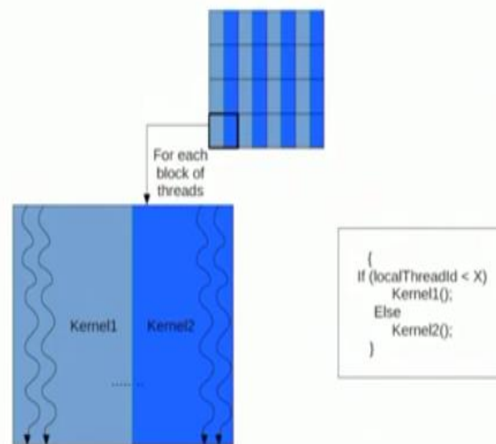
GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Lecture 37
Kernel Fusion, Thread and Block Coarsening (Contd.)

(Refer Slide Time: 00:30)

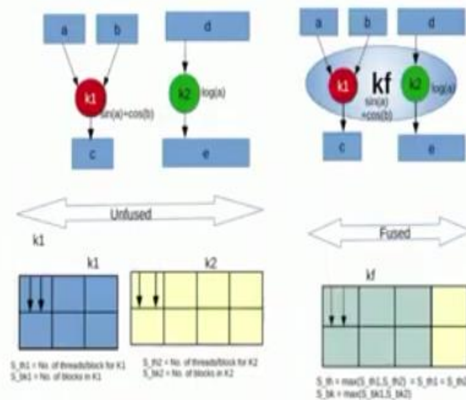


Hi, welcome back to the lectures on GPU Architectures and Programming. If you just recall, in the last lecture, we started with optimization that is how to fuse multiple kernels together and we covered the idea of inner thread fusion, one possible way to do the fusion of two kernels, and we figured out what is the relative advantage and disadvantage of doing inner thread fusion, when it is applicable and all that.

With that background, we will just move to the next possible way of doing fusion, which is inner block fusion. So the idea of inner thread fusion was very simple. We were just increasing the part thread activity, right, by making the thread do work for two or maybe some multiple data points. So if you just look into the example that we had.

(Refer Slide Time: 01:19)

Inner Thread Fusion Example



So this was our sample program for inner thread fusion. We applied it on independent kernels in two cases, one was same number of blocks and same number of threads per block and then we said that okay, let us allow it to have different number of blocks for the two kernels while we keep the number of threads per block same and then we generalize the concept.

So essentially we define these two operations that for the fused kernel, the number of threads would be the max of the original number of threads per block and original kernel in the number of blocks would again be the max of the total number of blocks in the two kernels, right and we made each of the threads when they are launched for the fused kernel to perform the activities of both the original kernels.

(Refer Slide Time: 02:11)

Inner Thread Fusion

Fusion of independent kernels with different data size but same thread/block size:

```
//Fused kernel
//Let n2>n1
kf(a, b, c, d, e, n1, n2):
i = global threadId
if(i<n1)
        c[i]=sin(a[i])+cos(b[i])
        e[i]=log(d[i])
else if(i<n2)
        e[i]=log(d[i])
```

So this was the sample code. So ideally these two lines were the activities of kernel 1 and this line was the activity of kernel 2. Now in the fused kernel, they are both together or part of a single thread activity, based on whether they satisfy the requirements of whether they are actually varied operations for both of the kernels or not. Now coming here, we take different stands.

What we do is, okay, we do not fuse the kernels at that granularity of threads, but we start looking into the fusion at the granularity of a block. So what we do? If you are given two threads, you increase the block size of a fused kernel and you say that okay my number of threads in the block should be such that some of them will be doing the activity for kernel 1and some of the threads in the block, we will be doing the activity for kernel 2.

So considering that these shaded areas represent activities for the individual kernels, when I am fusing them, this is my space containing the different launch threads and this granularity is showing the blocks now in the new fused kernel, where essentially this represents the original block size for k1 or kernel 1 and this is for k2, kernel 2 and they together form the new block for the fused kernel.

So we will just compute a global thread ID here, sorry a local thread ID here, that is I do not need to know what is the global thread ID, that is the ordering of the thread in a linearized

manner, but we are just interested in the offset or local thread ID with respect to this block, right. So I just look at what is the local thread ID. So just to recall how do I compute the local thread ID, so you will have the parameters dot x, dot y, dot z.

And you will like to use them with suitable multiplication factors in the x dimension, y dimension and z dimension and compute local thread ID here. So my point is the local thread ID does not represent the exact location of the thread in a linearized manner in the entire grid, but it just represents the linearization of the threads with respect to their ordering inside one block, right.

So with that, once we compute the local thread ID and then, we figure out that, okay what is the position of this thread inside this block. We decide that okay, whether to put the local threads with the threads with smaller local IDs to perform the activity for kernel 1 or kernel 2 and then we write the code and we will put in the activities like this. Suppose for kernel 1 the number of threads per block is x, right.

So for the fused kernel, we will just figure out whether the local thread ID is strictly less than this x or not and accordingly we will make the thread to do the job for kernel 1. If the local thread ID is equal to x or greater than x, then we will make it do the job for kernel 2. So these all things will work. We will have the work here for kernel 1 and then we will have the work here for kernel 2.

Just to note here and the difference earlier, the same thread was doing the job of both kernels. So we are doing a thread level fusion, inner thread fusion, but here we are doing a block level fusion. So we are adding more threads into the fused blocks, computing the local thread in the block that is the relative position of the thread inside the block. Using this relative position or local thread ID, we are computing whether this thread is supposed to do the activity of kernel 1 or kernel 2 and accordingly we are making it execute the code for kernel 1 or the code for kernel 2.

(Refer Slide Time: 06:33)

Inner Block Fusion

- Distribute computation of two different kernels among threads in single block
- For independent kernels with small block size(threads/block)
- Let, $S_{th,i}$ represents the number of threads in a thread block; $S_{bk,i}$ represents the number of blocks in kernel i(i = 1, 2)
- For fused kernel -
  - $S_{th} = sum(S_{th,1}, S_{th,2})$
  - $S_{bk} = max(S_{bk,1}, S_{bk,2})$
- Not suitable if -
  - S_{th} of fused kernel exceed upper bound of threads/block
  - kernels with synchronization statement

So if we just summarize it, essentially we will distribute the computation of two different kernels among the threads in a single block. For independent kernels with small block size or threads per block, this is the good idea. Why, because in case the sum of the threads per block for the kernels add up and cross the upper bound, then I cannot really do any inner block fusion, right. So this is a good idea for independent kernels with small block size, right.

So like earlier, let Sth,i represent the number of threads in a thread block and Sbk represents the number of blocks in the kernel, where i is either 1 or 2 representing kernel 1 or kernel 2. Of course, I can generalize I can have fusion of more than two kernels. So this is just a candidate example where I am showing how I will fuse two kernels. So for the fused kernel, we will have the total number of threads per block as the sum of these two.

Because I will delegate some of the threads to do the activity for kernel 1 and I will delegate some of the threads to do the activity for kernel 2, like that and what should be the block size. First of all, see I am widening the blocks. I am not interested in increasing the number of blocks. So the block size should not be any sum or things like that, as is done with the number of threads, but the block size is rather the maximum of the block size, sorry.

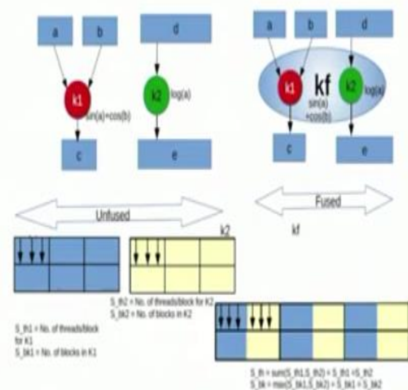
The number of blocks is rather the maximum of the number blocks for each of the two kernels. So this seems I have to cover both the kernels. So my number of blocks has to be the max of

Sbk1 and Sbk2. So when does this not work in general? Of course, if I am trying to fuse two kernels, where each of the kernels have lot of threads per block and the sum of Sth1 and Sth2 is greater than 1024, the upper bound and maybe for the higher level of architecture that would be 2048.

So whatever is the upper bound for the GPU, if it cross the sum crosses the upper bound, then this is not working. So apart from the issue with this upper bound, the other issue is that kernels with synchronization statement, for them also the idea of inner block fusion would not be suitable, simply for the reason that a synchronization statement will force synchronization for all the threads in a block and it cannot work for a subset of threads, which will be part of a block.

(Refer Slide Time: 09:18)

Inner Block Fusion Example



So let us come to the example of doing inner block fusion of independent kernels. So we go back to our original examples of k1 and k2, fine. So when we are fusing these two, as you can see that for k1, the operation was sin a + cos b and we are storing the value in the buffer c. For k2, the operation is log 2 and it is stored in the buffer e and when I am fusing them, all we will do is, we will first define the number of threads and the number of threads per block, and the number of blocks.

So we will follow the method we discussed earlier. So we will increase the block sizes now by making it the sum of original block sizes and the number of blocks will simply be the maximum

of the original two. So the original two kernels are 6 blocks, so our fused kernel will also have 6 blocks and inside these blocks, I just have the double number of threads. Half of the threads are doing the job for kernel 1 and half of the threads are doing the job for kernel 2, as simple as that.

And which threads are doing the job for which kernel is demarcated here using the different colors. So the blue marked threads are doing the job for kernel 1, the yellow marked threads are doing the job for kernel 2, which also is marked in yellow, like that.

(Refer Slide Time: 10:46)

Inner Block Fusion Example

Fusion of independent kernels with same dataspace size

```
//Unfused kernels
k1(a, b, c, n):
        i = global threadId
        c[i]=sin(a[i])+cos(b[i])

k2(d, e, n) :
        i = global threadId
        e[i]=log(d[i])

//Fused kernel
//S_th1 = No. of threads/block for K1
//S_th2 = No. of threads/block for K2
//S_th = sum(S_th1,S_th2)
```

So looking into the program example, so these are the unfused kernels, k1, this is the k2.

(Refer Slide Time: 10:56)

Inner Block Fusion Example

```
kf(a, b, c, d, e, n):
        b = blockId
        t = threadId

        if(t<S_th1)
                c[t]=sin(a[t])+cos(b[t])
        else if(t<S_th)
                e[t-S_th1]=log(d[t-S_th1])
```

So when I am going to fuse these kernels using inner block fusion, so this is the pseudocode of my fused kernel. So first thing I will do is, I will compute the block ID and then I will compute the local thread ID. I will check whether this local thread ID. Again, let us remember, this is the local thread ID, so the offset inside the block. This is not the global thread ID. So we will check whether this local thread ID is less than the value at Sth1.

That would mean that this thread is supposed to do the job for kernel 1. Otherwise, we will check whether it is inside the boundary of the total number of threads per block for the fused kernel, because if this is not satisfied, then either the value is out of bound or the value is for the second kernel's work activity and accordingly the illustrative statement would fire.

Now let us just take another example. So here we have a bit of unbalance in terms of the number of blocks. So kernel 1 has got four blocks, kernel 2 has got five blocks, sorry six blocks. Kernel 1 has two threads per block and kernel 2 has got three threads per block that would mean in the fused kernel, we are going to have five threads per block and the number of blocks would be the max, which is 6 here.

So again we are showing that which thread is going to do the job of which kernel. We are again showing that using the highlighting scheme we discussed earlier. Observe that in the last block, what is happening, essentially in this picture, remember one thing, although I am not showing the arrows in the other blocks, I hope you understand that the threads are present here. So essentially, this pattern is replicating here, here, here, like that.

So just to avoid any confusion, let me just say just like this, this entire pattern is also present here, is present here, is present here, but observe that I am not going to have activity for kernel 1 here. So in this portion, there would be no activity for kernel 1. So what are you just have is only the k2 threads. Here I have threads for everybody, similarly here, but here I do not have any activity.

I hope you understand that as the data arrangement of the kernels become a bit different while fusing them, we are introducing an absence of balance, because when these blocks are going to

execute, there will be unused blocks, when waps will be formed. So essentially, we will be losing on the occupancy of the GPU and we will be not extracting that much parallelism. So whenever we are working with blocks or threads, which are widely varying in terms of their arrangement of data, the fusion operation may not be a good choice.

(Refer Slide Time: 14:22)

Inner Block Fusion Example

Fusion of independent kernels with different dataspace size

```
//Unfused kernels
k1(a, b, c, n1):
        i = global threadId
        c[i]=sin(a[i])+cos(b[i])

k2(d, e, n2) :
        i = global threadId
        e[i]=log(d[i])

//Fused kernel
//S_th = sum(S_th1,S_th2)
//S_bk = max(S_bk1,S_bk2)
//Let S_th2 > S_th1
//Let S_bk2 > S_bk1
```

So how do I handle the fusion of independent kernels, if the data space size is different, like this. Again, we have different number of threads per block here and also we have different number of blocks here. For that, let us understand that the code is going to be a bit different. Now these are the unfused kernels.

(Refer Slide Time: 14:50)

Inner Block Fusion Example

```
kf(a, b, c, d, e, n1, n2, X):
        b = blockId
        t = threadId
        if(b<S_bk)
                if(t<S_th1 AND b<S_bk1)
                        c[t]=sin(a[t])+cos(b[t])
                else if(t<S_th)
                        e[t-S_th1]=log(d[t-S_th1])
```

So when we are going to fuse these kernels, the first part is again same. So I have the block ID and I am going to have the local thread ID computed using the thread IDx and block IDx parameters, that is of course fine. The next thing is, first the thread has to figure out that what is its block ID. Now why is this important? Because if the block ID is beyond the max, then of course, there is nothing to be done.

So just to remember, Sbk here is the number of blocks for the fused kernel. Sth1 and Sth2 are the threads per block setting of kernel 1 and kernel 2. Sbk1 and Sbk2 are the number of blocks for kernel 1 and kernel 2. So we will figure out whether I am really having to do some valid computation. So whether the block ID is less than Sbk. If so, then you get inside and figure out whether you have some job. This thread is going to do some job for the kernel 1 or kernel 2.

So the check would be that the thread local ID, whether that local ID of the thread is less than Sth1 and whether the block ID of the thread is less than Sbk1. Now just to understand that why do I need this, because earlier if you see when we are doing, I am just hoping that to the code and inner block fusion for kernels with identical data arrangement. So we just were computing the local thread ID and we are checking whether the thread ID is going to do the work for k1 or k2.

But now it is different, because we have a varying amount of number of blocks. So I should not only check for a given thread that whether it is part of its local thread ID is inside the data space for kernel 1 or it is inside the data space, is beyond the data space for kernel 1, but inside the data space for kernel 2. Only doing that check is not going to be valid. I additionally need to check whether the block of the thread is a valid block ID for k1 or a valid block ID for k2.

For example, if the block ID is this one, 012, so then in the x dimension of course, then I can understand that even if the thread ID is inside the thread boundary for kernel 1, kernel 1 does not have any computation for this block ID. So we will have a check with Sth1 as well as Sbk1 and then only do the computation for kernel 1. Otherwise, we know that okay there is no work for the kernel 1 and we will just check whether there is inside the boundary for the total number of threads per block. If so, there should be some activity for kernel 2.

(Refer Slide Time: 17:39)

Inner Block Fusion Limitation

Upper bound for threads/block is architecture dependent

- S_{th} of fused kernel must not exceed this upper bound

CUDA does not support synchronization for partial threads in a block

- kernels with sync() statement like reduction kernel not applicable for this type of fushion

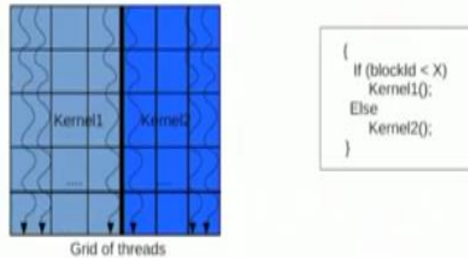
Now, coming to some of the important points like, as we have discussed earlier that total number of threads per block for fused kernel should not exceed the upper bound. We have to remember this, because the threads per block upper boundary architecture dependent and so if you are trying to write a parameterized code, like whether you should really have, I mean, suppose you are trying to write a code, where you are trying to generate a fusion of the kernels based on the GPU.

So you may first query the CUDA device property, figure out the allowed threads per block and then you can do the sum of the setting of threads per block for k1 and k2 and then decide in the runtime whether to execute a fused version of the kernel or not. So that is something which can be done, like, I mean you can actually decide on what kind of threading should you use, whether to fuse or not and also, this was the other important point we were discussing.

Like, CUDA does not support synchronization for partial threads in a block. So if we have synch or synch thread statements in the reduction kernels, one of the components, then this kind of fusion is not a good idea.

(Refer Slide Time: 18:59)

Inter block Fusion



So moving aside into the other situation, which is inter block fusion, so we will need to first understand what is this idea of inter block fusion. So the first thing we did was, I am just recalling back. We increased part thread activity, so that was enough thread fusion. Then, we increased the number of threads per block and decided whether the thread will do the job of kernel 1 or do the job of kernel 2. That was inner block fusion. So I am fusing blocks.

Now we are saying that okay, let us not do that. Let us not fuse threads or let us not fuse blocks, but let us just fuse the entire arrangements of blocks together and launch a kernel. So that just means that you do not disturb the internal structure of the data space of kernel 1 or kernel 2. So you just define a kernel with number of blocks being the sum of the original number of blocks of k1 and the number of blocks for k2 and then, you let some of the blocks do the job for kernel 1 and some of the blocks to do the job for kernel 2.

So that is a much more coarse grain fusion. All you are doing is, you are just fusing at a coarser grain. So essentially we are executing kernel 1 and kernel 2, but concurrently. Instead of not dispatching kernel 1 followed by kernel 2, you are dispatching both of them together as a fused kernel, but nothing changes in terms of their internal coding structure, you just delegate some of the blocks to do the job for kernel 1 and then you delegate the rest of the blocks to do the job for kernel 2.

So this is how it goes on. So you have the code for kernel 1, you have the code for kernel 2. You just take the block ID, you check whether it is a valid block ID for kernel 1, it is inside the number of blocks boundary for kernel 1, otherwise you execute kernel 2. So for kernel 1, you have progress of threads like this. For kernel 2, you have another alternate progress of threads like this and you have a boundary here and this is the entire grid of threads.

So this denotes, maybe a theoretical way for you to think, that this is x. So if you are on this side, you are executing kernel 1, for example this side, you are executing kernel 2. Of course, this is a 2D picture. Overall, this would be the summary of inter block fusion. You distribute the computation of two different kernels among different blocks, for independent kernels with similar computation time, this is a good idea. Now this is important.

Why do you say, first of all, we understand the kernels have to be independent, otherwise as we have discussed even in the case of inner thread fusion, if we are trying to fuse dependent kernels, you have to bring in synch thread statements. So that, you exactly know when the threads for one kernel finish, other kernel finish, and then you do some computation, so that is quite tricky thing to do and it may not actually increase your parallelism that you can extract from the architecture.

So overall, we are always keeping ourselves restricted to the domain of independent kernels and not going to think of using dependent kernels. Then also, we figured out that okay, the kernels should be fusing, there should be suitable amount of load balancing, the data size or the data arrangement should not be too much different. Again the similar thing will also hold here that we will be fusing independent kernels, but the kernels, it should not have that they have this similarity in terms of computation time.

Let us look at the situation why is it so. So then, while some of threads we are executing this, let us kernel 1 is a heavy kernel and kernel 2 is a light weight kernel. So when you launch the threads, the threads executing kernel to finish this much faster, but these threads are still waiting. So that does not make sense then, because fusion of these threads are not making sense, because you have got these threads, which are finished.

But waiting at the end of the execution, while the threads have not ended their execution, so entire kernel is still live in the GPU, while some of the threads are still waiting. So then you are really not exploring the parallelism in the nice way, because otherwise you could have done a fusion of these workloads with some other workloads and that would have really led to a bit more balanced execution, because overall what is your target?

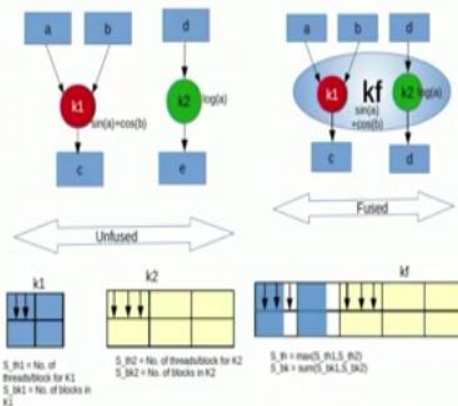
Your target is to increase the occupancy of the GPU and keep on using all the architectural elements, the compute units, the memory elements, the memory bandwidth in parallel as much with as much high through put as possible. So if the independent kernels have similar computation time, then this is a good idea, otherwise not. So for our case, let us understand what should be our options for fusion.

So let Sthi represent the number of threads in a thread block and Sbki represent the number of blocks in a kernel. So when I am fusing them, so Sth would be the maximum of Sth1 and Sth2. So when I am fusing kernels, Sth would be the maximum of Sth1 and Sth2, because I am not going to increase part thread activity. All I am doing is, I am increasing the number of blocks. So the number of threads per block remains the same.

So it should just be the max of the originals and the number of blocks will be the sum of the number of blocks in each of the constituent kernels. So again, just to remember that this is not a good idea, if workload for different thread blocks differ a lot. That means, the kernels differ a lot in terms of their computation ahead it. So there are nice interesting points to take. For example, consider that kernel 1 is of compute intensive kernel, kernel 2 is a memory intensive kernel, is it a good idea to fuse them?

These are foods for thought, you can actually look into, you can try simulating the example of two such kernels. Consider induction kernel, consider some other kernel together, maybe a convolution kernel and think whether fusing them makes sense or not.

(Refer Slide Time: 25:34)



Now if we come to the inter block fusion example, we are again restricting ourselves to the original kernels. So we have the unfused versions on the left and the fused versions on the right. So as you can see, again we are considering the general case, where we have unbalance in terms of the number of threads per block and also unbalance in the terms of number of blocks, but now things are quite easy, all we do is, we increase the number of blocks and we delegate some of the blocks exclusively for activities of kernel 1.

And we delegate some of the blocks exclusively for the activities of kernel 2, so that would mean, we will now have, since in the original kernel there were two threads per block k1 and there were two threads per block for k2. In this fused kernel, we are having three sets per block in general. So when we are fusing them, first of all you see that, this is essentially nothing but k1's working area. So all the threads belong to this area are going to do the job for k1.

The threads, which are belonging to this area, that means or correct me, the blocks which belong to this area, they are resident threads. We will be doing exactly the computation for k2. So just to recall that, all we are doing is we are figuring out the block ID. If the block IDs belong to k1's block IDs, then they are going to do the activities for k1. For example, if this is the block ID 00, this is 01, like that, or this is 10, this is 11, those are the block IDs for which I will have k1 working.

For the other block IDs, I will have k2 working. So whatever was 00 here, if the block ID is greater than k1's block ID, just immediately greater than k1's block ID, for that it should be this block of k2 and so on, so forth. These are essentially nothing but k2's blocks and these are essentially nothing but k1's block, but we discussed the point of loading balance in terms of the computational natures of the kernels, whether k1 is a heavy kernel or k1 is a light kernel and k2 is a heavy kernel, but also observe the other point, we mentioned here.

So first was whether the kernels have similar computation time, that was the loading balance issue, also observe that here when we are looking into this kernel k1, the number of threads per block is now max, max of k1 and k2. So we have three threads for k1 as well as three threads for block for k2. But since k1 is originally utilizing two threads, so this third when it is belonging to the block IDs for k1 does not have much to do.

So these are just threads which are getting launched without any activity. So this is also a wastage of thread, which is running in parallel or it has got no activity. So if we look into the program example here, the fusion of independent kernels we are considering here. The fusion style is inter block fusion, but we are considering for different data space size. So these are my original unfused kernels.

So, of course, first we figure out what is the Sth and Sbk. So Sth is max and Sbk is sum. So we also have similar comments for the previous kernels. They are kind of repeated here. So I am just talking about them here, where it is a bit more complicated. Assume that Sth2 is greater than Sth1 and Sbk2 is greater than Sbk1, of course, you have to write suitable programs for that.

(Refer Slide Time: 29:32)

Inter Block Fusion Example

```
kf(a, b, c, d, e, n1, n2):
b = blockId
t = threadId
i = global threadId

if(b<S_bk1)
        if(t<S_th1)
                c[t]=sin(a[t])+cos(b[t])
else if(b<S_bk)
        if(t<S_th2)
                e[t]=log(d[t])
```

Now when I have this fused kernel, first thing I will do is, I will figure out what is the block ID. I will figure out what is the thread ID and I will also figure out the global thread ID and then we will use the block ID first to figure out whether this thread belong to a block, which is for kernel 1 or kernel 2. So coming back here, again they will just repeat. So for the fused kernel, we have figured out what is Sth and Sbk and we are just considering.

The code is written in such a way that we are assuming that Sth2 is greater than Sth1. So of course, you can understand if for your second kernel, we are assuming that the number of threads per block is larger. If it is not so, just you have to switch the variant of the code. I hope that is clear. Here, the code is written assuming between k1 and k2, I have this property, otherwise your code has to be changed. I thought you should be able to do that.

So first you check, whether the block ID is less than Sbk1. If so, then you get inside and check that whether inside this block, you have some activity for the kernel 1 or not, because if so, you just check whether the local thread ID is less than Sth1, then you execute this code. Do I have an example where this will not execute? Yes, for example if you come here, consider a thread this one. For this thread, the block ID is less than Sbk1. So it will execute kernel 1's code.

But now, the thread ID is not less than Sth1. So it will fail the if case and it will not do any activity. So for the threads with the previous IDs, they will get inside this part and do the kernel

1's activities. Those which will fail this, they are the idle threads. They do not have anything to do. Otherwise, we will just check whether the block ID is beyond Sbk1, but inside Sbk, then I know that this thread belongs to kernel 2.

Again, we will just have the check whether it is less than Sth2, the thread ID. In that case, we will delegate the activity for the sets to kernel 2's activity. Now this inter block fusion also has limitation. So what are the limitations? So workload of different thread blocks, consider that, I mean, they may differ a lot. We are just considering the situation that suppose the workload of different thread blocks will differ a lot, then this may not be a good idea.

So that was what we are speaking about, like if the kernels have different computation time and also internally, if the threads per block differ a lot. If the threads per block differ a lot, then we will have a lot of idle threads like this.

(Refer Slide Time: 32:33)

Inter Block Fusion Limitation

Workload of different thread blocks differs a lot. For example, below two kernels are not suitable for this type of fusion.

```
k1(a, b, c, n1):
        i = global threadId
        c[i]=sqrt(sin(a[i])+cos(b[i]))
k2(d, e, f, n2) :
        i = global threadId
        f[i]=d[i]+e[i]
```

And if the activity of the block, the amount of all thread too differ a lot, then some of the thread blocks will finish, but some of the thread blocks will wait and that will again create a load imbalance. So if you just check here, for k1 and k2, for k1 I have this is a heavy kernel, because all it is doing is, it is doing a square root of sin a + cos b in a part thread manner. So it is going to actually use the special function units in the Sm.

Hope you remember the special function units, which we are actually going to do the transfer data functions, the operations there. So they will actually compute the sin, the trigonometric series, the mathematical expressions, the sin trigonometric series approximated for cos and finally with the sum, it will implement Sqrd operation. So these are highly costly mathematical operations, which are going on, whereas here I have a simple single cycle addition.

So these kernels are highly imbalanced with respect to their load. Due to this absence of workload balancing, they are not good candidate for fusion, because fusing them I do not gain in terms of concurrent execution of the kernels. Because one kernel finishes much faster, the blocks from that kernel, the other kernel's blocks are still executing. So there is not much of concurrence.

(Refer Slide Time: 34:03)

GPU Optimization techniques

Some examples of GPU Optimization techniques-

- Reduction
- Fusion
- **Coarsening**

So with this, we will end this lecture and in the next lecture, we will start with another optimization technique, which is thread coarsening. Thank you for your attention.