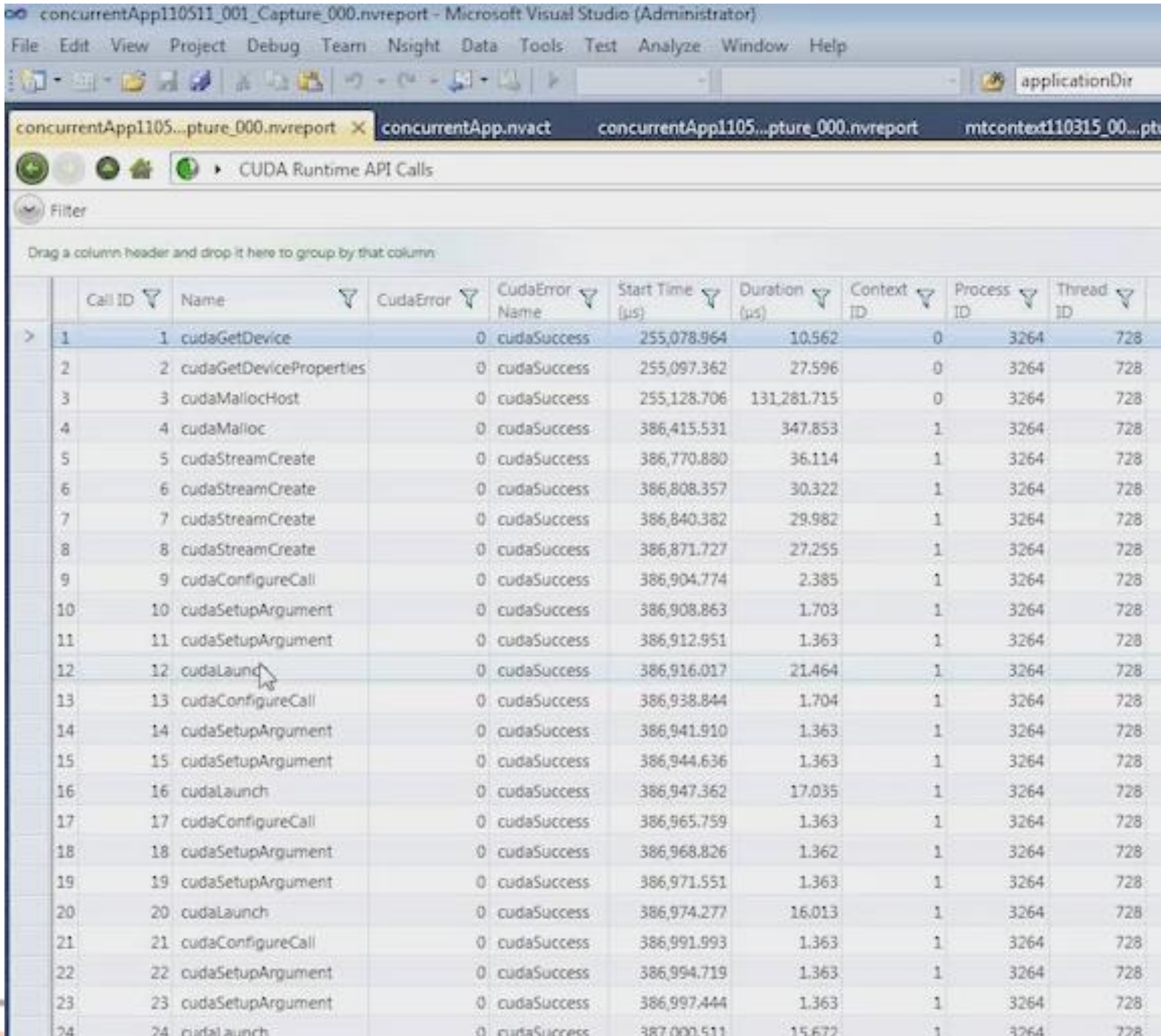


CUDA Driver API Calls



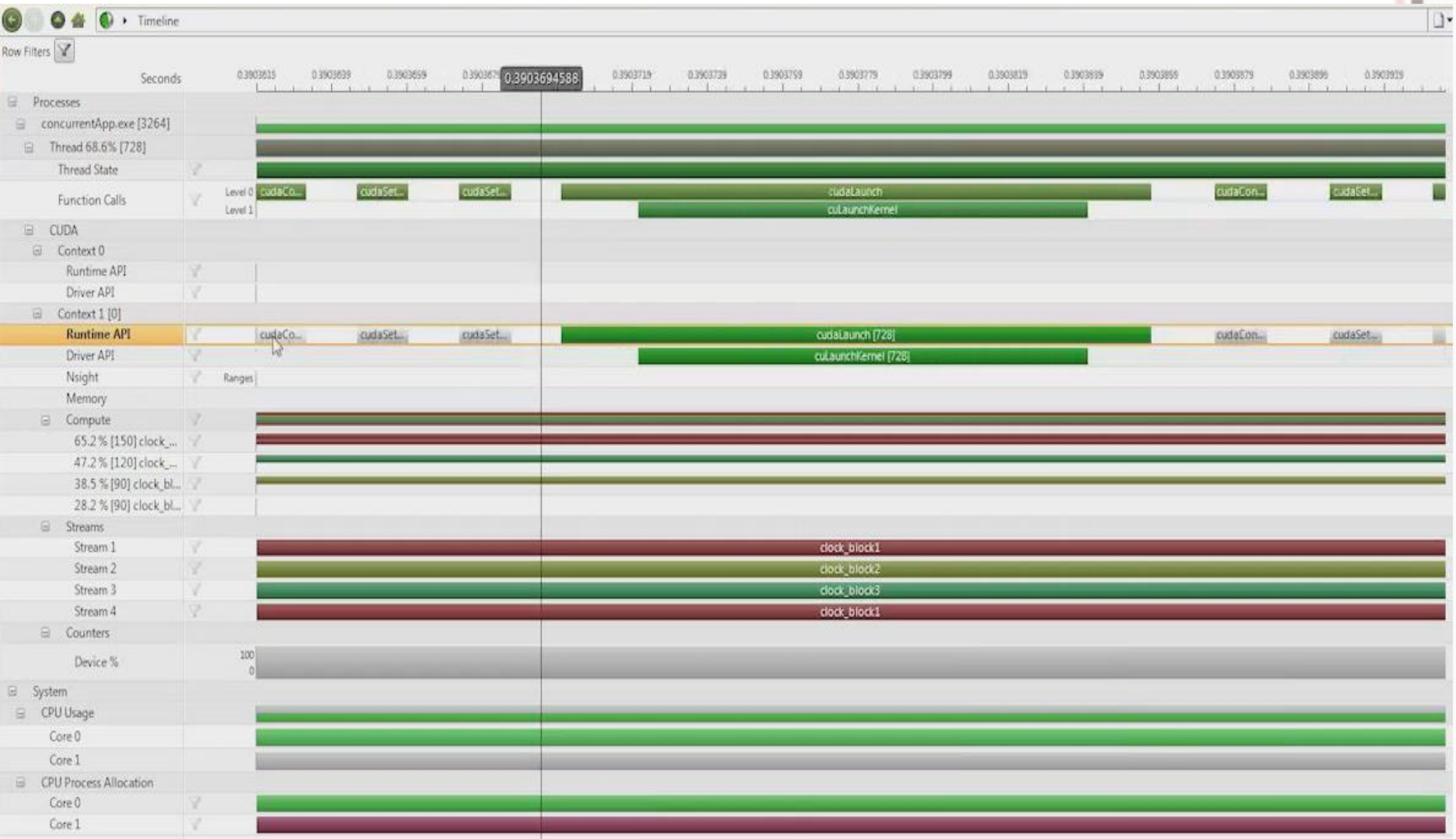
	Call ID	Name	CUres...	CUresult Name	Start Time (µs)	Duration (µs)	Context ID	Process ID	Thread ID
1	1	cuDeviceGetCount	0	CUDA_SUCCESS	251,417.136	2.044	0	3264	728
2	2	cuDeviceGet	0	CUDA_SUCCESS	251,440.644	1.363	0	3264	728
3	3	cuDeviceGetName	0	CUDA_SUCCESS	251,444.392	251.776	0	3264	728
4	4	cuDeviceTotalMem_v2	0	CUDA_SUCCESS	251,698.553	6.473	0	3264	728
5	5	cuDeviceComputeCapability	0	CUDA_SUCCESS	251,706.730	1.022	0	3264	728
6	6	cuDeviceGetAttribute	0	CUDA_SUCCESS	251,709.115	1.362	0	3264	728
7	7	cuDeviceGetAttribute	0	CUDA_SUCCESS	251,711.840	1.022	0	3264	728
8	8	cuDeviceGetAttribute	0	CUDA_SUCCESS	251,714.225	1.022	0	3264	728
9	9	cuDeviceGetAttribute	0	CUDA_SUCCESS	251,716.610	19.760	0	3264	728
10	10	cuDeviceGetAttribute	0	CUDA_SUCCESS	251,738.074	1.022	0	3264	728
11	11	cuDeviceGetAttribute	0	CUDA_SUCCESS	251,740.459	1.022	0	3264	728
12	12	cuDeviceGetAttribute	0	CUDA_SUCCESS	251,742.844	1.022	0	3264	728
13	13	cuDeviceGetAttribute	0	CUDA_SUCCESS	251,745.229	1.022	0	3264	728
14	14	cuDeviceGetAttribute	0	CUDA_SUCCESS	251,748.636	1.022	0	3264	728
15	15	cuDeviceGetAttribute	0	CUDA_SUCCESS	251,750.680	1.363	0	3264	728
16	16	cuDeviceGetAttribute	0	CUDA_SUCCESS	251,753.065	1.022	0	3264	728
17	17	cuDeviceGetAttribute	0	CUDA_SUCCESS	251,755.450	1.022	0	3264	728
18	18	cuDeviceGetAttribute	0	CUDA_SUCCESS	251,757.834	1.023	0	3264	728
19	19	cuDeviceGetAttribute	0	CUDA_SUCCESS	251,759.879	1.362	0	3264	728
20	20	cuDeviceGetAttribute	0	CUDA_SUCCESS	251,762.264	1.022	0	3264	728
21	21	cuDeviceGetAttribute	0	CUDA_SUCCESS	251,764.648	1.023	0	3264	728
22	22	cuDeviceGetAttribute	0	CUDA_SUCCESS	251,767.033	1.022	0	3264	728
23	23	cuDeviceGetAttribute	0	CUDA_SUCCESS	251,769.078	1.022	0	3264	728
24	24	cuDeviceGetAttribute	0	CUDA_SUCCESS	251,771.462	1.363	0	3264	728

CUDA Runtime API Calls



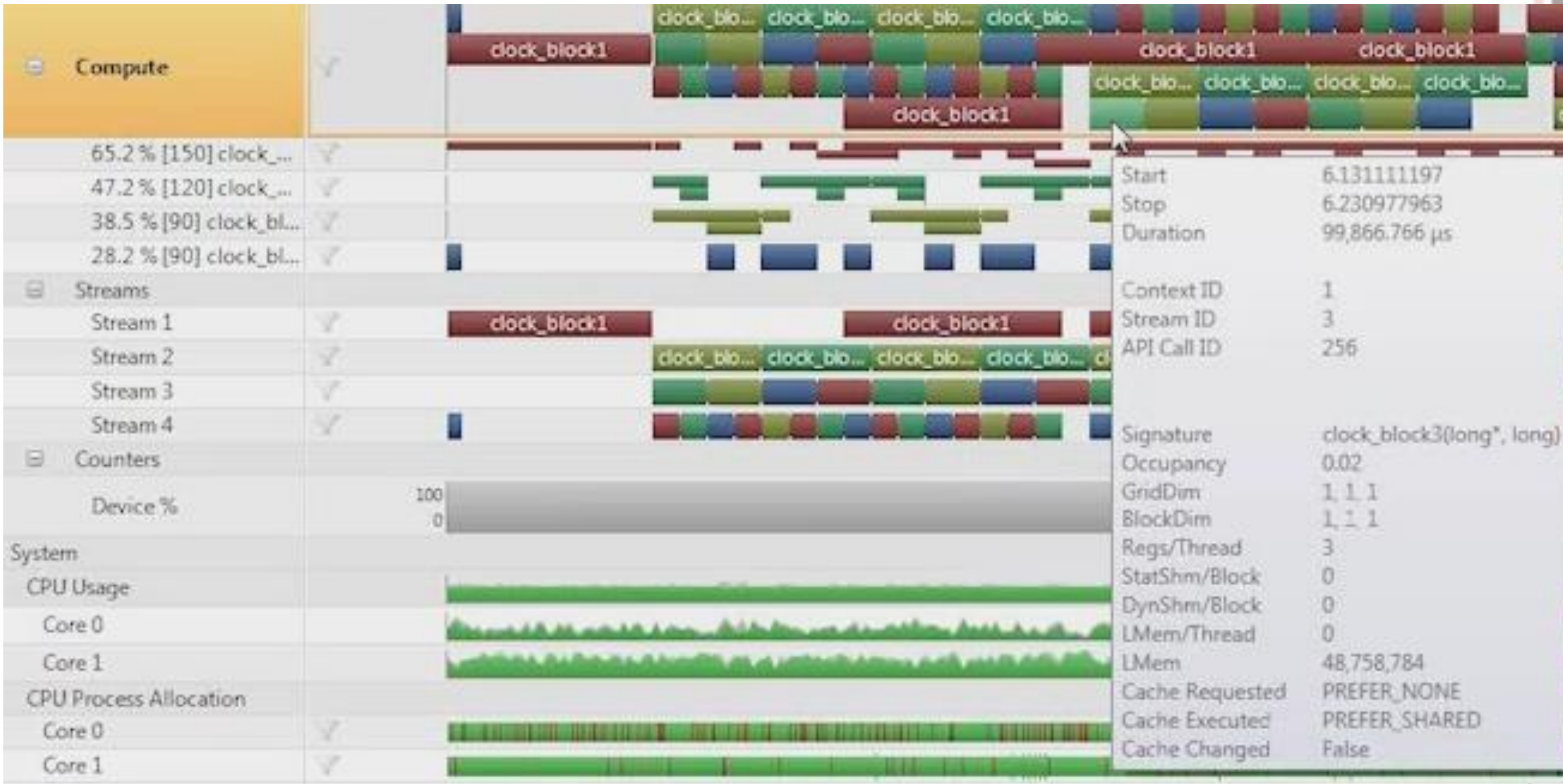
	Call ID	Name	CudaError	CudaError Name	Start Time (µs)	Duration (µs)	Context ID	Process ID	Thread ID
1	1	cudaGetDevice	0	cudaSuccess	255,078.964	10.562	0	3264	728
2	2	cudaGetDeviceProperties	0	cudaSuccess	255,097.362	27.596	0	3264	728
3	3	cudaMallocHost	0	cudaSuccess	255,128.706	131,281.715	0	3264	728
4	4	cudaMalloc	0	cudaSuccess	386,415.531	347.853	1	3264	728
5	5	cudaStreamCreate	0	cudaSuccess	386,770.880	36.114	1	3264	728
6	6	cudaStreamCreate	0	cudaSuccess	386,808.357	30.322	1	3264	728
7	7	cudaStreamCreate	0	cudaSuccess	386,840.382	29.982	1	3264	728
8	8	cudaStreamCreate	0	cudaSuccess	386,871.727	27.255	1	3264	728
9	9	cudaConfigureCall	0	cudaSuccess	386,904.774	2.385	1	3264	728
10	10	cudaSetupArgument	0	cudaSuccess	386,908.863	1.703	1	3264	728
11	11	cudaSetupArgument	0	cudaSuccess	386,912.951	1.363	1	3264	728
12	12	cudaLaunch	0	cudaSuccess	386,916.017	21.464	1	3264	728
13	13	cudaConfigureCall	0	cudaSuccess	386,938.844	1.704	1	3264	728
14	14	cudaSetupArgument	0	cudaSuccess	386,941.910	1.363	1	3264	728
15	15	cudaSetupArgument	0	cudaSuccess	386,944.636	1.363	1	3264	728
16	16	cudaLaunch	0	cudaSuccess	386,947.362	17.035	1	3264	728
17	17	cudaConfigureCall	0	cudaSuccess	386,965.759	1.363	1	3264	728
18	18	cudaSetupArgument	0	cudaSuccess	386,968.826	1.362	1	3264	728
19	19	cudaSetupArgument	0	cudaSuccess	386,971.551	1.363	1	3264	728
20	20	cudaLaunch	0	cudaSuccess	386,974.277	16.013	1	3264	728
21	21	cudaConfigureCall	0	cudaSuccess	386,991.993	1.363	1	3264	728
22	22	cudaSetupArgument	0	cudaSuccess	386,994.719	1.363	1	3264	728
23	23	cudaSetupArgument	0	cudaSuccess	386,997.444	1.363	1	3264	728
24	24	cudaLaunch	0	cudaSuccess	387,000.511	15.672	1	3264	728

Timeline

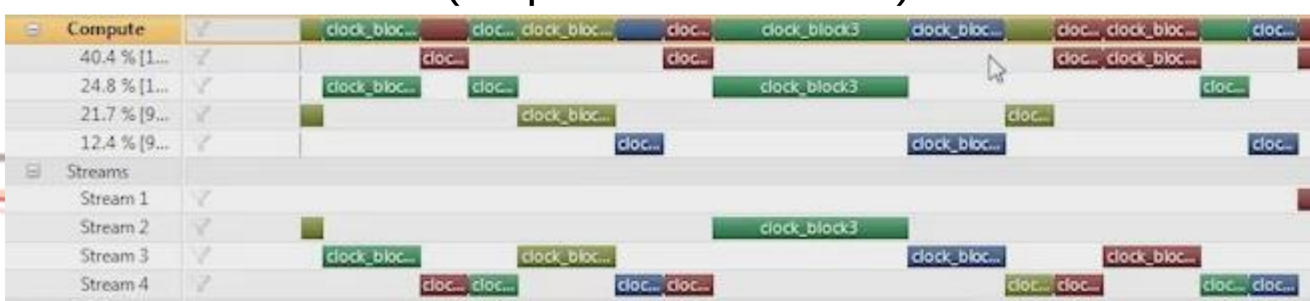


Timeline
Row Filters
Seconds
0.3903694588
Processes
concurrentApp.exe [3264]
Thread 68.6% [728]
Thread State
Function Calls
Level 0
Level 1
cudaCo...
cudaSet...
cudaSet...
cudaLaunch
cudaCon...
cudaSet...
cuLaunchKernel
CUDA
Context 0
Runtime API
Driver API
Context 1 [0]
Runtime API
cudaCo...
cudaSet...
cudaSet...
cudaLaunch [728]
cudaCon...
cudaSet...
Driver API
cuLaunchKernel [728]
Nsight
Ranges
Memory
Compute
65.2 % [150] clock_...
47.2 % [120] clock_...
38.5 % [90] clock_bl...
28.2 % [90] clock_bl...
Streams
Stream 1
clock_block1
Stream 2
clock_block2
Stream 3
clock_block3
Stream 4
clock_block1
Counters
Device %
100
0
System
CPU Usage
Core 0
Core 1
CPU Process Allocation
Core 0
Core 1

Timeline- Compute Row



Older version (sequential execution)

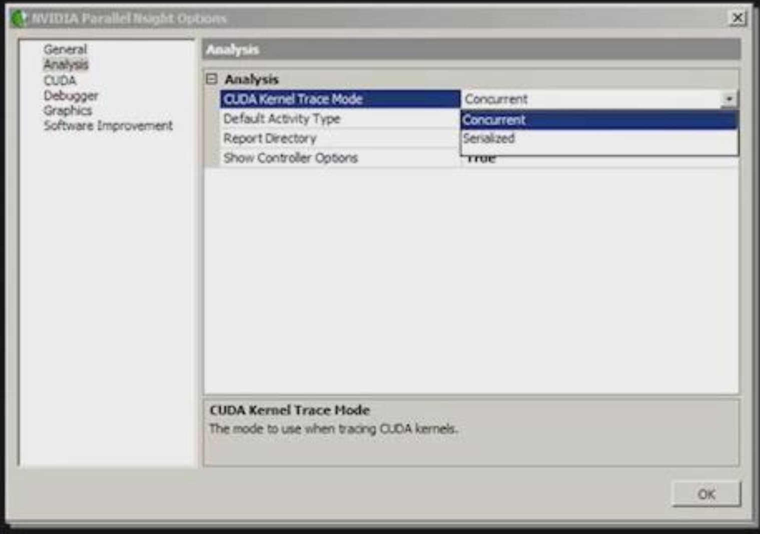


Concurrent/Serialized Execution

Concurrent Trace Modes

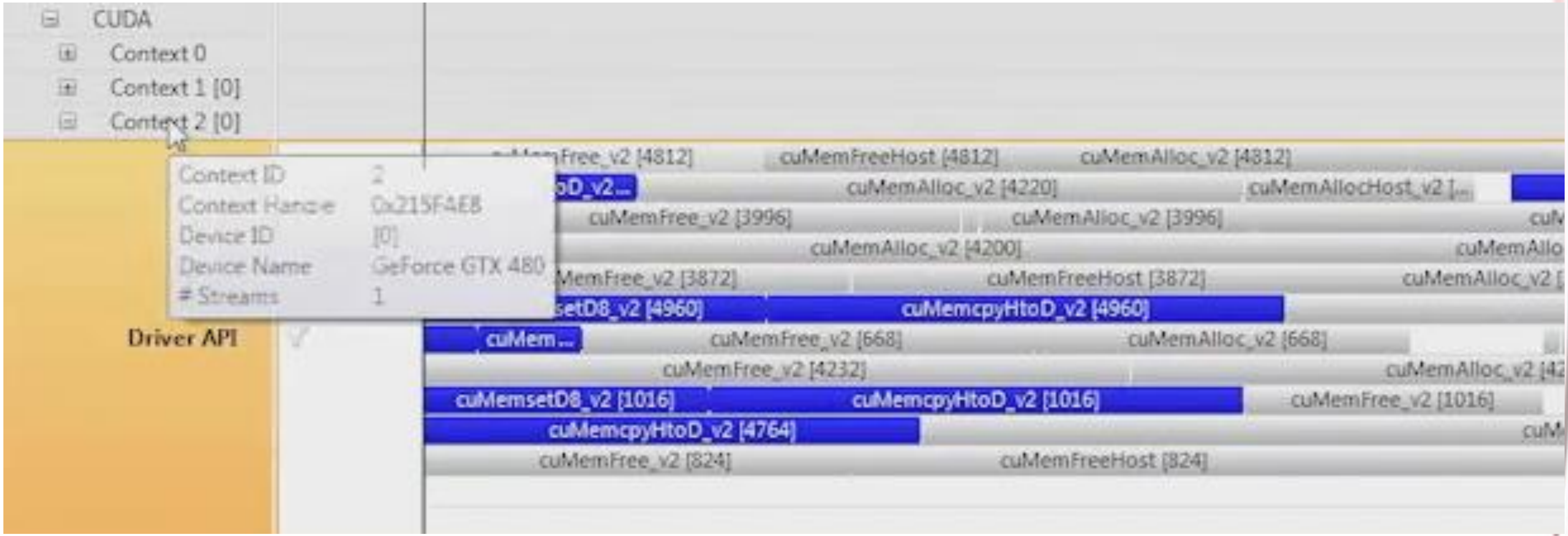
Trace Mode is configurable from the Nsight Options Menu

- **Concurrent:**
  - GPU overhead scales with number of warps launched
  - Additional GPU overhead introduced to `cuCtxSynchronize()`
  - Supported on Fermi cards only*
- **Serialized:**
  - Constant overhead per kernel launch
  - No additional GPU overhead

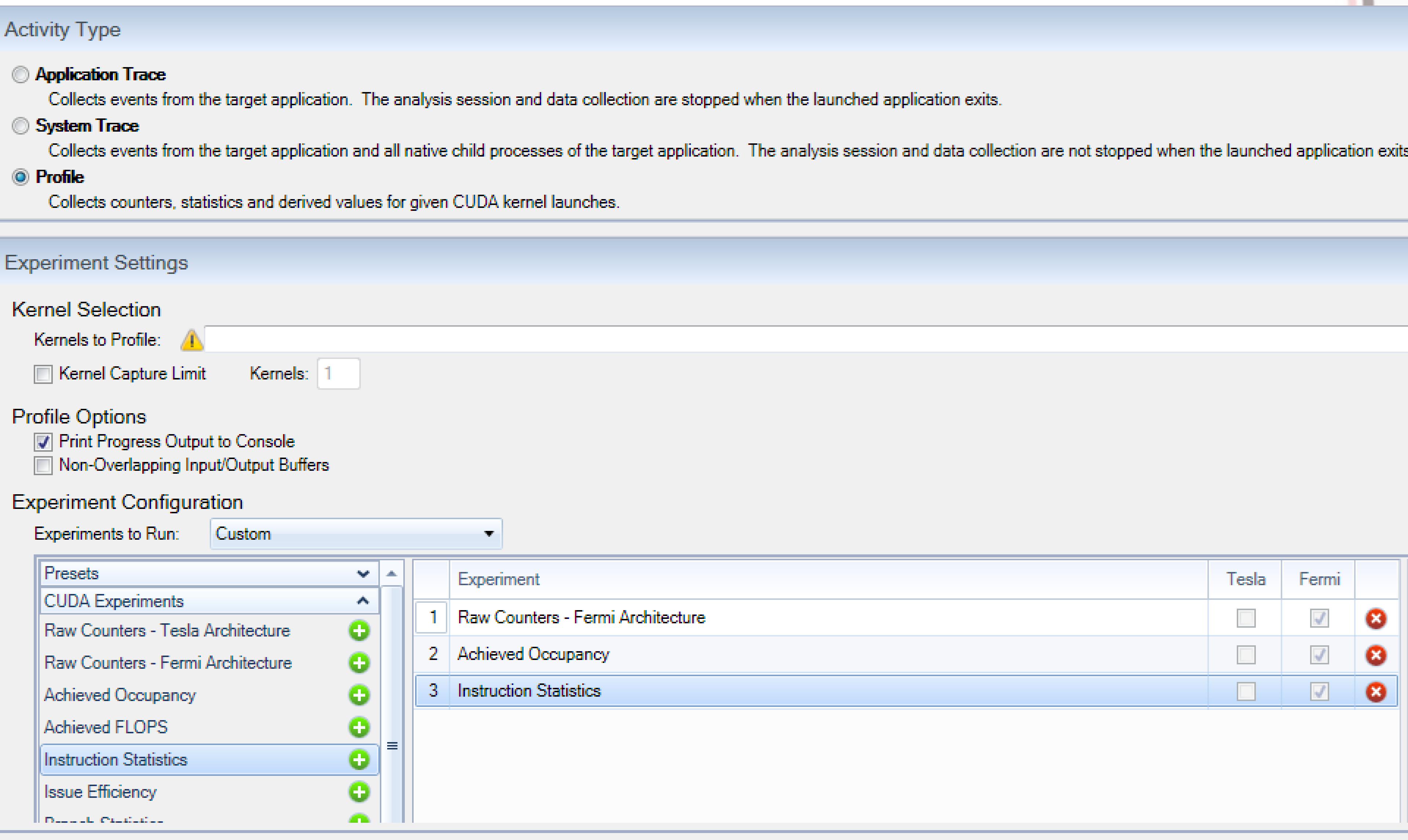


* Currently not available on GTX580, GTX570, and GT520

Parallel API calls



Profiling


Activity Type
Application Trace
Collects events from the target application. The analysis session and data collection are stopped when the launched application exits.
System Trace
Collects events from the target application and all native child processes of the target application. The analysis session and data collection are not stopped when the launched application exits
Profile
Collects counters, statistics and derived values for given CUDA kernel launches.
Experiment Settings
Kernel Selection
Kernels to Profile:
Kernel Capture Limit
Kernels: 1
Profile Options
Print Progress Output to Console
Non-Overlapping Input/Output Buffers
Experiment Configuration
Experiments to Run: Custom
Presets
CUDA Experiments
Raw Counters - Tesla Architecture
Raw Counters - Fermi Architecture
Achieved Occupancy
Achieved FLOPS
Instruction Statistics
Issue Efficiency
Experiment
Tesla
Fermi
1 Raw Counters - Fermi Architecture
2 Achieved Occupancy
3 Instruction Statistics

Parameters: Achieved Occupancy

- The multiprocessor occupancy is the ratio of active warps to the maximum number of warps supported on a multiprocessor of the GPU.

- The theoretical occupancy is the maximum occupancy given the execution configuration.

- Captures the achieved occupancy, which is the number of active warps per clock cycle divided by maximum warps per multiprocessor.

- Higher occupancy does not always equate to higher performance - there is a point above which additional occupancy does not improve performance. However, low occupancy always interferes with the ability to hide memory latency, resulting in performance degradation.

Parameters: Achieved Occupancy

- The achieved occupancy may be lower than the theoretical occupancy for the following reasons:
  - The total number of blocks in the execution configuration is insufficient to saturate the device.
  - The thread blocks have high variance in execution times, often resulting in low number of active warps toward the end of the kernel execution.
  - The warps in thread blocks have high variance in execution times resulting in poor number of active warps throughout the kernel execution.

Parameters: Instruction Statistics

- Provides key metrics to evaluate the efficiency to execute the kernel's instruction on the target device, including instructions per clock cycle (IPC), instruction serialization, SM activity, as well as instructions per warp (IPW).
- If the executed IPC is low and the serialization high, the kernel likely has poor memory access patterns to shared/global memory.
- If the executed IPC is low and the serialization low, the kernel may execute a large number of high latency operations, such as double instructions, transcendentals, or memory operations.
- If the executed IPC is low and the achieved occupancy is low, the kernel launch fails to successfully hide latency.

```
// Device Code
__global__ void VecAdd_A(const float* A, const float* B, float* C)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    C[i] = A[i] + B[i];
}

```

Occupancy Experiment- 8 blocks

Has 15 SMs

Kernel: VecAdd_A

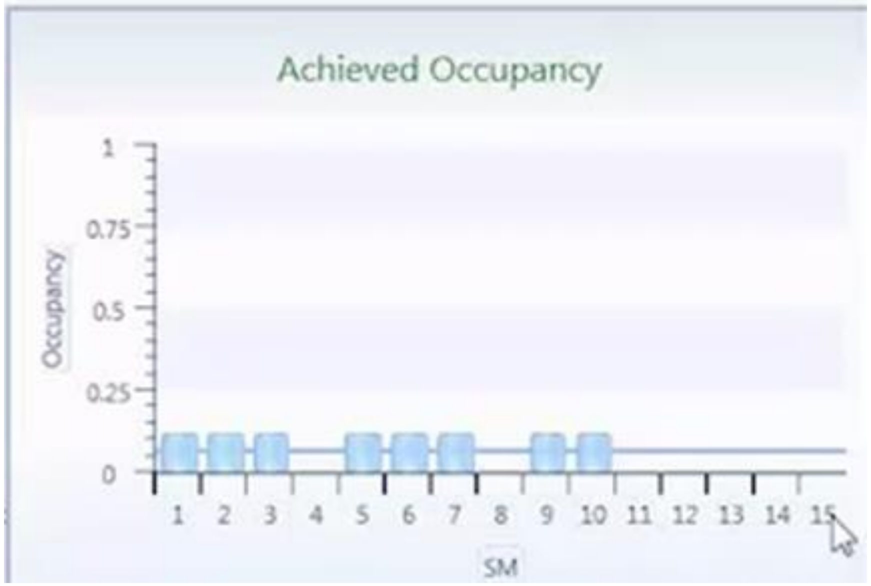
Grid Dim: (8, 1, 1) 8 **Block Dim:** (192, 1, 1) 192

Device: GeForce GTX 480

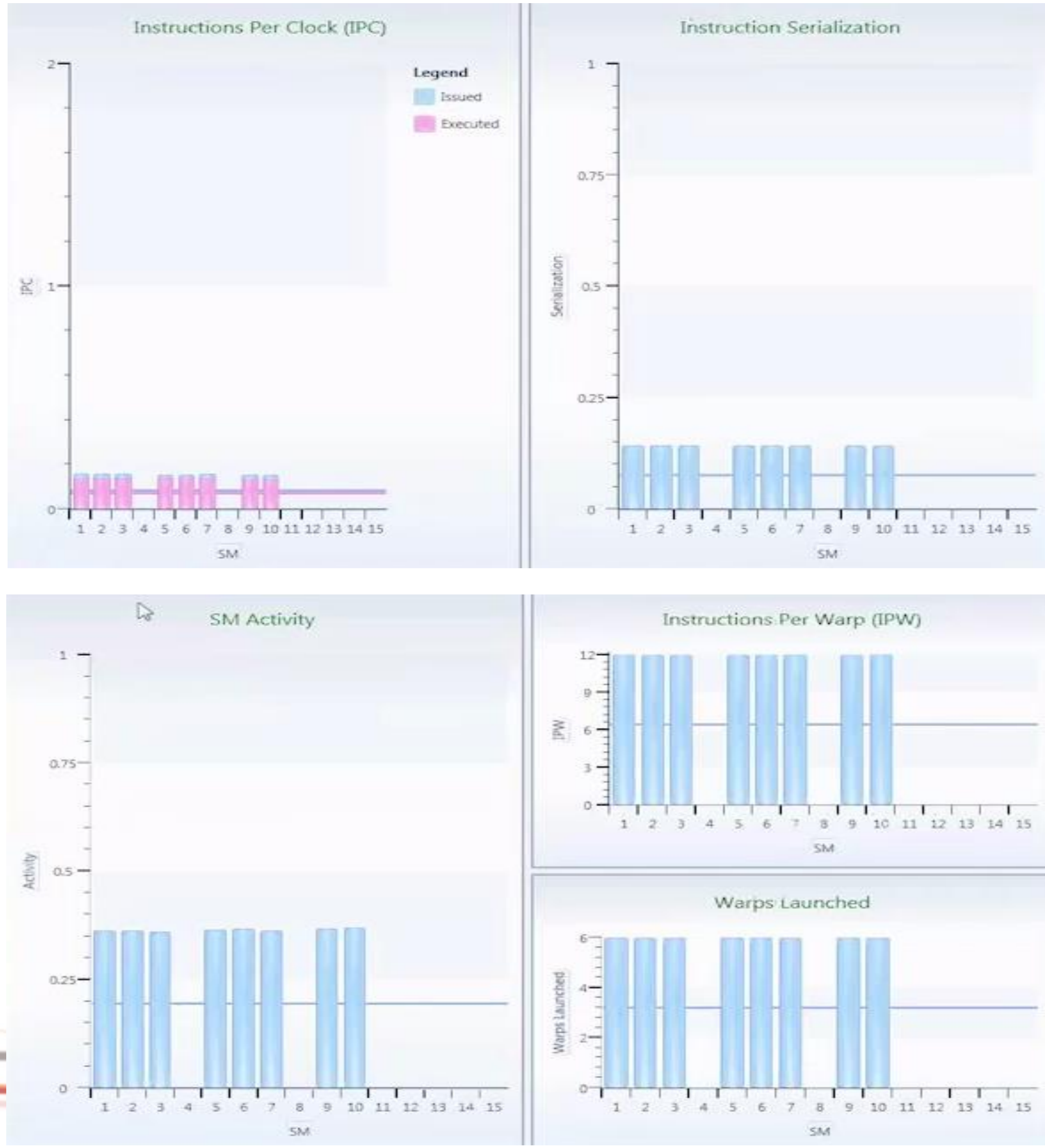
Compute Capability: 2.0

Variable	Achieved	Theoretical	Device Limit
Occupancy Per SM			
Active Blocks		8	8
Active Warps	3.19	48	48
Active Threads		1536	1536
Occupancy	6.64 %	100.00 %	100.00 %
Warps			
Threads/Block		192	1024
Warps/Block		6	32
Block Limit		8	8
Registers			
Registers/Thread		4	63
Registers/Block		768	32768
Block Limit		42	8
Shared Memory			
Shared Memory/Block		0	49152
Block Limit		8	8

Occupancy Experiment- 8 blocks



Occupancy Experiment- 8 blocks (Inst Stat. Tab)



Occupancy Experiment- 256 blocks

Kernel: VecAdd_A
Grid Dim: {256, 1, 1} 256 **Block Dim:** {192, 1, 1} 192
Device: GeForce GTX 480
Compute Capability: 2.0

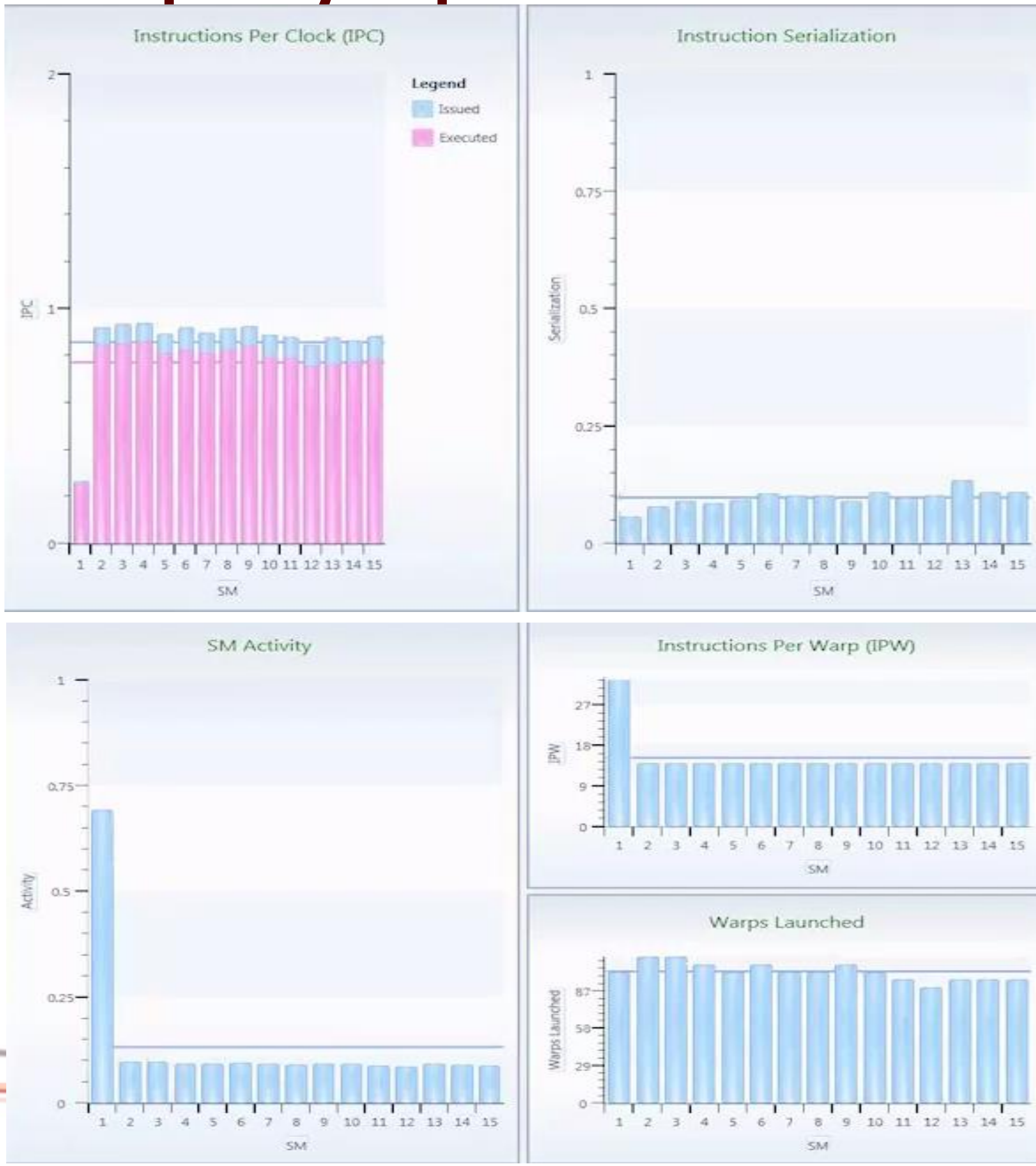
Variable	Achieved	Theoretical	Device Limit
Occupancy Per SM			
Active Blocks		8	8
Active Warps	36.[illegible]3	48	48
Active Threads		1536	1536
Occupancy	75.06 %	100.00 %	100.00 %
Warps			
Threads/Block		192	1024
Warps/Block		6	32
Block Limit		8	8
Registers			
Registers/Thread		4	63
Registers/Block		768	32768
Block Limit		42	8
Shared Memory			
Shared Memory/Block		0	49152
Block Limit		8	8

Occupancy Considerations

- Low occupancy is not always a bad thing.
  - If you are aiming for concurrent kernel operation then overall capacity could be distributed between different kernels.
  - Unused SMs would be used by other kernels.

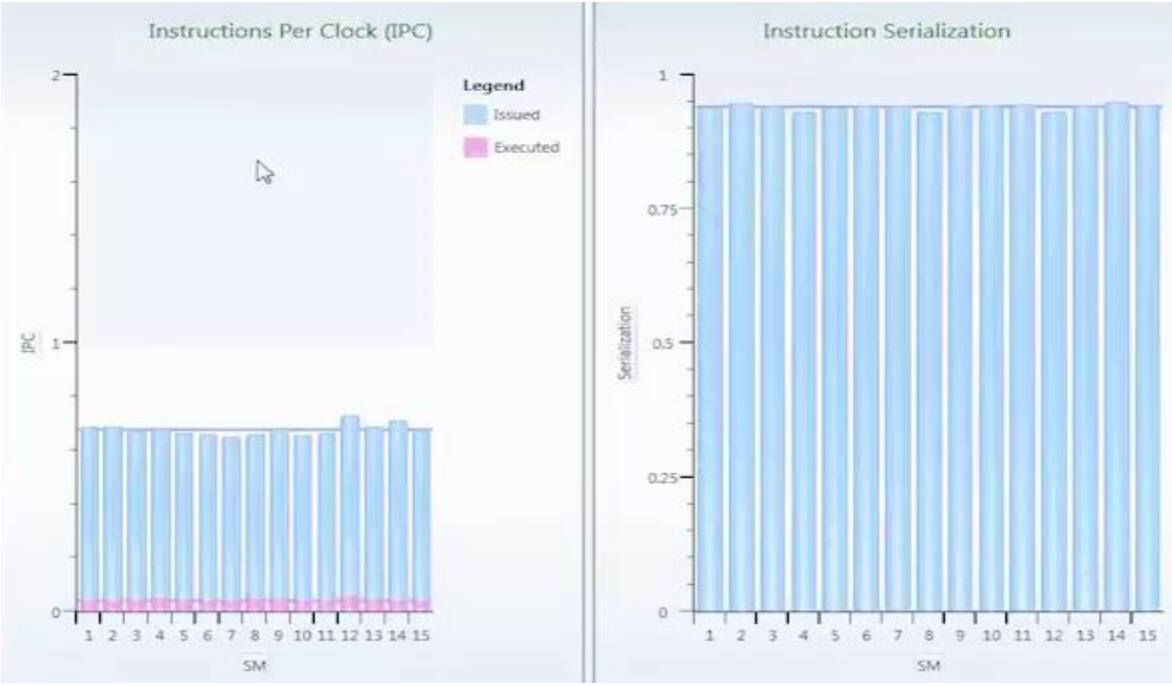
```
// Device Code
__global__ void VecAdd_A(const float* A, const float* B, float* C)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    C[i] = A[i] + B[i];
}

__global__ void VecAdd_B(const float* A, const float* B, float* C)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if (blockIdx.x == 0)
    {
        if (threadIdx.x == 0)
        {
            for (int j = 0; j < blockDim.x; j++)
            {
                C[i+j] = A[i+j] + B[i+j];
            }
        }
    }
    else
    {
        C[i] = A[i] + B[i];
    }
}
```

Occupancy Experiment- Kernel VecAdd_B

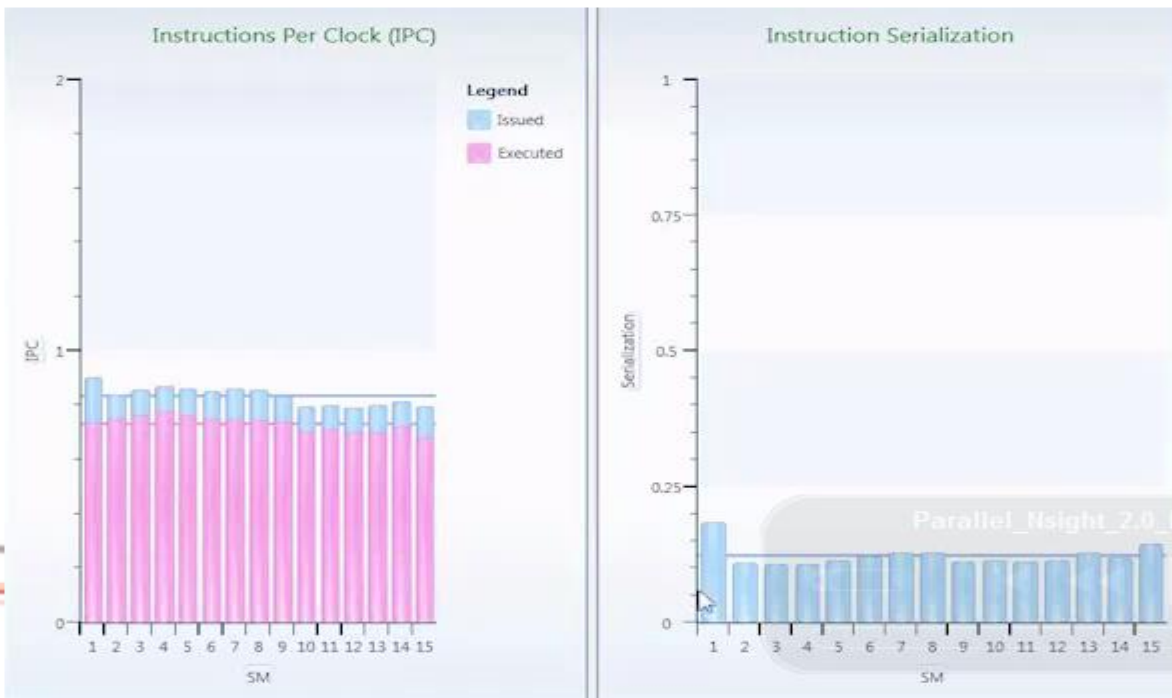


```
__global__ void VecAdd_A(const float* A, const float* B, float* C)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    C[i] = A[i] + B[i];
}
```

```
__global__ void VecAdd_C(const float* A, const float* B, float* C)
{
    int i = blockDim.x * threadIdx.x + blockIdx.x;
    C[i] = A[i] + B[i];
}
```



VecAdd_C



VecAdd_A

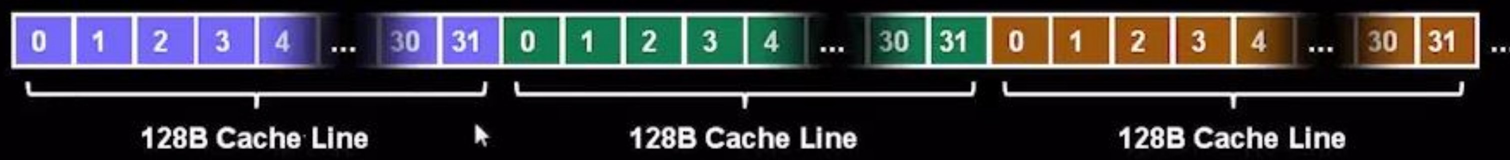
Instruction Serialization

- "instruction replays" for warp's threads
- Occurs when threads in a warp execute/issue the same instruction *after* each other instead of in parallel
- Think of it as “replaying” the same instruction for different threads in a warp
- Some causes:
  - Shared memory bank conflicts
  - Constant memory access inefficiency

Serialization: Memory Access Pattern

- **VecAdd_A()**

```
int i = blockDim.x * blockIdx.x + threadIdx.x;
C[i] = A[i] + B[i];
```



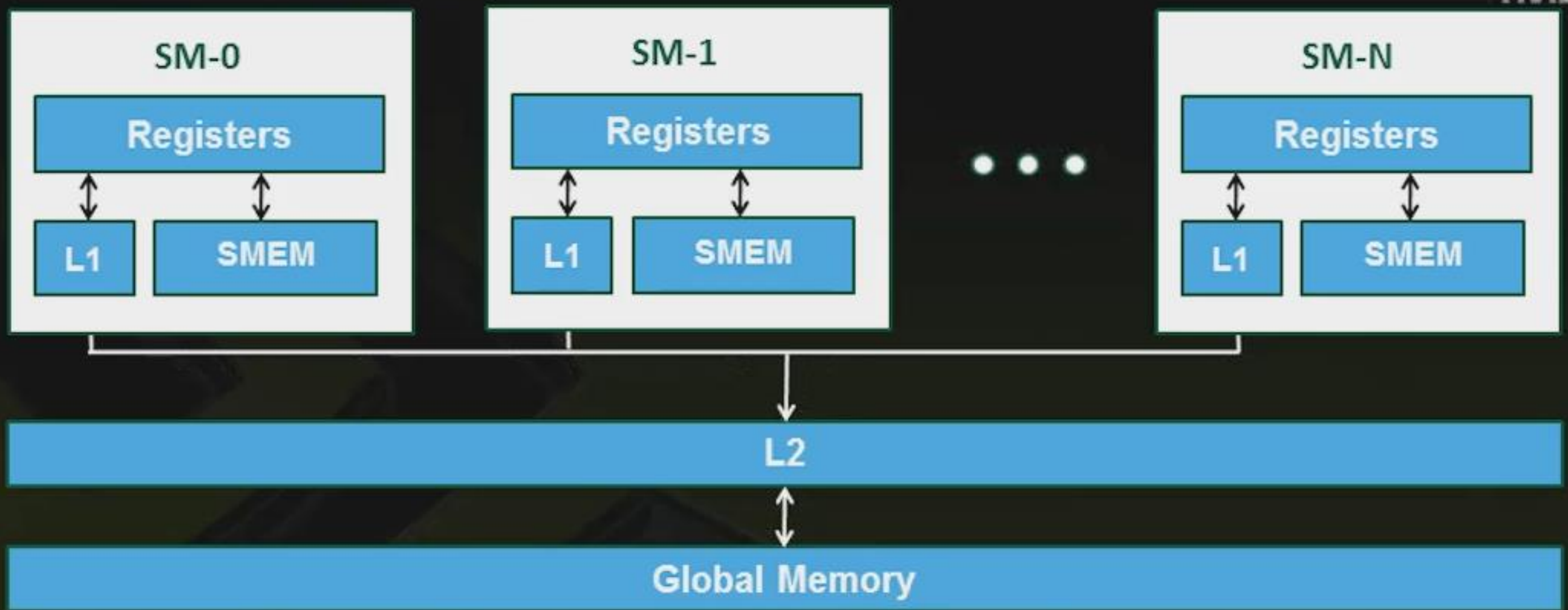
Each Memory Access: 1 Instruction Executed 1 Instruction Issued

- **VecAdd_C()**

```
int i = blockDim.x * threadIdx.x + blockIdx.x;
C[i] = A[i] + B[i];
```

Each Memory Access: 1 Instruction Executed <u>32</u> Instructions Issued

Fermi Architecture



- SM – Streaming multi-processors with multiple processing cores
  - Each SM contains 32 processing cores
  - Execute in a Single Instruction Multiple Thread (SIMT) fashion
  - Up to 16 SMs on a card for a maximum of 512 compute cores

Context Switch

- A **grid** is composed of blocks which are completely independent
- A **block** is composed of threads which can communicate within their own block
- 32 threads form a warp
  - **Instructions are issued per warp**
- If an operand is not ready the warp will stall
  - Context switch between warps when stalled
  - Context switch must be very fast

Context Switch

- Registers and shared memory are allocated for a block as long as that block is active
  - Once a block is active it will stay active until all threads in that block have completed
  - Context switching is very fast because registers and shared memory do not need to be saved and restored
- **Goal: Have enough transactions in flight to saturate the memory bus**
  - Latency can be hidden by having more transactions in flight
  - Increase active threads or Instruction Level Parallelism (ILP)
- Fermi can have up to 48 active warps per SM (1536 threads)