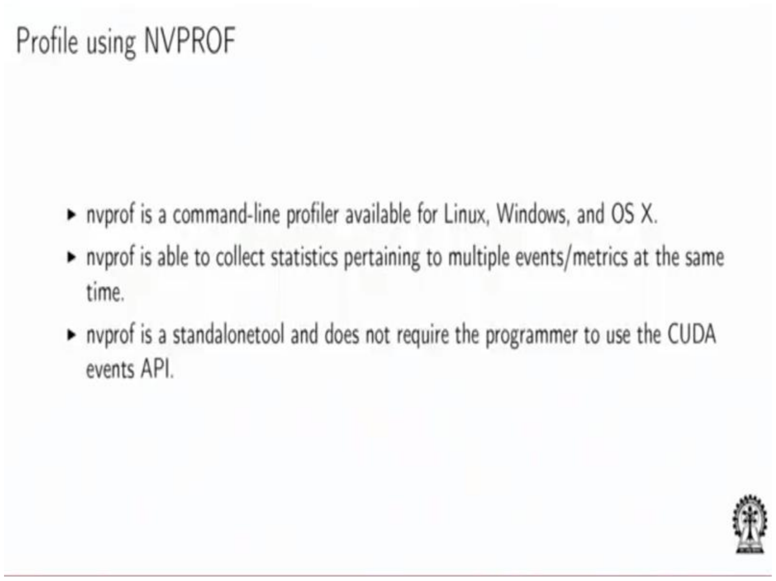
GPU Architectures and Programming

Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Module No # 05
Lecture No # 24
Memory Access Coalescing (Contd.)

(Refer Slide Time 00:25)



Profile using NVPROF

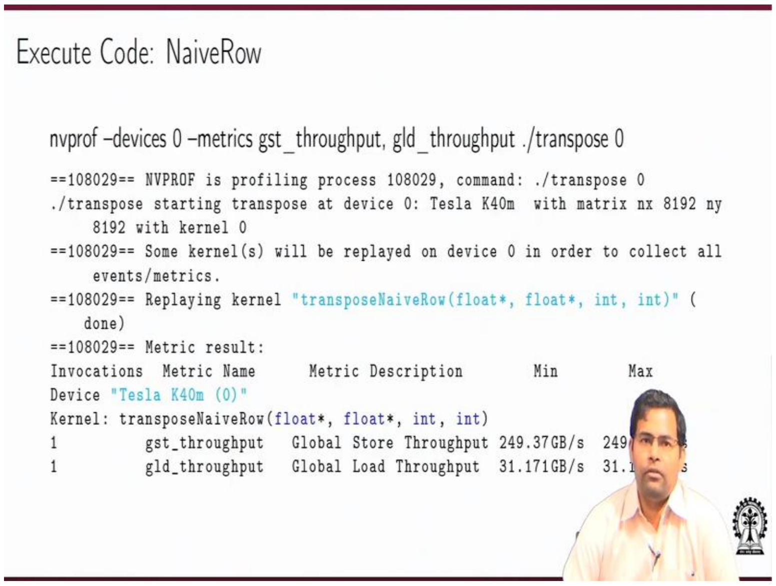
- nvprof is a command-line profiler available for Linux, Windows, and OS X.
- nvprof is able to collect statistics pertaining to multiple events/metrics at the same time.
- nvprof is a standalonetool and does not require the programmer to use the CUDA events API.

Hi welcome to the lectures on GPU architectures and programming in the previous lecture we have stared our initial discussion on profilers like NVPROF. So this is a profiler which essentially you can execute in command line it is available both in Linux, Windows, and OS X mac OS version. It can collect statistic pertaining to different program events and metrics at the same time.

It is a standard line tool and does not require the programmer to use the CUDA events API. This is important because earlier we discussed about how to use the CUDA events API which actually samples the timings from the program and using the CUDA events API you can actually catch the timings. But that actually we mix is a burden on the programmer because then you actually have to insert suitable timing codes or CUDA event calls inside your program.

So in case you do not want to use that path this is an alternate way to use it. That means you run your program under this profiler setup. We will just see some examples that how does it differ. I hope you all remember how to use the CUDA events API. You can consult or earlier lectures where you showed how I can insert this kind of suitable data types from the CUDA event API and sample out time from the CPU or the GPUs performance counters.

(Refer Slide Time 01:43)



Execute Code: NaiveRow

```
nvprof –devices 0 –metrics gst_throughput, gld_throughput ./transpose 0
==108029== NVPROF is profiling process 108029, command: ./transpose 0
./transpose starting transpose at device 0: Tesla K40m  with matrix nx 8192 ny
     8192 with kernel 0
==108029== Some kernel(s) will be replayed on device 0 in order to collect all
     events/metrics.
==108029== Replaying kernel "transposeNaiveRow(float*, float*, int, int)" (
    done)
==108029== Metric result:
Invocations  Metric Name      Metric Description        Min        Max
Device "Tesla K40m (0)"
Kernel: transposeNaiveRow(float*, float*, int, int)
1          gst_throughput   Global Store Throughput 249.37GB/s  249
1          gld_throughput   Global Load Throughput  31.171GB/s  31.1          s
```

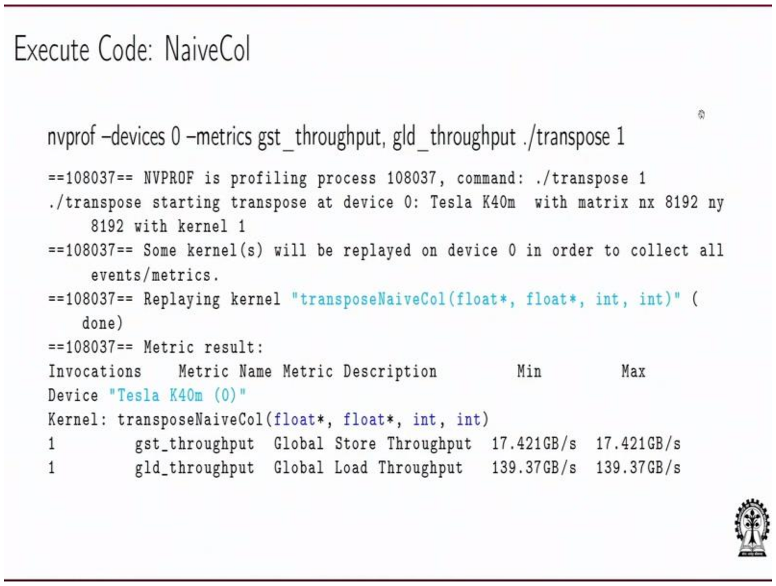
So here is an example where what we are doing is we are executing the code of transpose right the host this is the original executable. I am passing it parameter 0. So that would execute the naive row kernel right and if I pass it transfer parameter 1 it will execute the naive column kernel. If you can just see so here case 0 was for naive column case 1 was for naive column. So this is our original transpose executable. But when I executed I do not I do not execute stand alone.

I execute this with the under the NVPROF profiler I pass it these different parameters which essentially say what are the performance statistic I am interested in. So essentially it will run this kernel possibly multiple times and try to compute some statistics out of it and give me the required architectural parameters which I am interested in. So here essentially when I specify global store and global load throughput that means I am asking the profiler that ok you provide me with these statistics.

What is the number of loads and stores happening? And what is the bandwidth that is getting utilized that means how many loads and stores are happening per second in terms of how many bytes of data is getting load and loaded and stored per second or this is the minimum and maximum ranges and all that. So you can take a minute and just look into the statistic that has been provided here by the by the profiler. So it is separately giving me the global load and global store throughput values.

The number of bytes the maximum that have been written or being read per second and also minimum. So essentially this is profiler which is going to run the code multiple times and do a statistical observation out of it.

(Refer Slide Time 03:50)



Execute Code: NaiveCol

nvprof –devices 0 –metrics gst_throughput, gld_throughput ./transpose 1

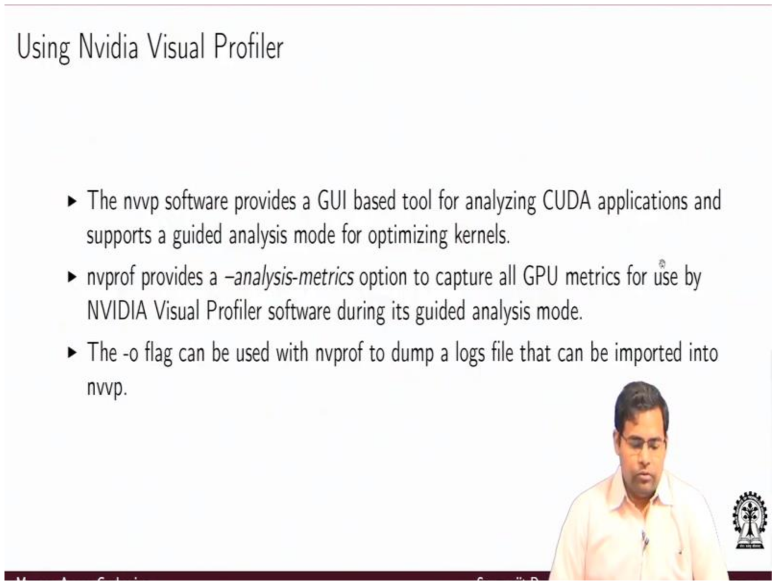
```
==108037== NVPROF is profiling process 108037, command: ./transpose 1
./transpose starting transpose at device 0: Tesla K40m  with matrix nx 8192 ny
     8192 with kernel 1
==108037== Some kernel(s) will be replayed on device 0 in order to collect all
     events/metrics.
==108037== Replaying kernel "transposeNaiveCol(float*, float*, int, int)" (
    done)
==108037== Metric result:
Invocations    Metric Name Metric Description         Min         Max
Device "Tesla K40m (0)"
Kernel: transposeNaiveCol(float*, float*, int, int)
1         gst_throughput  Global Store Throughput  17.421GB/s  17.421GB/s
1         gld_throughput  Global Load Throughput   139.37GB/s  139.37GB/s
```

So the same thing we will just so just if we look at the performance statistic once again and try to match with our original ideas. So when you are doing naive row essentially your number of loads will be much smaller than the number of stores you are doing because you are loading by row which would be coalesced and you are storing by column. And as you can see that the idea quite really matches here right because the load throughput is quite low whereas the stored throughput is quite high for the naive row implementation.

Now if you look into the other implementation of the naive column. So if you just remember our original program if the user passes that command I mean the command line option of 1 with the executable transpose. Then that one will get into the RV1 part right the RV met the RV part and that would transfer to I kernel which will indirectly get into the switch part switch statement and it will actually give the function pointer to the kernel it will make it point to the kernel for doing the naive column execution right.

So again when I execute that code with this option 1 under the NVPROF profiler I again get the load and store throughput. So now since I am doing a now a naive column implementation so I am loading by column that means that would be too many loads and the number of stored would be much smaller because the store is now coalesced and it gets reflected right. As you can see that the throughput for the store is much smaller whereas the throughput for load is much higher.

(Refer Slide Time 05:34)



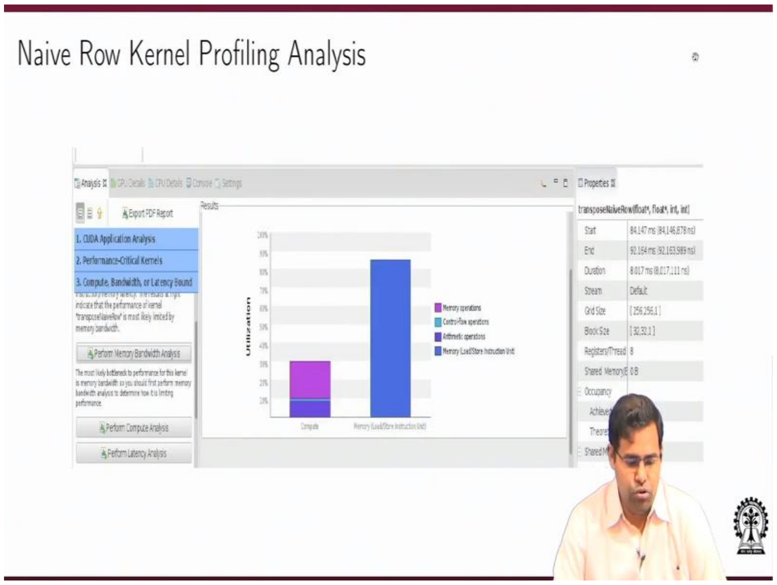
Using Nvidia Visual Profiler

- The nvvp software provides a GUI based tool for analyzing CUDA applications and supports a guided analysis mode for optimizing kernels.
- nvprof provides a *–analysis-metrics* option to capture all GPU metrics for use by NVIDIA Visual Profiler software during its guided analysis mode.
- The -o flag can be used with nvprof to dump a logs file that can be imported into nvvp.

Now with a I mean the statistics that gets collected by the NVPROF they can actually be rendered into a nice GUI based tool called NVPP NVIDIA visual profiler. So this profiler this software provides a GUI based tool for analyzing CUDA applications and it supports a guided analysis more for optimizing across kernels. So I can actually get the statistics from the NVPROF. The NVPROF provides the minus analyses minus matrix option when I execute NVPROF which can capture all the GPU matrix for use and this statistic will be kept actually where stored in a format which is usable by the visual profiler.

So next when I run the visual profiler it can render the statistic in a nice manner and it can help me to analyze the performance from the GUI right. Now of course I can have to use this minus of flag with the NVPROF dump a log file and that has to be imported to NVPP. So you can just learn to use this thing.

(Refer Slide Time 06:44)



Naive Row Kernel Profiling Analysis

indicate that the performance of kernel "transposeNaiveRow" is most likely limited by memory bandwidth.

The most likely bottleneck to performance for this kernel is memory bandwidth so you should first perform memory bandwidth analysis to determine how it is limiting performance.

transposeNaiveRow(float*, float*, int, int)

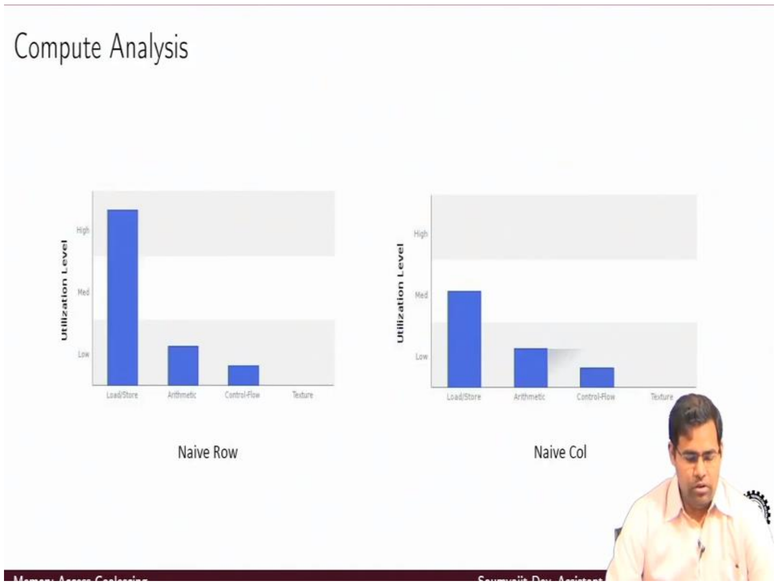
Start	84.147 ms (84,146,878 ns)
End	92.164 ms (92,163,989 ns)
Duration	8.017 ms (8,017,111 ns)
Stream	Default
Grid Size	[256,256,1]
Block Size	[32,32,1]
Registers/Thread	8
Shared Memory/B	0 B

We will just show that how when we run this original NVPROF commands we run this in NVPROF command the outputs can actually be rendered through NVPP and through this kind of a nice UI. So this is statistic profiling statistics for the naive row kernel and as we can see we the exact timings for the start end their provided. And we also have the execution duration some 8.017 millisecond and similarly and also a the statistic we have asked here is for the different utilizations.

So compute utilization the memory load store utilization they are actually plotted nicely. Since this is a less compute intensive but more communication intensive load store intensive kernel and that can quickly be gouged from this statistic chart that has been provided right. So we have lots of memory operation but much less number of compute operations here. And significant amount of very large number of memory loads and stores here. Similarly I can also do that for the load by column case.

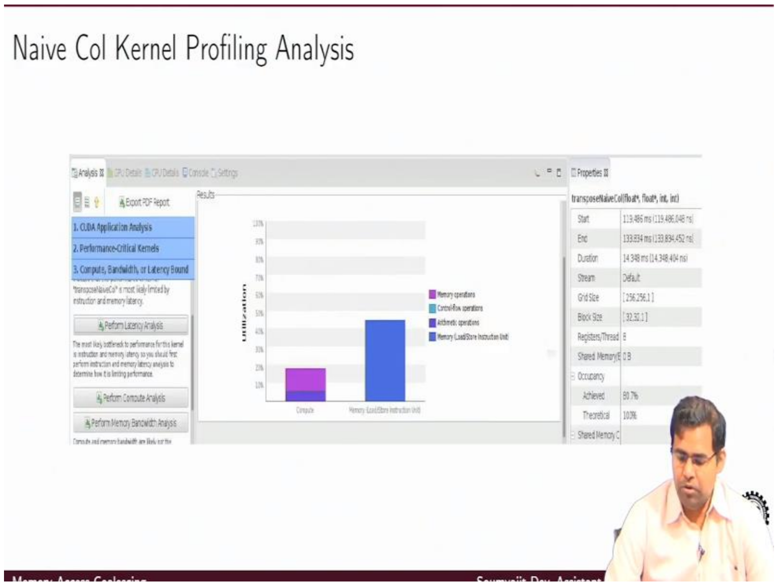
So this is a naive column implementation as you can see naive row I have some timing which is 8 milli second for naive column I have even more timing I even more timing requirement which is around 14 milliseconds. Now there is interesting question here right because of course in one case the naive row case the loads are coalesced, but stores are not coalesced. Whereas in the naive column case loads are not coalesced but stored are coalesced.

(Refer Slide Time 08:24)



So if I do the compute analysis, I also can get the charts here in terms of utilization. So this is the utilization chart here if I plot them side by side as we can see the loads stores apparently if I just provide them the number of the utilization for the load store is further small in the naive column case with respective to the naive row case.

(Refer Slide Time 08:54)



Naive Col Kernel Profiling Analysis

Analysis | GPU Details | CPU Details | Console | Settings

Export PDF Report

1. CUDA Application Analysis
2. Performance-Critical Kernels
3. Compute, Bandwidth, or Latency Bound

"transposeNaiveCol" is most likely limited by instruction and memory latency.

Perform Latency Analysis

The most likely bottleneck to performance for this kernel is instruction and memory latency so you should first perform instruction and memory latency analysis to determine how it is limiting performance.

Perform Compute Analysis

Perform Memory Bandwidth Analysis

Compute and memory bandwidth are likely not the

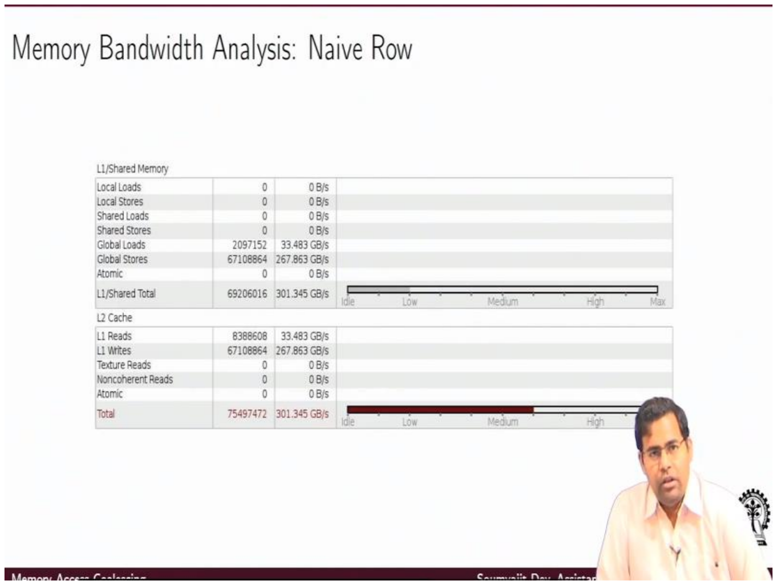
Properties

transposeNaiveCol(float*, float*, int, int)

Start	119.486 ms (119,486,048 ns)
End	133.834 ms (133,834,452 ns)
Duration	14.348 ms (14,348,404 ns)
Stream	Default
Grid Size	[256,256,1]
Block Size	[32,32,1]
Registers/Thread	8
Shared Memory/E	0 B
Occupancy	
Achieved	80.7%
Theoretical	100%
Shared Memory C	

And that actually matches with the execution time statistics because from our observation for the naive row case the execution time was smaller it was 8 milliseconds. Whereas for the naive column case the execution time is much larger it is 14 milliseconds. So it is much larger and that gets reflected here in terms of the utilization of loads stores in case of naive column the utilization is less.

(Refer Slide Time 09:19)



Memory Bandwidth Analysis: Naive Row

L1/Shared Memory

Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	0	0 B/s	
Shared Stores	0	0 B/s	
Global Loads	2097152	33.483 GB/s	
Global Stores	67108864	267.863 GB/s	
Atomic	0	0 B/s	
L1/Shared Total	69206016	301.345 GB/s	Idle Low Medium High Max

L2 Cache

L1 Reads	8388608	33.483 GB/s	
L1 Writes	67108864	267.863 GB/s	
Texture Reads	0	0 B/s	
Noncoherent Reads	0	0 B/s	
Atomic	0	0 B/s	
Total	75497472	301.345 GB/s	Idle Low Medium High

But again if we come to doing the comparison with respect to the memory band memory band width that has been confused and that has been consumed here for the naive row as well as the naive column implementation. Let us go one by one so for the knife row implementation this is what we have. So as we can see that we have synthesis by row so the number of global loads is less.

The number of global stores are much more and so this is basically the number of global loads that we are having and then this is the this also provides you with a statistics of the L2 cache that how many reads are performed by the L1 from L2 cache and similarly how many writes are performed by the L1 to the L2 cache right. Now something interesting as you can see that the number of L1 reads is much more with respect to the global loads right it is of course 4 times.

Now what is the reason for that? The fundamental reason is that in G in our GPUs the L1 is 128 byte wide. The cache line of L1 are 128 byte wide where so the essentially that is 32 times 4 bytes so that is equal to the global memory transaction width. So the L1 which is also is configurable as L1 or share memory. The width of the cache line is same as the global memory transactions.

But in between I have the L2 cache even unified L2 cache across the SM we for which the lines are 32 bit width 32 byte width right. So essential when the L1 has to read the number of reads is

4 per unit and in that way whatever is the number of global loads since the number of global loads have to go through L2 to L1. L1 is why having a width which is matching the load width right the load width which matches with the warps read width.

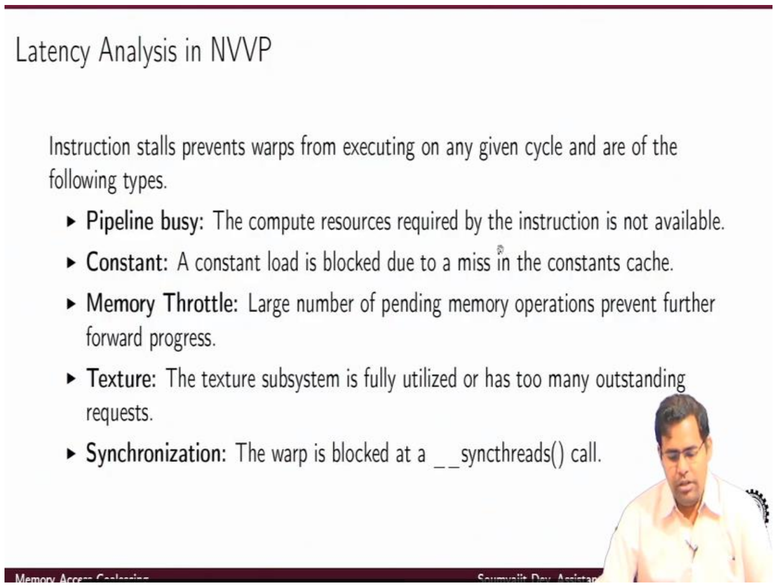
But in between you have a L2 cache which is 32 bytes it is having cache lines which are 32 byte width. So you have a factor of 4 increase in the number of L1 reads require from the L2 right. So overall the if you can look at here since this naive row so the number of global loads is small the stores are larger and similar things we can also say for the L1 reads and writes. The number of reads is small the number of writes are much more which is (()) (12:17) coherent with the number of stores here.

But then if we go to the naive column implementation we similarly see that this other way around the number of global loads is much more because they are not coalesced they are load from columns. The number of stores is much smaller and similarly so when the L1 is reading they are same as the number of global loads but when the L2 writes when the L1 writes to the L2 cache I mean of course it is much smaller in number than the number of reads.

But again we have this factor of 4 increased when I look at the number of times L1 writes to L2 and the number of stores right because again we have this multiplication factor of 4 here because of the width issue that I just discussed. So this would be advisable at this point that you keep these things in mind that whenever doing reads and write what is the width of the cache lines and what is the width of the global memory transactions and these things may or may not change with architecture families.

This is something we need to be aware of if you have to correctly interpret profiling data from any execution run.

(Refer Slide Time 13:39)


Latency Analysis in NVVP

Instruction stalls prevents warps from executing on any given cycle and are of the following types.

- **Pipeline busy:** The compute resources required by the instruction is not available.
- **Constant:** A constant load is blocked due to a miss in the constants cache.
- **Memory Throttle:** Large number of pending memory operations prevent further forward progress.
- **Texture:** The texture subsystem is fully utilized or has too many outstanding requests.
- **Synchronization:** The warp is blocked at a __syncthreads() call.

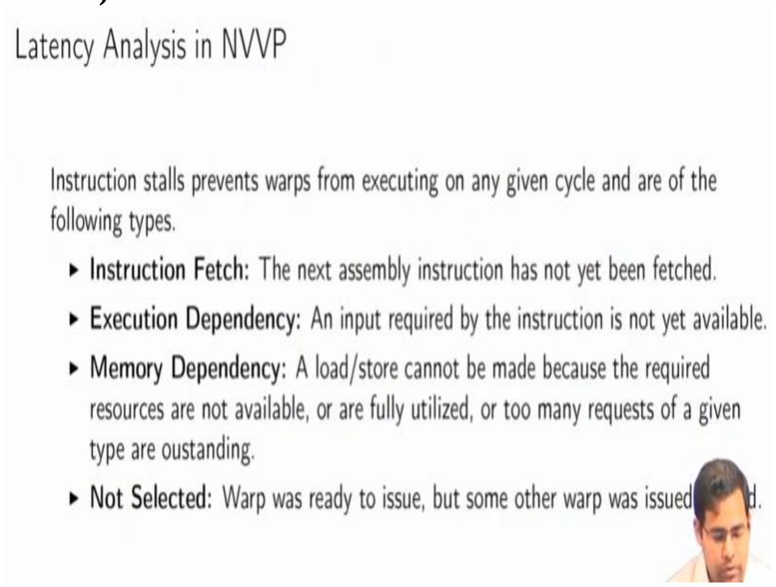
Now this still does not really explain to us that why the number of number of seconds or milliseconds and the total amount of time taken by the naive row implementation was much smaller with respect to the naive column implementation because in both case I have some penalty. In one case the penalties with respect to the loads and in the other case the penalties with respect to the stores.

So if you do a latency analysis in using the NVIDIA visual profiler you see that it provides the reasons why instruction stall and it prevents the warps from executing. So we always know that the we have lot of warps to execute but warps stalled due to certain reasons right and the reasons can be very for example the pipeline maybe busy. If the pipeline is busy then the compute resources that are required by the instruction for the warp is not available right. So the warp is stalled.

Then the issue can be with constant that some constant value is going to be required and that constant loads is blocked due to a miss in the constant cache. So as we know that the constants can be stored in the separate constant cache rather than using the L1 or shared value L1 or shared memory. The other issue can be of memory throttling like there can be large number of pending memory operations which prevent further forward progress.

Lots of many memory operations are pending for a warps so it has to stall. Then the texture subsystem if it is already fully utilized and so this is more for the graphics part if it is fully utilized and there are too many outstanding requests from the texture systems then also warps can get stalled. And of course the other thing is a warp can be blocked at a sync thread call because the other threads which are part of the block have not part and hence part of the other warps have not progressed up to the synchronization barrier.

(Refer Slide Time: 15:37)



Latency Analysis in NVVP

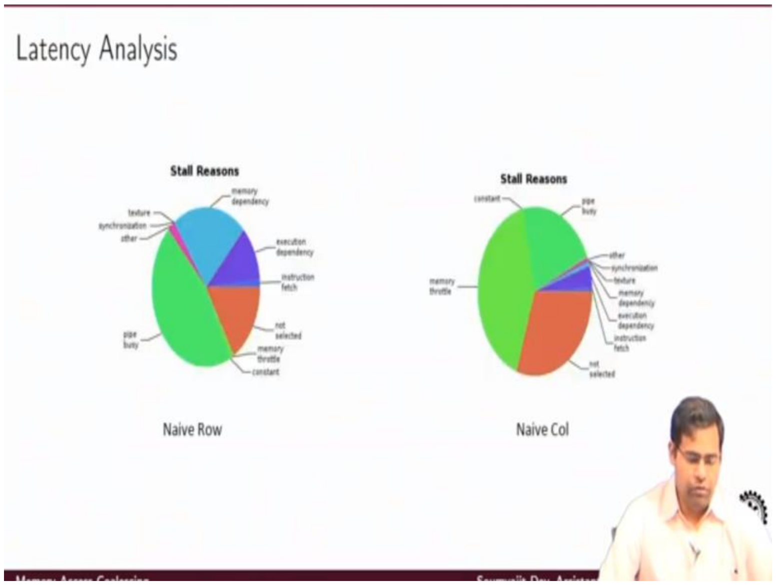
Instruction stalls prevents warps from executing on any given cycle and are of the following types.

- **Instruction Fetch:** The next assembly instruction has not yet been fetched.
- **Execution Dependency:** An input required by the instruction is not yet available.
- **Memory Dependency:** A load/store cannot be made because the required resources are not available, or are fully utilized, or too many requests of a given type are oustanding.
- **Not Selected:** Warp was ready to issue, but some other warp was issued d.

So the instruction if the if there is a stall in any of given cycle then it prevents the warps from executing and the different kind of instruction stalls that can happen can be classified into the following type like this instruction which in this case if there is a stall then if there is then the next assembly instruction will not be fetched right. Now of course there can be execution dependency like an input is required by the instruction and that is not available. Of course there can be memory dependency that I require the data points from the memory for I require to store data point into the memory.

It cannot be done because the resource says that a memory system is quite busy and then the warp is ready but the warp scheduler is not ready to issue it. The warp scheduler may be busy with some other threads which it has already issued. So there can be these several reasons right due to which warps may stall from execution.

(Refer Slide Time 16:56)



But if we do a analysis of these different types so as you can see we have all these listed types of different reasons for which the warp can stall from execution and if we do a latency analysis of both the kernels that we executed the naive row and naive column of kernel and see what was the primary reason for the warps stalling. Then we can see the naive column kernel has significant amount of stall due to the warps not getting selected and the memory throttling issue.

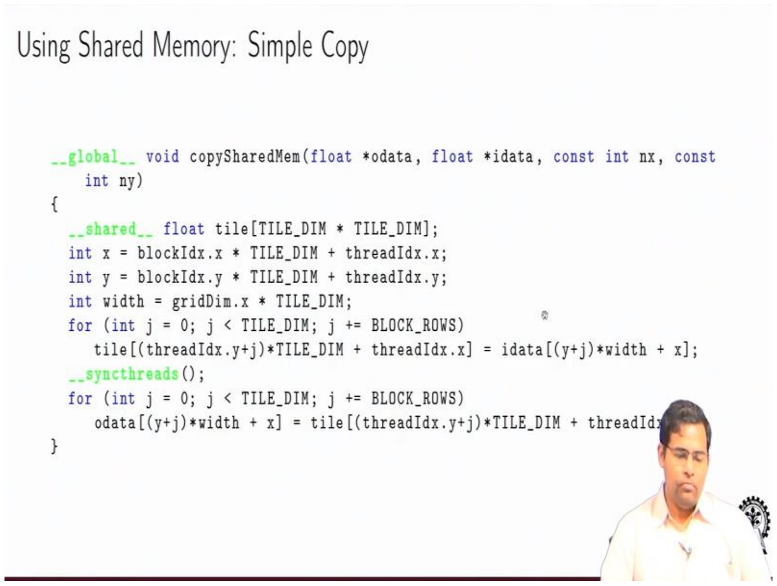
Whereas in the naive row whenever there was some stall it was primarily to do to the pipeline being busy executing some operation right. Now observe something that if the pipeline is busy in executing some operation it really is not a bad thing because still you have progress in terms of operation. But when you have the memory throttling issue if you want you can just have a look into the memory throttling issue.

So essentially you have large number of memory operations that are pending right or memory dependency issue that sperate that loads store cannot be made because the required resources are not available. For example I am reading from multiple banks or 2 warps going to read from the same bank but there is a I cannot do that right. So due to all these separate issues I can have stalls happening but suppose take this example in case of naive column memory throttling are a very big issue.

That means there are lot of pending memory operations and the warp not getting selected is also significant big issue. Now why would that happen. If you remember the naive row implementation you are able to load very fast. Since you are able to load very fast the warp would make progress. But they will get stalled in the right part. But here your warps cannot even make progress because you have got the issue that you are loads are getting stalled because the loads are not coalesced in case of naive column right.

Since your loads are not coalesced the warps really cannot start executing or at least copying the data. So that would actually create the more significant proportion of the scenario that memory throttling is happening and the warp is not selected. Now of course every kind of stalled has a corresponding penalty but things related to the memory will always have the larger penalty and since you are having lot of more load operations which are stalled that also contributes to the naive column implementation being faster slower in this case.

(Refer Slide Time 19:27)



Using Shared Memory: Simple Copy

```
__global__ void copySharedMem(float *odata, float *idata, const int nx, const
    int ny)
{
  __shared__ float tile[TILE_DIM * TILE_DIM];
  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;
  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     tile[(threadIdx.y+j)*TILE_DIM + threadIdx.x] = idata[(y+j)*width + x];
  __syncthreads();
  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     odata[(y+j)*width + x] = tile[(threadIdx.y+j)*TILE_DIM + threadIdx
}
```

Now of course there can be further optimization that can be made using the shared memory or shared memory based implementation of the code. Now this is something which we will like to take up in the next lecture so we will like to see that how the shared memory can actually help in doing that transpose operations because still now all we have done is looked into the transpose operation from the point of view of global memory loads and stores. So with this we will end this lecture here thank you.