Resources partitioning and latency hiding Part 2:

**SM** are resources constraint environments. **Excessive usage of resources** like **registers** and **shared memory** from a **single** **thread**, **causes** **SM** to have **smaller number of active warps residing on that SM at a given time**, resulting in **underutilization of computing resources**.

We are going to look at another concept called **latency hiding** which **highlights the need of enough active warps to utilize the computation power optimally**.

**Latency (instruction latency):**

**Latency** = **number of clock cycles between:**

1. instructions being issued
2. instructions being completed.

When the ***controlling unit*** inside the core **issues an instruction** to do an **arithmetic operation** to the **ALU(Arithmetic Logic Unit)**, the **ALU** ***might take some clock cycles to complete that issued instruction*** and ***in case of memory operations***, ***the number of clock cycles to complete that issue instruction*** is the **number of clock cycles between** the **memory instruction** **being issued** and ***the memory to arrive at the destination***.

The table bellow is **part of the analysis** of the **GPGPU pipeline latency** done by Michael Anderson and John Lucas for the **embedded system microarchitecture symposium** at Berlin.

Table

Description automatically generatedIt highlights the latencies of the **integer**, **logic**, and **floating-point** operations for **4** NVIDIA microarchitectures. You can see that for Maxwell GM107 chips “***ADD/SUB” operations*** have instruction latency of **6 clock cycles**. ***The MAX/MIN, and MAD (Multiplication Add) and MUL (multiplication) operations*** have instruction latency of **12 to 13 clock cycles.** Operations **like division (Div) and the remainder(Rem)** have **instruction latency more than 200 clock cycles**. Logical operators like “AND, OR, XOR” have **instruction latency of 6 clock cycles**.

The table shows you **instruction latency** for **memory load** and **store operations** for different types of memories like **global memory**, **local memory**, **shared memory**, **texture memory** and **constant memory**.

Table

Description automatically generatedOn Global & Local Memory for **Maxwell GM107** chips, the L2 cache have a **memory latency of 194 clock cycles**. The **DRAM access has a 350 clock cycles memory latency.**

But shared memory access has only **28 clock cycles of latency** which **highlights the importance of shared memory**. So, when a **warp** is executing, if it performs an arithmetic operation that warp **will hold for 15 or so clock cycles**. This will be a performance penalty.

And *in case* of **memory operations** this ***holding time can raise to hundreds of clock cycles*** which will ***degrade the performance of our application greatly***.

Fortunately, with **CUDA latency hiding mechanisms**, if we have enough number of **eligible** **warps** residing on **SM,** we can hide this instruction latencies.

How the latency Auditing Mechanism Works:

The **execution context of each warp processed** by a **SM** are **maintain on-chip during the entire lifetime of a warp**. Therefore, switching from one execution context to another has no penalty. So, to hide the above-mentioned latency, we are going to have a large number of **eligible** **warps** so when one **warp** holds, then the **warp scheduler** can dispatch another **eligible** **warp** to the **core**.

Chart, histogram

Description automatically generatedEx:

This diagram demonstrates the execution of a **warp** in **SM** and the current execution cycle we are at.

A screenshot of a computer

Description automatically generated with medium confidenceLet's say this warp is going to execute an arithmetic operation which take **20 instruction cycles to complete**. So, as you can see our warp will execute the arithmetic operation in the **first execution cycle**

Chart, box and whisker chart

Description automatically generated

Then **it has to hold for 20 clock cycles**. So, in the next 20 clock cycles there will not be any execution in these cores.

**Chart

Description automatically generated with medium confidenceAnd in the 21st clock cycle** (after the results arrived) **execution continues.**

Chart, box and whisker chart

Description automatically generatedBut if we have more **eligible** **warps** residing in the **SM**, then we could **switch the execution context to a new** **warp**, so that the **stalling (stopped) cores** can continue the *execution* for that new **warp**. If that new **warp** also belongs to the **same kernel** as the first **warp**, the new **warp** will also hold for **20 clock cycles**, after the execution of instructions.

Diagram

Description automatically generated So, if we have 20 such **warps**, we can **switch context** with all of those **warps** and **completely occupy these computation cores** **while results are ready for the first** **warp**. And when the results for the first **warp** is ready, then we can **switch the context back to it**. So, there will not be any latency as we filled the execution cycles which first **warp** has to stall with other eligible **warps**.

So, *the question* is for a given **kernel** **how many** **warps** **we need to hide the latencies of the instructions**. The answer can be found by doing the following calculation:

If we have:

Text

Description automatically generated**SM** has 128 CORES = 4 Warps (32 cores per wrap)

We need 20 **eligible** **warp** to hide the instruction latency for one **warp.**

To hide the latency of 4 Parallel Warps, we need = 4 \* 20 = 80 **eligible** **warps**

Now, if we have:

13 **SM** 🡪So well need: Number of Eligible warps (previous example) \* (Number of SM) = 80\* 13 = 1040 eligible **warps** to hide the latency of our device.

Now we’ll learn how to calculate the number of wraps needed to hide a memory instruction latency:

**Let's consider a device with Maxwell architecture which has:**

350 cycles of DRAM latency

Now, our approach is to **find how many bytes of data can be transferred within 350** **clock cycle periods** and **then we can calculate the needed number of** **warps** for **particular kernel to hide this memory latency**.

Let's consider GTX 970 device which belongs to Maxwell microarchitecture and calculate specific values. This device has **196Gb/S DRAM bandwidth**. So first we how to **convert this bandwidth information to bytes for cycle value** since **we are considering latency in instruction cycles**. For that you have to **know the memory clock speed of the given device**.

You can find out that value by running NVDIA SMI tool with following command:

My location of the nvidia-smi.exe file is:

C:\Windows\System32\DriverStore\FileRepository\nvami.inf\_amd64\_ab82c08bd4d2bb6c

For older versions, you’ll have to execute by going to the folder where the NVIDIA SMI tool is located and run the command prompt there. In my case, I can execute the command anywhere I want to:

Nvidia-smi -a -q -d CLOCK

The results should look like this:

Graphical user interface, text, application

Description automatically generated

My results were:

A picture containing text

Description automatically generated

Max clock speed represent the frequency of memory clock or how many cycles of clock ticking per second. We have 3.6 GHz of max clock speed for memory transfer. Which means 3.6 billion clock cycles per second. So, to get to that level of maximum number of bytes which can be transferred for cycle, we can:

Divide this bandwidth from this Max clock speed. So,

Diagram

Description automatically generated with medium confidenceThe previous value represents how many bytes of data can be transfer from all the **SMs** in my device to DRAM through the DDR5 memory bus. Now we have memory latency of 350 clock cycles. So:

350 clock cycles we can transfer 350\*54 =18900 bytes between DRAN and the SM.

Text

Description automatically generated If the kernel uses **4 bytes of memory** (like one integer from memory) for each **thread**, then we need:

we’ll need 4725 **threads** to hide the memory latency. Which means 148 eligible **warps** should be reside in the device to hide the memory transfer latency:

And If we have 13 SM, then we can calculate needed **warps** per SM by:

So, each SM should have at least 12 eligible **warps** to hide the memory latency. This is how you calculate how many **warps** you need to have to hide the **arithmetic and memory latencies** and **the higher of those values indicate which latency type effect the application most**. And **depending on which instruction latency affects the application the most, we can categorize any CUDA application into two categories**:

* **Bandwidth Bound** (**Memory Bound Application**) = If memory latency are the ones which contribute most to the latency.
* **Computation Bound Applications** = If the latency of arithmetic operations effect most to the execution of application.

Once we categorize our application into one of those categories, we can first focus on optimizing the application by following specific steps to optimize that category of applications.