

Practicle 5

Debugging a CUDA application

30/01/19

Abstract

1 Nsight

Nsight is a development environment for CUDA. This tool is integrated into Visual Studio. When you run your program through Nsight you can reach breakpoint into your kernel. Frist you need to specify your application to generate GPU Debug Information. Go to the Property panel of your project and go on the Device section of CUDA C/C++. Here you can enable the Generate GPU Debug Information (Yes (-G)). Now we can use breakpoint into our kernel. Put one breakpoint after the index computation and run your program. You can read in debug mode the value of your index for the first thread of the first block. Be carefull, adding the -G debug information add check on your kernel and use more registers. Sometime you may reduce the number of thread per block for debugging.

1.1 Warp Info

Most of the time debug only the first thread solve problem because we use SIMD architecture. However, if there is a branch in the kernel you may want to debug another thread. On the Nsight panel, open Windows/Warp Info.

Context	Grid ID	Shader Type	Shader Info	Threads	PC	Active Mask	Status
21023bb1e00	00000003	Compute	CTA: (0, 0, 0), Thread: (0, 0, 0)		310b02f8	FFFFFFFF	None
21023bb1e00	00000003	Compute	CTA: (0, 0, 0), Thread: (32, 0, 0)		310b02f8	FFFFFFFF	None
21023bb1e00	00000003	Compute	CTA: (0, 0, 0), Thread: (64, 0, 0)		310b02f8	FFFFFFFF	None
21023bb1e00	00000003	Compute	CTA: (0, 0, 0), Thread: (96, 0, 0)		310b02f8	FFFFFFFF	None
21023bb1e00	00000003	Compute	CTA: (0, 0, 0), Thread: (128, 0, 0)		310b02f8	FFFFFFFF	None
21023bb1e00	00000003	Compute	CTA: (0, 0, 0), Thread: (160, 0, 0)		310b02f8	FFFFFFFF	None
21023bb1e00	00000003	Compute	CTA: (0, 0, 0), Thread: (192, 0, 0)		310b02f8	FFFFFFFF	None
21023bb1e00	00000003	Compute	CTA: (0, 0, 0), Thread: (224, 0, 0)		310b02f8	FFFFFFFF	None
21023bb1e00	00000003	Compute	CTA: (1, 0, 0), Thread: (0, 0, 0)		310b02f8	FFFFFFFF	Breakpoint
21023bb1e00	00000003	Compute	CTA: (1, 0, 0), Thread: (32, 0, 0)		310b02f8	FFFFFFFF	Breakpoint
21023bb1e00	00000003	Compute	CTA: (1, 0, 0), Thread: (64, 0, 0)		310b02f8	FFFFFFFF	Breakpoint
21023bb1e00	00000003	Compute	CTA: (1, 0, 0), Thread: (96, 0, 0)		310b02f8	FFFFFFFF	Breakpoint
21023bb1e00	00000003	Compute	CTA: (1, 0, 0), Thread: (128, 0, 0)		310b02f8	FFFFFFFF	Breakpoint
21023bb1e00	00000003	Compute	CTA: (1, 0, 0), Thread: (160, 0, 0)		310b02f8	FFFFFFFF	Breakpoint
21023bb1e00	00000003	Compute	CTA: (1, 0, 0), Thread: (192, 0, 0)		310b02f8	FFFFFFFF	Breakpoint
21023bb1e00	00000003	Compute	CTA: (1, 0, 0), Thread: (224, 0, 0)		310b02f8	FFFFFFFF	Breakpoint
21023bb1e00	00000003	Compute	CTA: (2, 0, 0), Thread: (0, 0, 0)		310b02a8	FFFFFFFF	None
21023bb1e00	00000003	Compute	CTA: (2, 0, 0), Thread: (32, 0, 0)		310b02a8	FFFFFFFF	None
21023bb1e00	00000003	Compute	CTA: (2, 0, 0), Thread: (64, 0, 0)		310b02a8	FFFFFFFF	None
21023bb1e00	00000003	Compute	CTA: (2, 0, 0), Thread: (96, 0, 0)		310b02a8	FFFFFFFF	None
21023bb1e00	00000003	Compute	CTA: (2, 0, 0), Thread: (128, 0, 0)		310b02a8	FFFFFFFF	None
21023bb1e00	00000003	Compute	CTA: (2, 0, 0), Thread: (160, 0, 0)		310b02a8	FFFFFFFF	None
21023bb1e00	00000003	Compute	CTA: (2, 0, 0), Thread: (192, 0, 0)		310b02a8	FFFFFFFF	None
21023bb1e00	00000003	Compute	CTA: (2, 0, 0), Thread: (224, 0, 0)		310b02a8	FFFFFFFF	None

Figure 1: Warp Info

You can see several color infos for each warps. If it is red is beause the execution reach the breakpoint. If it is green it's mean the thread is active. There is 8 threads states.

	Color	Thread State
	Gray	Inactive
	Forest Green	Active
	Light Sea Green	At Barrier
	Red	At Breakpoint
	Orange	At Assert
	Dark Red	At Exception
	Dark Gray	Not Launched
	Light Gray	Exited

Figure 2: Warp Info

1.2 Lanes

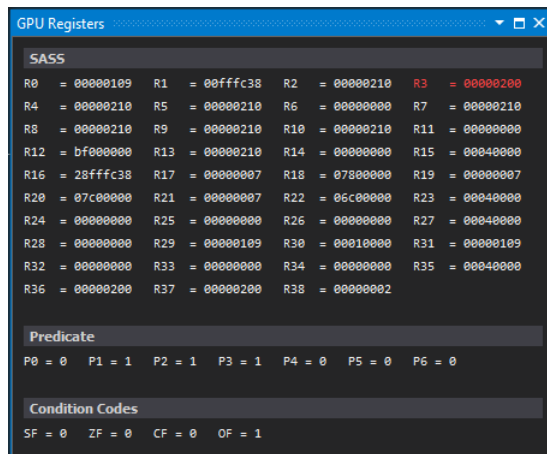
The lane section allow us to debug threads for the active warp. We can find the same informations as the warp info. If you change the current warp on the warp info it will impact this window too.

Lane	Thread Index	Status	PC	Exception
0	(0, 0, 0)	Active	000bf678	None
1	(1, 0, 0)	Active	000bf678	None
2	(2, 0, 0)	Active	000bf678	None
3	(3, 0, 0)	Active	000bf678	None
4	(4, 0, 0)	Active	000bf678	None
5	(5, 0, 0)	Active	000bf678	None
6	(6, 0, 0)	Active	000bf678	None
7	(7, 0, 0)	Active	000bf678	None
8	(8, 0, 0)	Active	000bf678	None
9	(9, 0, 0)	Breakpoint	000bf678	None
10	(10, 0, 0)	Active	000bf678	None
11	(11, 0, 0)	Active	000bf678	None
12	(12, 0, 0)	Active	000bf678	None
13	(13, 0, 0)	Active	000bf678	None
14	(14, 0, 0)	Active	000bf678	None
15	(15, 0, 0)	Active	000bf678	None
16	(16, 0, 0)	Active	000bf678	None
17	(17, 0, 0)	Active	000bf678	None
18	(18, 0, 0)	Active	000bf678	None
19	(19, 0, 0)	Active	000bf678	None
20	(20, 0, 0)	Active	000bf678	None
21	(21, 0, 0)	Active	000bf678	None
22	(22, 0, 0)	Active	000bf678	None
23	(23, 0, 0)	Active	000bf678	None
24	(24, 0, 0)	Active	000bf678	None
25	(25, 0, 0)	Active	000bf678	None
26	(26, 0, 0)	Active	000bf678	None
27	(27, 0, 0)	Active	000bf678	None
28	(28, 0, 0)	Active	000bf678	None
29	(29, 0, 0)	Active	000bf678	None
30	(30, 0, 0)	Active	000bf678	None
31	(31, 0, 0)	Active	000bf678	None

Figure 3: Warp Info

1.3 GPU Registers

Sometime you may need to see the disassembly code for debugging (Right clic on your code/Go to disassembly). You can follow the value stored into the GPU register on this last window.



The screenshot shows a window titled "GPU Registers" with a dark background. It contains three sections: "SASS", "Predicate", and "Condition Codes".

SASS			
R0 = 00000109	R1 = 00ffffc38	R2 = 00000210	R3 = 00000200
R4 = 00000210	R5 = 00000210	R6 = 00000000	R7 = 00000210
R8 = 00000210	R9 = 00000210	R10 = 00000210	R11 = 00000000
R12 = bf000000	R13 = 00000210	R14 = 00000000	R15 = 00040000
R16 = 28fffc38	R17 = 00000007	R18 = 07800000	R19 = 00000007
R20 = 07c00000	R21 = 00000007	R22 = 06c00000	R23 = 00040000
R24 = 00000000	R25 = 00000000	R26 = 00000000	R27 = 00040000
R28 = 00000000	R29 = 00000109	R30 = 00010000	R31 = 00000109
R32 = 00000000	R33 = 00000000	R34 = 00000000	R35 = 00040000
R36 = 00000200	R37 = 00000200	R38 = 00000002	

Predicate						
P0 = 0	P1 = 1	P2 = 1	P3 = 1	P4 = 0	P5 = 0	P6 = 0

Condition Codes			
SF = 0	ZF = 0	CF = 0	OF = 1

Figure 4: Warp Info

2 Test limits of your program

2.1 Reach the maximum of the callstack

We see on the last practical how to grab an error. Now we'll try to get errors. Create a recursive function called more than 100 times and look at the error thrown. It is because you don't have enough memory for your stack. You can get your device stack size in this way:

```
size_t value;
cudaDeviceGetLimit(&value, cudaLimitStackSize);
```

And if you want to modify this limit just call:

```
cudaDeviceSetLimit(cudaLimitStackSize, value);
```

2.2 Branches

Into one of your simple kernel (like the clear color), add a if statement to do the job only for the half of the threadIdx.

```
if (threadIdx.x < 16) {
    for (int i = index; i < w*h; i += stride) {
        image[i] = backgroundColor;
    }
}
```

On the Warp Info panel we can see the half of the thread are disabled.

3 Performance analysis

On the Nsight panel you can start the Performance analysis tool. You just need to check the radio button Profile CUDA Application and on the Application Control panel click on Launch. This action will profile each kernel of your application. If needed you can specify which kernel you want to profile.

3.1 Latency and Occupancy

Latency systems oriented like CPU use speculative fetching, branch prediction and large caches to avoid the delay to get data computed. For SIMT architecture we talk about occupancy. It is the measure of thread parallelism. If we look into a thread we are talking about Instruction-level Parallelism. In this example a and d can be computed at the same time to avoid latency:

```
a = b + c
d = e + f
g = a + d
```

Remember that more the system use his threads active more the application is efficient.

On the result of the performance analysis tool you can see on the CUDA launches all informations relatives to your kernel.

Identify heavy kernels and try to split them into smaller process. For example we can create a kernel named computeStartRay to compute an image that store data instead of pixel color. We chose an camera origin on (0, 0, 0) so the Red value can store the X value of the ray direction, same for Green and Blue.