Introduction to Warps:

When launching a kernel, it seems that all the threads in the kernel runs in parallel.

From a logical point of view, this is true. But from the **hardware point of view**, not all the threads physically execute in parallel at the same time.

Text

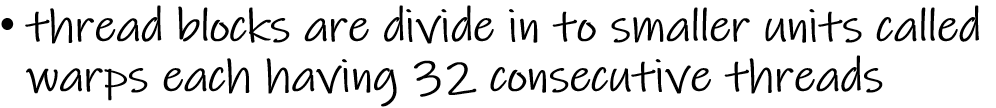
Description automatically generated with medium confidence Ex: Let say we have a grid with 1 million threads with each thread block having 512 threads.

Company name

Description automatically generated with low confidenceNow from the **software point of view** we assume that all of one million threads in the grid executes parallel. But if we ran this kernel on a device with 13 **SMs** each having 128 ***cores*** so altogether we have only 13 \*128 which is 1664 cores.

So, this device can **only execute 1664 threads in parallel**. So, there is no way of running 1 million threads in parallel in this device.

Also, **one** **block** is going to run in a single **SM** and here we have only ***128 threads*** per **SM**. But since we have ***512 threads*** ***per thread block***, there is no way for parallel execution to occur in all the threads.

So, this is how we address the problem:

And all the **warps** for a given **thread block** are going to execute in a single **SM**.

**Ex:**

Graphical user interface

Description automatically generated with low confidence

A picture containing text

Description automatically generated**Sixteen warps** will contain all the threads. And since we have **128 cores** per **SMs**, **only 4 warps** can execute at a given time **(4\*32)**.

Now there is *no particular order which warp is going to execute next*. That is **dependent on things like readiness of memory for threads in those warps**.

Here we can define **warps** as the **basic unit of execution** in a **SM**. And when we launch a **grid**, the **thread blocks** are going to **distributed among** **SMs**. Once a **thread block** is ***schedule*** (assigned?) to a **SM**, **threads** in that **thread block** are **further partitioned** into **warps**.

A picture containing icon

Description automatically generatedA **warp** consist of **thirty-two consecutive threads** and **all threads** in a **warp** executes in a SIMT fashion. That is all the **threads** **execute the same instruction** and each **thread** conducts that operation on its own private data. A thread block can be configured in 1D, 2D or 3D. However, from the **hardware point of view**, all **threads** are arranged in a Single Dimension. Here, each thread has a unique ID.

**Ex:** We have 1D block with 128 threads in X dimension

A picture containing graphical user interface

Description automatically generated Those threads will be divided into four warps.

Warp [0] will have threads with threadIdx.x values from **0 to 31**

Warp [1] will have threads with threadIdx.x values from **32 to 63**

Warp [2] will have threads with threadIdx.x values from **64 to 95**

Warp [3] will have threads with threadIdx.x values from **96 to 127**

Graphical user interface

Description automatically generatedIf we had a **2D thread block** with **64 threads** in X-dimension and **2 threads** in Y-dimension

A picture containing graphical user interface

Description automatically generated

From the hardware point of view threads will be divided in to **4** **warps** in a **single dimension**.

Graphical user interface, website

Description automatically generated

Now, we have a **2D thread block** with **40** **threads** in X-dimension and **2 threads** in Y-dimension. From the application perspective there are **80** **threads** laid out in **two-dimensional grid**.

Diagram

Description automatically generated with low confidenceFrom the hardware point of view we have **80** **threads** in X-dimension. One can suggest that the hardware will allocate **3** **warps** for this **thread block** which will result in a total of **80** ***hardware*** **threads** to support **80** ***software*** **threads** in this way. If so, only **16** **threads** will be in the **active state in the final warp**.

**THE PREVIOUS IS NOT TRUE**. **Warps** exists within a **thread block**. If we do it in this way, the ***second*** **warp** is going to have **threads** from **two** **thread blocks**. So here if we consider it this way, we are suggesting that **threads** belong to **two** **thread blocks** residing in **one** **warp** which is **AN INACCURATE STATEMENT**.

**What really happens here is:**

To handle **80** ***software*** **threads** in this case **CUDA** runtime will allocate **4** **warps** or **128** **threads** as shown below.

A picture containing waterfall chart

Description automatically generated

In this case only **8** **threads** in both **second** and **fourth** **warps** are going to be in ***active state***.

All other **threads** for these **warps** are going to be in ***inactive state*** when our device execute this **thread block**.

**Warp** size is **32** and even when we want to run a **thread block** with a **single** **thread**, **CUDA** runtime still will assign a **warp** to it which means it will have **32** **threads**. Thus, only **1** **thread** will be in ***active state*** and all other **31** **threads** will be in ***inactive state***.

The problem here is that, even though those are **inactive** **threads** (those threads will not execute instructions) it still accounts it when allocating resources like shared memory for the block (we will have wraps that almost do nothing.)

So having **inactive** **threads** in a warp will be a **great waste of resources** in **SM**. Therefore, we tend to have values which are multiples of **32** for **block’s** sizes (**thread** amount) in X-dimension, so that we can guarantee that all the **threads** in a warp are going to be in **active state**. Let’s do a simple example to make a sense of the previous.

**Ex:**

Launch a **kernel** with **two-dimensional grid** and each is going to printout threadIdx, blockIdx and **warp** **index values**. There are not any built-in variables (like blockIdx) to indicate the **warp index**. But ***we can easily calculate this value by dividing*** **threadIdx** **value by warp size which is** **32**. So, this **value represents index of a warp in a block**. So, for a **single thread block** we are going to **have different warp value for each 32 consecutive threads**. But for different **thread blocks** there will be **warps** with **same warp index**.

Timeline

Description automatically generated with low confidenceShape

Description automatically generatedOur grid has **2** **blocks** in X-dimension and **2** **blocks** in Y-dimension. And each **block** is going to have **42** **threads** in X-dimension. So, all together we are going to launch this **kernel** with **168** **threads**. **42** is a far from ideal value we should have as a **block** size. But we’ll use it to demonstrate something.

Now let's look at how our warp allocation may look like.

Recall that warp cannot have **threads** from **2** **thread blocks**. So, to handle **168** software **threads** in this case, we are going to have **256** hardware threads or **8 warps**. And here we have **4** **warps** with only **10** **active** **threads** in each. Other threads in these warps is going to be in inactive state. Ok, let's printout above mention value for this grid.