Introduction to Warp Divergence:

As we discussed **threads** in a **warp** are going to **execute the same instruction**. But what if **we code our program** in a way that if **will force some threads in the warp to execute different instructions**.

**EX:**

Text

Description automatically generated with medium confidenceWe have a **grid** with **blocks** which have **32** threads.

There will be only **one** **warp** to execute. And this **grid** will execute the following condition checks in the **kernel**. The **threads** with threadIdx.x values **less than 16** will execute statements in the if block, and the **threads** with threadIdx.x values **greater than 16** will execute the else block. But we know **warps** execute in **SIMT** fashion. So, every **thread** in this **warp** has to **execute** **the same instruction**.

If **threads** in a **warp** **diverge** then that **warp** **serially execute** each **branch path** and disabling **threads** that do not take that path. So, **warp** will execute the “if block” while disabling the second half of the **threads** inside the **warp**. Then, it will **execute the instructions** in the “else block**”** while disabling the first half of the **warp**. So, **if we have multiple condition checks** which causes ***warp to diverge*** then it ***will be a significant performance penalty***.

**Another EX:**

We are going to execute a **kernel** with the code bellow.

Text

Description automatically generatedWe have a condition check which forces **threads** **with odd** threadIdx.x values to follow different path than the **threads** **with** **even** threadIdx.x values. So, all the **threads** in the **warp** **with** **even** threadIdx.x values will execute “if statement”, and all the **threads** in the **warp** **with odd** threadIdx.x values will execute the “else statement”. To adhere **SIMT** *execution convention*, **CUDA** runtime will force each **thread** to **sequentially go through all the statements in both** “*if and else* ***block”*** while **disabling** the **threads** that ***does not take that path logically***.

Such an execution for just **4** **threads** in that particular **warp** is shown here.

Chart

Description automatically generated

* Blue colored boxes represent **coherent code** which are **going to execute by all the threads in the warp**.
* Only **threads** **with even** threadIdx.x values, which in this case **threads** with threadIdx.x values of 0 and 2, will execute the **green colored** instructions or “**if clause**”. Moreover, **threads** with threadIdx.x values of 1 and 3 will *stall(stop)* its execution.

A picture containing chart

Description automatically generatedAnd when the **threads** with threadIdx.x values of 1 and 3 execute the “**else block”,** the **threads** with threadIdx.x values of 0 and 2 have to *stall* the execution. In this example, the amount of parallelism in the **warp** execution was cut by **half**. Only 16 **threads** were actively executing at a time while other 16 were disable.

**WITH MORE CONDITIONAL BRANCHES THE LOSS OF PARALLELISM WOULD BE EVEN GREATER.**

Be aware in the presence of **control flow statements** like if and else statements and switch statements which gives us a hint of divergent code. However, u**sing control flow *statements is not always result in a divergent*** ***warp***.

**WARP** **IS DIVERGED WHEN THERE ARE MULTIPLE PATHS OF EXECUTION WITHIN THE SAME WARP**.

So**, condition checks which result in all the threads in a warp to execute in same path, will not induce any warp divergence**.

**EX:**

Text

Description automatically generated with low confidenceHere "**tid**" represent a threadIdx.x value for a single dimensional block. Here if we take 64 as our **block** size CUDA runtime will allocate 2 **warps** to execute this **thread** block. The first 32 **threads** in our **thread** **block** will have tid from 0 to 31, thus **tid**/32 will be 0.

A picture containing diagram

Description automatically generated

So, the first 32 **threads** or “the first **warp”** for our **thread** **block** will execute the **if block** and none of the **threads** of the **first warp** will execute the **else block**.

Graphical user interface

Description automatically generatedAnd for the second warp, the 32 **threads tid values** will be 32 to 63, making **tid**/32 equal to 1 and all the **threads** in the **second warp** are going to execute in the else block. So, the **first** **warp** will execute **if block** and **second warp** will execute else block. Thus, even if we have a condition check, there is no **warp** divergence.

Shape

Description automatically generated with medium confidence If we change the condition, then *first half* of **first** **warp** will execute the **if statement** and *second half* of **first warp** will execute the **else statement.** And this is true for the **second** **warp** as well. If we use **conditions** in this way, there is ***warp divergence*** and **PERFORMANCE WILL BE DEGRADED**. Be vigilant when you are using condition checks in a **kernel**.

Always try to find a proper condition check that doesn’t generate ***warp divergence.***

You can measure whether your **kernel** have **divergent code** or not by using performance metrics with the **nvprof profiling tool**.

Performance metrics are measurements of different properties of your **kernel**. There are metrics called **branch efficiency** which indicates the presence of a **divergent code** in a **kernel**.

**Branch** is a **path of execution** for the **threads** in a **warp**.

Text

Description automatically generated with medium confidence**Ex**, If we have a condition that forces **threads** with odd threadIdx.x values to execute the **if block** and **threads** with even threadIdx.x values to execute the **else block**, then we have **two branches** here: an **If Branch** and an **Else Branch**. If we have a code which forces **threads** within the same **warp** to execute in **3** **different paths**, then we have **3 branches**. Keep in mind the **threads** within a single **warp** should follow **different path**. Otherwise, there is no **branch divergence**.

**Branch efficiency** is defined as the ratio of **non-divergent branch** to **total branches**. It can be calculated using following equation:

Ex2:

Text, letter

Description automatically generatedIf we consider this condition check, **half** of the **warp** will execute the **if block** and other half will execute the **else block**. So we have two branches and we need to consider one of them as the **main execution path**. Thus, we have one divergent branch, and the branch efficiency will be:

Branch efficiency can be measure for a **kernel** using the **nvprof profiling tool**.

To compile the code (kernel.cu) in your computer (In Windows), you’ll have to make sure you can use the nvcc compiler. Also, don’t forget to add the cuda\_common.cu and cuda\_common.cuh file from previous examples or the “Common Header and Implementation Files” folder.

Moreover, you’ll have to add this to your environment variables PATH: Link to the [StackOverflow](https://stackoverflow.com/questions/8125826/error-compiling-cuda-from-command-prompt/8126263#8126263)

C:\Program Files\Microsoft Visual Studio\2022\Community\VC\Tools\MSVC\14.33.31629\bin\Hostx64\x64

Finally, you need to add the following address to the previous environment PATH:

Link to [StackOverflow](https://stackoverflow.com/questions/61865257/nvprof-command-error-cupti64-102-dll-was-not-found)

After we compile the code using the following command:

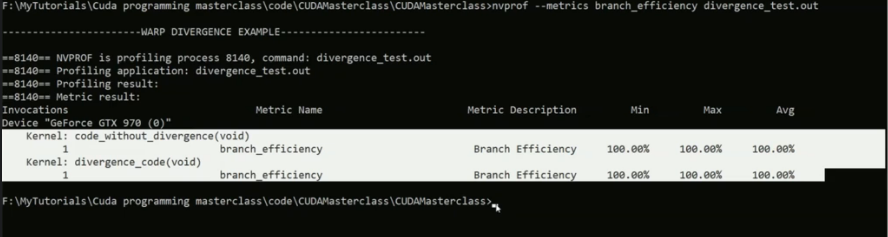
nvcc -o divergence\_test.out 2\_Warp\_Divergency.cu

We will get warnings about unused variables. This is because we have not perform any effective task in our kernel (we only assigned values to **a** and **b**).

After that, we will profile our program with the **nvprof profiling tool** with the following command:

nvprof --metrics branch\_efficiency divergence\_test.out

In the previous, we have specified **performance metric** using the “--metrics” option and we will observe the branch efficiency metrics as specified in the command. Notice that we need to refer our executable file we create before.

Here are the results we should be looking based on the course material:

The reason why we don’t see a performance difference between both kernels is due that nvcc optimizes our divergent code for us, so it runs without divergence.

If we want to compile our code without any **compiler optimizations**, We should type the following code:

Nvcc -G -o divergence\_test.out 2\_Warp\_Divergency.cu

Text

Description automatically generatedAnd then run the **nvprof profiling tool** again. You should get the following results:

Now, when we calculate the theorical value of branch efficiency as before (50%), it won’t match the results we see there.

I got the following Warning after running the previous command:

Warning: Skipping profiling on device 0 since profiling is not

supported on devices with compute capability 7.5 and higher. Use NVIDIA Nsight Compute for GPU profiling and NVIDIA Nsight Systems for GPU tracing and CPU sampling. Refer https://developer.nvidia.com/tools-overview for more details.

Due to that, I’ll attempt to use the Nsight Computer tool.

Here are the steps I’m taking:

1. I installed the NVIDIA Nsight Compute installer from this [website](https://developer.nvidia.com/gameworksdownload#?dn=nsight-compute-2022-3-0)

Note: I already had it installed so I won’t need to install it again.

Also, the file is 460 Mb on size.

1. I followed the QuickStart guide from this [website](https://docs.nvidia.com/nsight-compute/2022.3/NsightCompute/index.html)
2. Due to the being too time consuming, I decided to not continue working on this until we get to more advanced chapters since this tool is for optimization.