GPU Architecture and Programming

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

Module No # 04
Lecture No # 20
Memory Access Coalescing (Contd.)

Hi welcome to this lecture on GPU architectures and program in the last lecture we have been providing you with an introduction on how GPU memory access is managed and what is the impact of the way in which GPU is accessed with respect to performance of your program and how this performance factors changed with different variance possible for access expressions inside your program.

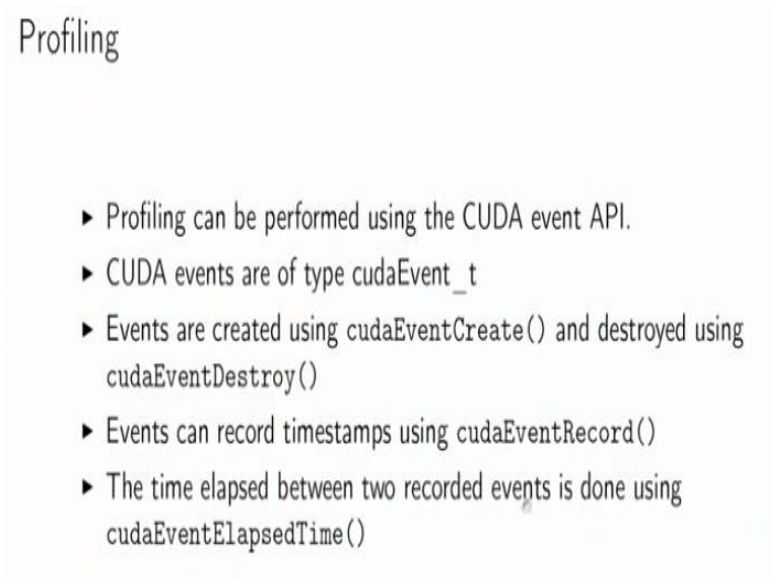
And but of course whenever we are talking about the performance you need a metric through which you can actually measure the performance of your program. So even if my metric is execution time how do I really measure the execution time of a GPU program. For measuring execution time of programs one standard way is to profile the execution of the program. So what is profiling of a program?

Essentially there are profiling software's through which you can monitor the execution that means it will simply monitor some hardware counters present with your architecture through which you can actually times stamp the execution of the program. And the typical way to do that is also a I mean I can use a different profiler for profiling the execution of my program or in modern programming languages itself they provide with lot of profiling primitives.

So I do not need a separate profiler but just I can gather the times the execution time instants of difference segments of program and which will essentially make use of the hardware performance counters present in the architecture. So this programming primitives will actually tell me when some part of the program actually started and when some part of the program actually ended and by just measuring the difference and all that I can simply point out what is the exact execution time for that segment of the code for that architecture.

Now of course we can see that this is an important scenario because finally we want efficient code to be deliver and efficiency of the code is a function of lot of things in terms of what is the memory architecture design of the system how many threads are executing what is the global memory transaction width and all the parameters were discussed earlier. So let us first figure out how a CUDA program can be profiled using programing primitives.

(Refer Slide Time: 02:53)



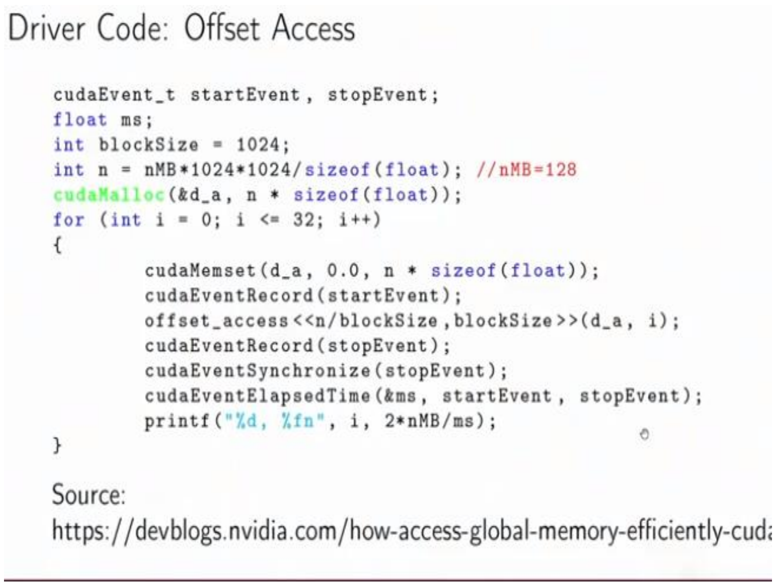
Profiling

- Profiling can be performed using the CUDA event API.
- CUDA events are of type cudaEvent_t
- Events are created using `cudaEventCreate()` and destroyed using `cudaEventDestroy()`
- Events can record timestamps using `cudaEventRecord()`
- The time elapsed between two recorded events is done using `cudaEventElapsedTime()`

So in CUDA the event API provides you a with the profiling primitives and there are several types of CUDA events they are available under this type CUDA event underscore t. Now events can be created using a function CUDA event create and created events can be destroyed from its (()) (03:22) I can just destroy the event using the other function CUDA event destroy of this event variables can be used for recording timestamps using other primitive known as CUDA event record.

Now of course the difference between this recorded times I need to gather for that again there is a API function which is called CUDA event elapsed time.

(Refer Slide Time: 03:47)



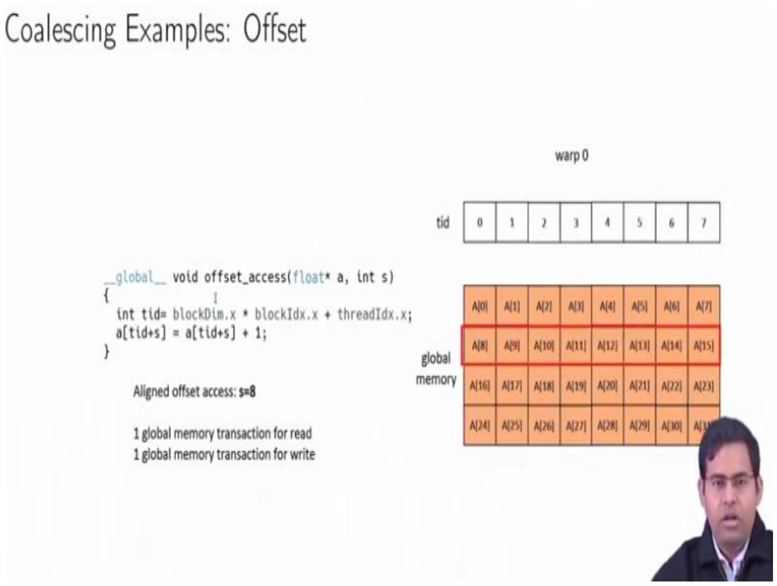
Driver Code: Offset Access

```
cudaEvent_t startEvent, stopEvent;
float ms;
int blockSize = 1024;
int n = nMB*1024*1024/sizeof(float); //nMB=128
cudaMalloc(&d_a, n * sizeof(float));
for (int i = 0; i <= 32; i++)
{
        cudaMemset(d_a, 0.0, n * sizeof(float));
        cudaEventRecord(startEvent);
        offset_access<<n/blockSize,blockSize>>(d_a, i);
        cudaEventRecord(stopEvent);
        cudaEventSynchronize(stopEvent);
        cudaEventElapsedTime(&ms, startEvent, stopEvent);
        printf("%d, %fn", i, 2*nMB/ms);
}
```

Source:
https://devblogs.nvidia.com/how-access-global-memory-efficiently-cud

So we will just see some example program where this things can be put into good use. So consider the offset access example program that we looked into earlier so just to remember what it really was.

(Refer Slide Time: 04:02)



Coalescing Examples: Offset

```
__global__ void offset_access(float* a, int s)
{
  int tid= blockDim.x * blockIdx.x + threadIdx.x;
  a[tid+s] = a[tid+s] + 1;
}
```

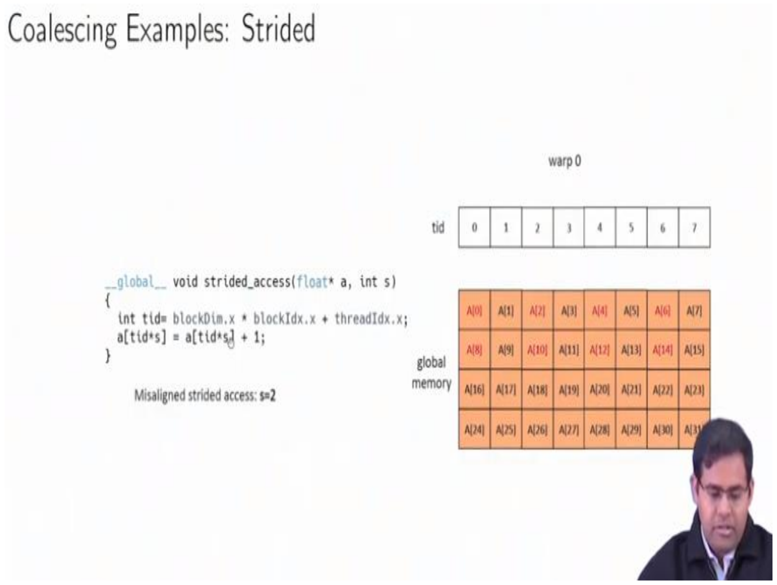
Aligned offset access: s=8

1 global memory transaction for read
1 global memory transaction for write

warp 0
tid
0
1
2
3
4
5
6
7
global memory
A[0] A[1] A[2] A[3] A[4] A[5] A[6] A[7]
A[8] A[9] A[10] A[11] A[12] A[13] A[14] A[15]
A[16] A[17] A[18] A[19] A[20] A[21] A[22] A[23]
A[24] A[25] A[26] A[27] A[28] A[29] A[30]

Let me just walk back to that earlier program on offset access this was our program we are trying to run this small piece of code with an offset parameter is. So all we want to do is I want to check what is the execution time of this kernel on some specific GPU and from our knowledge we have understood that if I keep on changing the value of this definitely the performance is going to change I mean depending on whether I am doing offset access there should be some pattern of change in the performance.

(Refer Slide Time: 04:43)



Coalescing Examples: Strided

```
__global__ void strided_access(float* a, int s)
{
  int tid= blockDim.x * blockIdx.x + threadIdx.x;
  a[tid*s] = a[tid*s] + 1;
}
```

Misaligned strided access: s=2

warp 0
tid
0
1
2
3
4
5
6
7
global memory
A[0] A[1] A[2] A[3] A[4] A[5] A[6] A[7]
A[8] A[9] A[10] A[11] A[12] A[13] A[14] A[15]
A[16] A[17] A[18] A[19] A[20] A[21] A[22] A[23]
A[24] A[25] A[26] A[27] A[28] A[29] A[30] A[31]

Whether i can doing strided access which is the other variant of the program so this is strided access there would be some other kind of impact on performance we want to see that how such things can really be measured. So as we have discussed this CUDA event API is providing me with this functions CUDA event create destroy and event record and finally I can measure the elapse time using the CUDA even lapse time function.

So let us now have a look into sample program here so the first thing we are doing is where defining 2 variables whose type is CUDA event t the variables are start event and stop event and I am going to use them to record timings at 2 different instances of course the start and end time. So here we are writing a small driver program for the host side so essentially for loop through which the thing that I do is in the loop is going to run for these many times in every iteration of the loop what am I really doing is I am putting in the CUDA event record.

So what is that if you look into our earlier instances that this CUDA even record function can be called to make one of the CUDA event types variables to start recording the timestamp right that is all we have done. So we have define this CUDA event type variables and using the CUDA event record primitive I am starting to the execution time from this time point. Immediately after this starting point for recording I am launching the kernel offset access with the parameter N.

I mean so essentially I am launching N number of threads and it is been divided into I mean N / block size number of blocks then each block is containing block size number of threads. So the offset code is being lunched and it is being passed with this parameter da and i now if you look into the earlier code of offset access. So this second parameter essentially offset value yeah so this is offset access code and this is offset value.

So in this function I am going to launch this kernel for each multiple number of times inside the loop in every iteration I am changing the offset values. So essentially I am trying to launch this function multiple times each time with a different offset immediately after the kernel execution finishes and the GPU is notifying the host that it has finished I have this CUDA event record with the stop event.

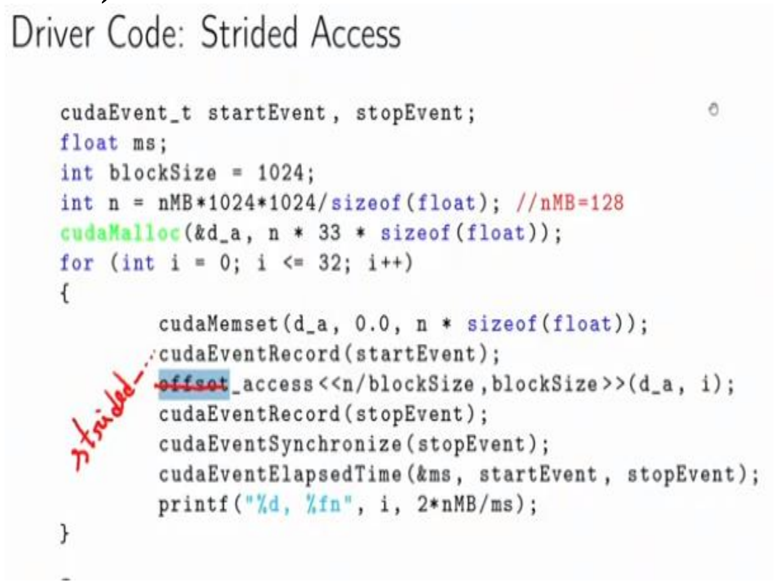
So this function CUDA event record will start recording the timing with the stop event that means essentially from this time I am measuring that when this counter is providing me the time point for when the kernel is going to finish right. And then I would need a CUDA event synchronize for the stop event because essentially this is all asyncrhonize calls and finally I need to synchronize this point.

So essentially using the start even monitor I have stared the recording of time at this point using the stop event monitor that is the CUDA event I have started recording the time from this point. So essentially from this two so if I provide this 2 different monitors to the other call CUDA event elapse time now of course I can consider their doing monitoring from starting from 2 different points they are doing the time event recording from this 2 points.

This function is provided with this 2 event monitors which have been recording all the events starting from this 2 time points. This function will actually go through them whatever is there in their statistics because they are doing a recording of events that have been going on from different start times so this function will take care of those differences in start times and record that difference in the other first in the first augment variable right.

So I hope the point is clear I can use this data structure to monitor several kind of a architecture phenomena's. But in this case I am only interested in the difference in the start time of this recorders that specific information can be provided to me by using this function which is doing the work of actually looking into the recoding events and they have corresponding start times for these two monitors and providing me just with whatever I am interested in which is the total elapse time between the kernel launch and the kernels execution finish.

(Refer Slide Time: 09:42)



Driver Code: Strided Access

```
cudaEvent_t startEvent, stopEvent;
float ms;
int blockSize = 1024;
int n = nMB*1024*1024/sizeof(float); //nMB=128
cudaMalloc(&d_a, n * 33 * sizeof(float));
for (int i = 0; i <= 32; i++)
{
        cudaMemset(d_a, 0.0, n * sizeof(float));
        cudaEventRecord(startEvent);
        ~~offset~~_access<<n/blockSize,blockSize>>(d_a, i);
        cudaEventRecord(stopEvent);
        cudaEventSynchronize(stopEvent);
        cudaEventElapsedTime(&ms, startEvent, stopEvent);
        printf("%d, %fn", i, 2*nMB/ms);
}
```

strided

Now the similar thing I can also do for a strided access so when I do I will just change code from offset access to strided access. So one thing you have to take care that when i am talking about strided access this function call should also change to strided I mean so that is some error in our part. As you can see for strided access this is our kernel with is being the stride which is there.

(Refer Slide Time: 10:52)

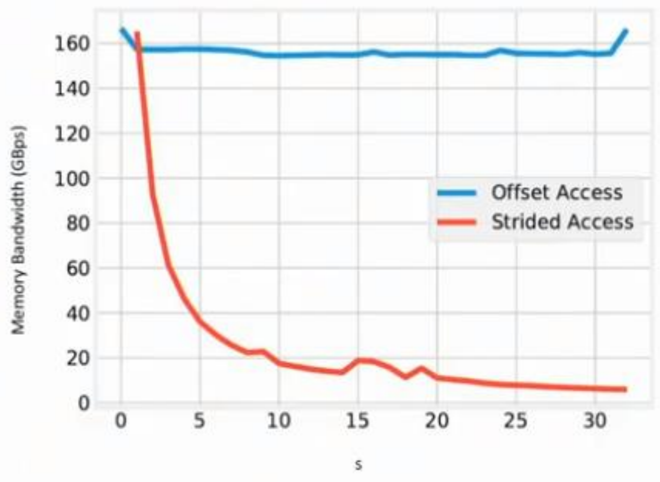


Figure: Memory Bandwidth Plot

So I can do a similar thing for strided access so I just need to change this name so then as you can see that using this 2 programs we are able to figure out how the total execution time of the kernel changes with the change of the offset parameter and with the change of the stride parameter. If we plot this thing so here we have a nice plot so essentially we are changing the S parameter and we are trying to figure out that okay what is the total memory bandwidth consumed by the system essentially the elapsed time of the kernel.

When I am changing the stride of the kernel or I am changing the offset we have got this 2 curves. Just take a minute to look minutely into these curves and figure out why they behave like the way they have been plotted. So let us first figure out what happens in the offset curve for that let us first go back to the code offset access. So this is how it is right so considering an offset of 1 you are getting first when you do not have any offset for every execution of the kernel we have 1 global memory read followed by 1 right.

Consider an offset value of one so now you have 2 global memory reads and 2 global memory write what happens with the offset value of 2. So with offset value of 2 you will have the data points A2 upto the data points in A9 and then you will have a data points from A10 upto the data points upto A17 then data points from A18 upto the data points A25 right. This is how things will go on so essentially whether my offset value is 1 or 2 in both cases the global memory reads will require 2 transactions followed my global memory writes requiring 2 transactions with offset value 3.

Again I would have the same thing so with offset value 3 I am accessing let us say from A11 upto A18 but again what is the number of transactions 2 global memory transactions for read and 2 global memory transactions for write. So in this way as you can see that with the increase of S from 0 to 1 I have a band width reduction but after that things remains same with offset 2, 3, 4 things remains same.

But what happens when the offset value is same as the warp with when s is 8 then what will happen is for warp 0 you instead of accessing from A0 to A7 you would be accessing A8 to A15 right. So again everything is nicely coalesced so you are back to global memory transaction 1 for read and 1 global memory transaction for write. So essentially with respect to offset what is the situation when you have no offset for this program you have the best performance whenever you have some offset your performance decreases but it will decreases to the point which will remain constant for further increase in offset.

Until the point where the offset is aligned with the warp with so that again you have the nice coalesced global memory transactions and that will again increase the bandwidth. So now with this idea if we go back and see that the total performance reduces first but that just for the first value from S0 to S1 and then this is all specific to that program it remains the same. Then again when it will get access it will ahh get byte line so here we are doing all our runs on (()) (15:23) GPU.

And of course as we know the standard warp is 32 size so immediately at 32 things will be getting byte aligned nicely. The global memory is again nicely coalesced as the offset 0 cased so I have a reduction but then the reduction does not change I am just doubled up the usage I mean for all the threads for that specific program line then again the performance is back to what it was 40 offset right.

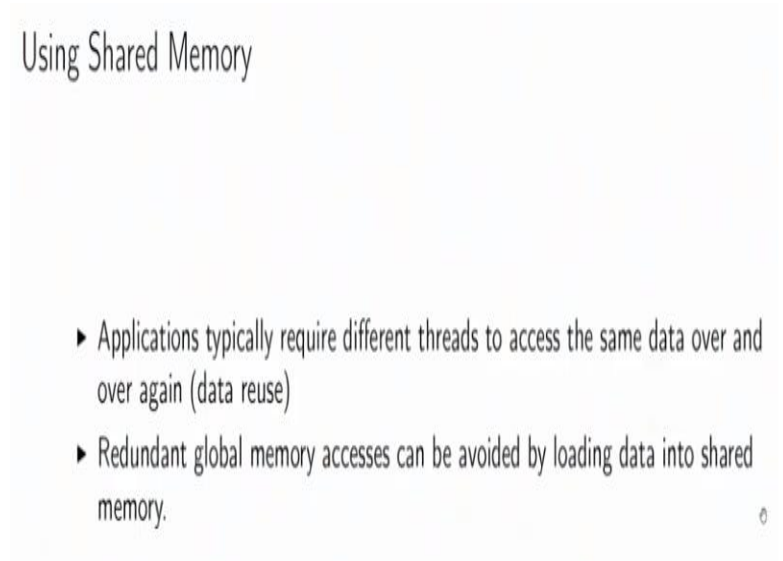
But that is not such case for strided right let us go back to strided code once again so as you can see when you are running the strided code when your stride is 0 so there is no stride so you have again for every instance of this operation 1 global read and 1 global memory write what happens with stride. So this is the stride which is changing from 1 to 2 now the number of global memory reads and writes have doubled up it is stride 4 the number of global memory reads and writes have further doubled up with respect to s = 4.

So if I keep on increasing the stride factor the global memory transaction the number of transaction is doubling up right. So this would actually lead to drastic reduction in the memory bandwidth provided by the (()) (17:04) to this kernel. So which will lead to huge performance loss and if I look at the effective memory bandwidth that I am getting it is reducing drastically so the more I increase the stride the memory accesses are getting even more scattered.

So there would be at some point I will loss all kinds of coalescing and in a worst case I will have for all thread id’s inside the warp I am doing separate global memory transactions. And that would be the worst case so I hope that is understood because nothing can be worse than that I am executing a warp. The warp as 32 threads each of the thread are requesting locations in the memory which are far apart and they are so much apart that in they cannot be brought in a single transaction.

So that is the worst case right I mean so after that if I go on increasing the stride it really does not matter because essentially then by that time when for each access I am doing a separate volume transaction that would mean for every warp every instance of execution of the warp for some load or store instruction I am going to do 32 separate global memory transaction. So that is the worst case performance and that would give me the saturation point in this curve the lower saturation point in this curve right.

(Refer Slide Time: 18:28)



Using Shared Memory

- Applications typically require different threads to access the same data over and over again (data reuse)
- Redundant global memory accesses can be avoided by loading data into shared memory.

Now of course I can use I mean better arrangements of data points for whatever computation I am doing to reduce the number of global memory access I need to do in such pathological cases. So overall the recommendation is that whenever you are writing your program you need to make good use of your memory segments you have to understand the GPU’s memory hierarchy and accordingly you should write the code so that the access patterns are nice with respect to coalescing.

I mean this is where the role of shared memory it coming because as we all that we have discussed earlier many times that inside each SM you have a portion of the memory which can be configured as shared memory or L1 cache if you are team your algorithm and write the code in such a way that you are doing less number of global memory accesses and by doing some smart usage of shared memory that can increase the performance of your code drastically.

So although the I mean in general the property of applications is that they typically required different threads to access the same data over and over again. So that is our we know that our principle of locality or data use I have a principle of locality which is special as well as temporal. So different threads may need access to the same data or threads may need access to data points which are located in neighboring regions of the memory.

So the primary things is since different threads would need access to the same data if I keep the data in the memory at some point some other thread will require the data and also if I bring in chunks which is actually done by I mean the wide transactions that we always do the basic philosophy is that whenever we are bringing in the data to the memory and you are bringing data in chucks I mean it may so happen that in future with a high probability you would be actually reading the nearby memory elements which are already available to you in the cache.

So the shared memory can be configured as partly as cache and shared memory and the good thing about the shared memory is all the threads inside the blocks have a consistent view to the memory. So in many cases based on this idea that different threads can actually access to the same data over and over again at different time points maybe I can reduce the global memory accesses by avoiding the multiple load and store from the global memory.

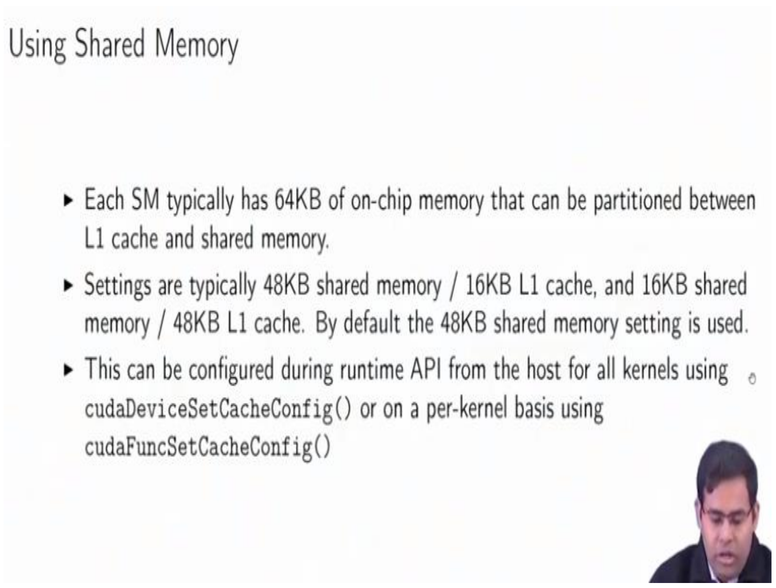
But rather I put the data in the shared memory and keep it there for collaborative access by other threads at different points of time. So I make one thread responsible for bringing the data and putting into the shared memory using a shared data type and then that data points can be used by some other data point or by some other thread by other threads at different points of their computation.

Now if this can be done then as we can understand that see every thread whenever it is doing a memory reference if it can get in from the shared memory then there is a order of magnitude reduction in the access time with respect to the global memory access and that is quite useful as an optimization this is one of the primary motivations are keeping a shared memory segment in the GPU.

It allows threads inside the block to collaborate among each other the data elements brought in by threads in the block can be used by other threads in the block at different points of execution by accessing them from the shared memory if they are actually made to be resident in the shared memory by the programmer. Now this is something important the hardware does not even decide what should be or should not be in a shared memory the hardware decides this for the cache by using standard cache principles.

I mean using the caches read and write policies right but for the shared memory access it is programmer who has to be decide by giving suitable data types that which data point should be resident in a shared memory and which data points should not be resident in the shared memory.

(Refer Slide Time: 22:51)



Using Shared Memory

- Each SM typically has 64KB of on-chip memory that can be partitioned between L1 cache and shared memory.
- Settings are typically 48KB shared memory / 16KB L1 cache, and 16KB shared memory / 48KB L1 cache. By default the 48KB shared memory setting is used.
- This can be configured during runtime API from the host for all kernels using `cudaDeviceSetCacheConfig()` or on a per-kernel basis using `cudaFuncSetCacheConfig()`

Now further deeper into the shared memory so how is the shared memory structure this is something we have discussed again is the small recap. So each SM typically have 64 kilobytes of on chip memory which can be partitioned between the L1 cache and the shared memory now the setting is typically like this that you have 48 kilobyte out of that total 64 configured as shared memory and rest 16 kilobyte configured is L1 cache.

But it can also be the reverse by doing suitable programing setting you can make it like 16 KB shared and 48KB L1 cache. As I told earlier that if it is L1 cache then it is managed by the hardware just like the cache memory is managed but if it is a the shared memory if it is defined as the shared memory then the programmer has to manage it through suitable type definitions for the variables in the code.

Now this can be configured during the run time API I mean which part of I mean that is given memory block which is on chip in the SM whether it should be a shared memory or how much of its should be shared memory and how much of it is should be L1 cache essentially it is going to be 2 options that can be actually can be configured using the CUDA API. Now suppose you want to configure it permanently from the host for all the kernels so the host can technically launch multiple kernels one after another asynchronously.

But if I want a specific setting of shared memory and L1 cache available for all the kernels I can do that using this function CUDA device set cache I config. So this is the function which will take of doing the configuration of the shared memory I mean you can just look into the manuals online for NVIDIA and figure out what are the parameters for this or if I can do it in a part kernel basis that means I have a host code for 1 kernel I can configure the shared memory and the L1 cache in the one way and then for the next kernel I can configure in a different way.

Of course it depends on what is the property of the kernel how many threads are getting launched how the threads are being arranged whether it is actually useful to have more shared memory or whether it is useful to have more amount of L1 cache it depends on the programmers perspective he has to figure out whether having more shared memory actually helps that specific kernel then you would actually like to use this function to configure that memory segment with more number of shared more amount of shared memory.

Otherwise depending on if you want fast access to decrease the latency and you do not have really control over what I mean what really would be used along the variables by which thread you can actually have more amount of cache. Typically the principle I mean just to before concluding here I will also like to put in one important point which I have seen earlier that GPU is typically have less amount of L1 cache with respect to CPU's. The primary reason is that since I am going first of all I have the shared memory that is one segment that I have so I can afford to have some L1 cache.

But the other reason is that in this case the smaller L1 cache I am trying to hide the latency of L1 cache is by using more threads. So when I compare computing and GPU computing in CPU computing although I have multi-core multi threaded CPU course in a CPU the number of threads executing parallely in the CPU are much less with respect to GPU. So objective is there is to execute the threads fast enough one important impediment in terms of executing a thread faster enough this memory operation.

So I have good motivation for using large caches so that there is a high probability I can find whatever data is required in the L1 cache and that would reduce the memory access penalty and that will give you better performance. When I am using GPU essentially what I am doing is that using more I am actually using lot of threads in parallel right I am launching more number of threads in parallel.

So I have really do not care if some of the threads are suffering due to non-availability of data in the L1 cache because technically I have too many other threads to execute I have many warp waiting I have launched more number of warps then I have actually available physical space. So I can just stole that work and execute some other work whenever this work is stalked due to some high latency operation.

So this is how things are managed they are at different philosophies since in this case I have more number of threads to manage if some threads are waiting due to non-availability of data that is fine with me. So in this case I can affords to have smaller amount of L1 cache. So with this we like to end this lecture and may in the next lecture we will have to see a good program example through which we show how the usage of shared memory can be a nice optimization with respect to performance thank you.