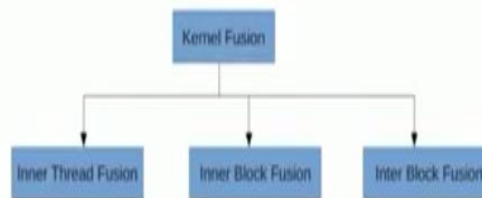
GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Lecture - 36
Kernel Fusion, Thread and Block Coarsening (Contd.)

Hi, welcome back to the lectures on GPU Architectures and Programming. So in the last lecture, we have just started the discussion on kernel fusion.

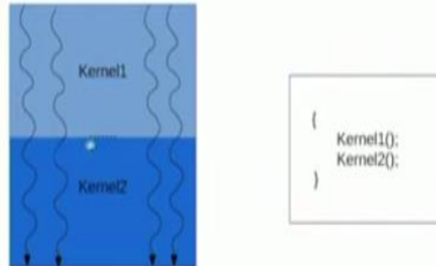
(Refer Slide Time: 00:32)



One of the possible optimizations in GPU programs, and we have classified the different kinds of fusion optimizations that are possible. Just to recall, they were inner thread fusion, inner block fusion, and inter block fusion.

(Refer Slide Time: 00:47)

Inner Thread Fusion

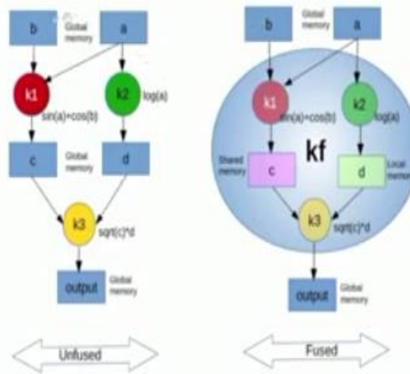


So let us get into the detail of what is inner thread fusion. So essentially, when we speak about inner thread fusion, suppose I have kernel 1 executing on some data segments and computing some output, which is to be then again passed to kernel 2. So consider that there is a dependency graph, where you have an input buffer. So this data stream would be operated by kernel 1, it would be returned to some output buffer and then it would be again read by kernel 2 and then it would be output again to some output buffer.

All we are doing is essentially just fuse this part, create one kernel, and let each of the threads perform the operations of kernel k1 followed by the operations of kernel k2. So essentially we are allocating the job per thread from kernel 1 followed by kernel 2 to a single kernel, where each thread performs the operations of kernel 1 followed by kernel 2. So if you remember the example we took earlier, let me just go back to that task graph here.

(Refer Slide Time: 02:15)

Kernel Fusion example



This was the task graph of kernels and we fused all of them and this was my version of the fused kernel.

(Refer Slide Time: 02:23)

Code snippet for fused kernel

```
__global__ void
process_fused_kernel(float *a, float *b, float *output, int datasize) {

        __shared__ float c[blockDim.x * blockDim.y * blockDim.z];
        float d;
        int blockNum=...
        int threadNum=...
        int i=...
        if (i<datasize) {
                c[threadNum]=sin(a[i])+cos(b[i]);
                d=log(a[i];
                output[i]=sqrt(c[threadNum])*d;
        }
```

As you can see that, the final fusion is at the thread level. We compute the global thread ID. We examine whether it is inside the valid data size, then I just perform part thread, the sequence of operations, which were to be performed by each of the three kernels individually. I have just written down those operations in a sequence. Without violating the sequence that was already provided by the partial order, that was enforced by the original kernel dag.

So this is in a nutshell the idea of kernel fusion, when we do inner thread fusion. So you just write a part thread activity, which is a sequence of the two kernel codes.

(Refer Slide Time: 03:07)

Inner Thread Fusion

- Combines computation of two kernel into single thread
- Suitable for both dependent and independent kernels if dataspace size is same
- Let, $S_{th,i}$ represents the size of threads in a thread block $S_{bk,i}$ represent the size of blocks in kernel $i(i = 1, 2)$
- For fused kernel -
  - $S_{th} = max(S_{th,1}, S_{th,2})$
  - $S_{bk} = max(S_{bk,1}, S_{bk,2})$
- Not suitable if -
  - Kernels not having same thread/block space
  - Results in unbalanced workloads between threads

Now the issues is how do I decide on the threads per block and the number of thread blocks while designing an inner thread fused kernel. So you are essentially combining the computation of the two kernels at the single thread level. It is suitable for both dependent and independent kernels. So dependent kernel would mean that the first kernel's output is the input for the second kernel, as we saw in the task graph. So every edge in that graph is like a dependency edge.

So suitable in that case and it is also suitable for independent kernels, but there is a constant, that the data space size has to be sent, so that we can use the same data space size to define identical thread blocks for the fused kernel. So consider Sth, i represents the size of threads in a thread block. So that is the thread block size and Sbk, i represents the size of blocks in the kernel or essentially I would say the number of blocks in the kernel.

Let us just write this, just represents the size of thread block. So this is the number of thread per block and Sbk, i represents the number of blocks in kernel, I write. So when I fuse them, I have to set the number of blocks and number of threads per block. I need to identify, what is my number of blocks and I need to identify what is my number of threads per block.

So when I fuse the two kernels to create a single kernel, I choose Sth, which is the size of thread block as the max of the individual thread block sizes and I choose the number of blocks as the maximum of the number of blocks in the individual kernels. Now this is important because of course, when I am fusing two different kernels, they may be considering different data space sizes, I mean they may be considering different ways in which the threads are arranged in the data space, although may have been the data space are just same.

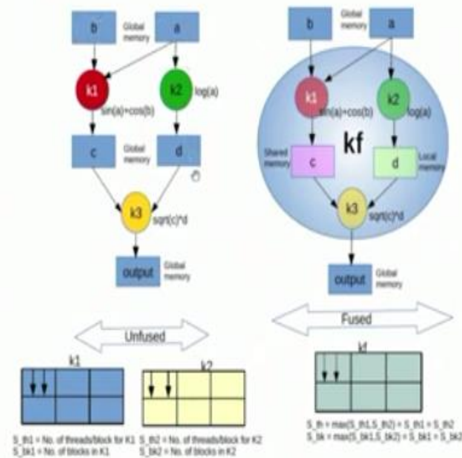
So I write the fused kernel, I have to choose Sth, the thread block size and I have to choose Sbk, which is the number of blocks. So in that way, when I am doing an inner thread fusion, I set these parameters by doing a max in each case and the issue is this is not a good optimization in case the kernels do not have same thread or block, I mean, the arrangement of threads and blocks, and threads per block space is very different.

So if the kernels are not having the same threads per block in the data space, then it is not a suitable optimization. Primarily, the reason is that it would result in an unbalanced issue of workload, I mean the kernels different thread blocks, we will have different amount of work to do and that would be issue with respect to exploiting the parallelizing of the GPU in general.

So we will actually prefer this kind of an optimization, if the number of blocks and number of threads per block, primarily the number of threads per block are kind of similar across the different kernels. Let us move to an example and try and figure out why this is really so.

(Refer Slide Time: 07:48)

Inner Thread Fusion Example



Consider this situation, so I am doing an inner thread fusion. So this is original unfused version and this is the fused version. Now this is an example where we are showing that the number of Sth1, the number of threads per block, that is 2 and Sbk1, number of blocks is 6 and we are just considering a very ideal situation, where everything is same for k1, k2 and I am just fusing them to create the fused kernel kf with Sth and Sbk as the number of threads per block and the number of blocks. So it is just a max of Sth1, Sth2 and max of Sbk1 and Sbk2. Since they are all same, so I get the same values here.

(Refer Slide Time: 08:42)

Inner Thread Fusion Example

Fusion of dependent kernels with same dataspace size and thread/block size

```
//Unfused kernels
k1(a, b, c, n):
        i = global threadId
        c[i]=sin(a[i])+cos(b[i])
k2(a, d, n) :
        i = global threadId
        d[i]=log(a[i])
//Fused kernel
kf(a, b, out , n):
        local c,d
        i = global threadId
        if(i<n)
                c=sin(a[i])+cos(b[i]);
                d=log(a[i];
                output[i]=sqrt(c)*d);
```

So when I am going to fuse, the important thing here is these kernels, if I am considering k1 and k2, they are independent, but then k3 depends on k1 and k2. So it is not that all the kernels are

independent. They have some ordering as is enforced by the dag structure here. So as long as we are fusing dependent kernels with the same data space size and threads per block size, the code is very simple.

So let us say these are the unfused kernels and we are just trying to provide. So as you can see that we are just providing a structure of the kernel code, let us say k1 structure is like this that you just compute the global thread ID. Again, this is not the valid kernel code, we are just providing a structure of k1 and structure of k2 here. So for k1, you compute the global thread ID, do the operation for k2, you compute the global thread ID and do the operation.

When you fuse them, you will have some local variables to hold intermediate results. So for that you should have some more variable c and d here. Compute the global thread ID as long as it inside the data space size. You compute the first operation of k1, store it in local variable c, compute the operation of k2, store it in local variable d, use c and d to compute the final output. So you use c and d to compute the final output, which would be square root of the first kernel's output multiplied by the second kernel's output.

That is the final value, which is the final fused value. So this was a very easy piece of code, simply because the threads per block and the number of blocks are all same across the kernels. So things look very simple here.

(Refer Slide Time: 11:07)

Inner Thread Fusion Example

Fusion of independent kernels with same dataspace size and thread/block 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
kf(a, b, c, d, e, n):
i = global threadId
if(i<n)
        c[i]=sin(a[i])+cos(b[i])
        e[i]=log(d[i])
```

So now consider this simple situation, where you have the same data space sizes, but now we have some different data space size, but same threads per block size. So again I am just recapitulating here. This is the inner thread fusion example and this is the very simple situation and this picture is just showing. So unlike this program, which was of total fused kernel, this picture is just showing how the k1 and k2's fusion will look like, very similar, just missing the code for k3 here.

But this is very simple and the fusion is quite simple for k1 and k2. So difference with this is here I also have k3 in the final output. Here I am just showing the example of fusing k1 and k2 here. The two simple operations are done, but now consider the situation where you have a big difference in the data space size. So suppose the situation is that you have same number of threads per block, but you have a difference in the number of blocks.

So that is what the picture is showing to convey. So k1 is operating like this. So for k1 the block structure is this. You have two threads per block, in total you have 6 blocks, but k2 is operating on a larger size d here. So here, for k2, you have this 4 blocks, but the threads per block is same. Now how do I fuse them? Because I am going to produce the output c and e. So as you can see, we have now taken an example where the data space size is varying.

But since I do not have k3 in this example, I just have k1 and k2, I do not have a problem, because k1 and k2 are independent kernels. So as long as they are independent kernels, I can always use them together, even if the data space size is different, as is the case here. The data space size is different here, but in this example we have the same structure of threads per block. That is also important in this case.

Because since the thread per block is same, it does not change even in the fused kernel, if I follow the formula, it is equal to whatever was for the k1 and whatever was k2, is same, but since I am doing a max over block size, the number of blocks is now same as the kernel with the larger block size, the number of blocks. So when I am fusing the kernels k1 and k2, which are independent for the output kernel, the kf, I have the larger number of blocks, where the thread per block will be same as the two origins.

So I hope the point is clear. When we are fusing all the three kernels together, we had the requirement that data space size has to be same, but now since k1 and k2 are independent kernels, I do not have the requirement because k3 is not there to work on the final output and since they are independent, they can actually have different data space sizes, which is the situation here and when I am fusing them, so it is again quite simple.

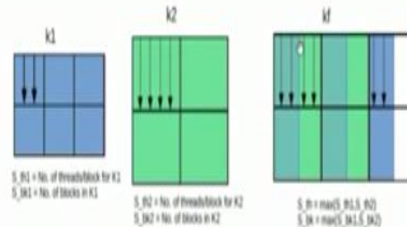
So i is less than n, then I have this sequence of values. So the program automatically takes care of the situation, that since I have considered the larger number of blocks in my definition of kf. So automatically, the fused kernel will launch the number of threads similar to whatever is required for processing. It is going to launch the number of threads which is similar to the requirements of k2, which had the increased number of blocks and so there will be enough threads to process the data buffer d.

Apart from some of the threads, which are lying in these blocks, all the other threads will also be processing the data buffers from a and b, as was the requirement for k1. So I hope this is clear. Since the two kernels are independent, the fused kernel code is also very simple in this case.

(Refer Slide Time: 15:44)

Inner Thread Fusion Limitation

Fusion of independent kernels with different data size and thread/block size:
NOT SUITABLE: Due to unbalanced workloads between threads



Now coming to the limitations of the inner thread fusion, if I am fusing independent kernels with both different data size and threads per block size. Now it may be unsuitable in this case, again I will just come back here. Here, we had the number of blocks differing across independent kernels, but the number of threads per block was same. That was a good thing and that lead to a very simple code. So this was my code for the inner thread fused kernel.

Sorry, we keep this earlier. This is my code of the inner thread fused kernel considering difference in the overall data size, due to the difference in the number of thread blocks, but both of them having the same threads per block size. So as you can see, that you compute the global thread ID and then you have to see that out, whether the global thread ID would be inside the range of both the kernels, or is it outside the range of kernel 1, but inside the range of kernel 2.

If it is inside the range of both kernels, which is n1 here, then I should actually compute whatever is the operations for both k1 and k2. So then, I am inside thread ID i, which is lying in this common region and otherwise, I will go to other region, where I have to only execute the port for kernel 2. So again we will just recall that these are independent kernels with unbalanced number of blocks, but same amount of threads per block.

So the different data size is there due to the unbalanced number of blocks. Now consider the situation where I have different number of blocks and also different number of threads per block.

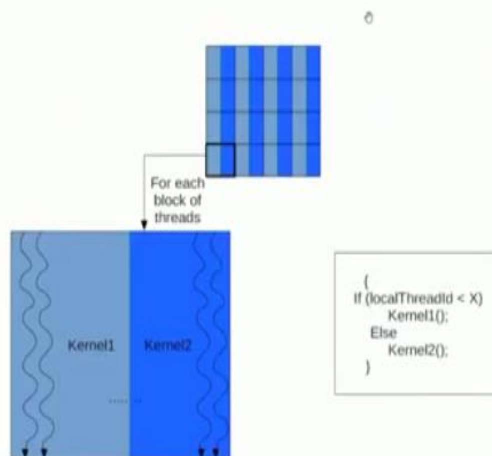
Now that may be bad thing, even for independent kernels, because consider k1 and k2 with these arrangement. So 2 threads per block and 6 blocks and 4 thread per block and 4 blocks. So what are the kf I am going to have 6 blocks and in 6 blocks, some of the blocks, I have full fusion.

So I have 4 threads. In some of the blocks, I have only 2 threads because if we notice the first block, I should have 4 threads, 2 of the threads will execute code for k1 as well as k2, 2 of the threads will execute only the code for k2 and the same pattern will continue for the other three blocks also here. I hope this is clear. Why these may not be a good solution in this case?

First of all the classification of this case is unlike the earlier case, where I had same number of threads per block, but varying blocks, here I have varying threads per block as well as varying number of blocks. Now due to that, when I am fusing, due to the issue of varying number of blocks, I will have some blocks where there is no activity of kernel 2, which has less number of blocks. So such blocks are these 2 blocks. For the other blocks, I have unbalanced execution.

For example, all these 4 blocks, as you can see, there are 2 threads. So each block has 4 threads, which is the max of the threads per block of k1 and k2 and inside these 4 threads, there are 2 threads, which are executing code for both k1 and k2 and 2 threads, which are executing the code only for k2. So this leads to a lot of unbalanced execution, which may not be good.

(Refer Slide Time: 19:36)



Now this leads us to another possible way of fusion. I hope by now, this idea of inner thread fusion is clear to you and this would actually lead us to the other possible ways, in which we can fuse and they are inner block fusion and inter block fusion. So before going into them, let us again summarize the good and bad things of inner thread fusion.

So when you do a thread level fusion of two different kernels, if they are independent kernels with differences in both the number of blocks as well as threads per block size, it may lead to a bad situation, completely unbalanced workloads and that may not be suitable. In some cases, it is okay. For example, the data size is different, but because the number of blocks are different, the threads per block are same.

So in that case, the code does not have too much operate and there is significant amount of balance in execution of the majority of the blocks. This was the picture for that. And when I considered the total fusion of the blocks, our demand was that okay, there should be absolutely same data space size and thread block size and we now coming back to this part, it would make sense. So this is the situation where I am fusing independent kernels.

As long as I consider only k1 and k2, they are just independent kernels. If we go back to the earlier example, which was this. So here I was using k1, k2, and k3. So there was also dependency, which means, I can start the code for k3 only when k1 and k2 both finishes. Now the issue is as long as I have identical number of threads per block and identical number of blocks, fusion of dependent kernels is simple in this case while considering inner thread fusion.

But now as we have understood now, what is the implication of differences among independent kernels. So this was my picture of independent kernels getting fused, where they have differences in number of blocks as well as threads per block. So we understood what is the issue here? Now one can easily visualize that how the issue becomes even difficult when you are trying to use this concept of fusion in the case where you have dependencies among kernels and there is no balance in terms of equality of threads per block and the number of blocks, while executing dependent kernels, if you are going for fusion.

So as you can understand that if I am trying to fuse this with identical number of threads per block and number of blocks, things are very simple. my checks are very simple, just check on the global thread ID and execute all the elements, operations and sequence. If there is difference in the number of blocks in some of the kernels, it would be really difficult here. I need to do more fine grain control by introducing more number of these statements and that would be further complicated, if I have also variation in threads per block setting also.

That is why, in case of dependent kernels, it is even more complicated and we had an idea that by even looking at the situation for independent kernels. Because as we saw, as long as it was independent kernel with everything same, that is I have same number of blocks and same number of threads per block, that is the same data space size and same threads per block, then the code was so simple, just a check on whether I am inside the data space, the moment we introduced one difference, that is okay, let the threads per block remain same.

But let there be more number of blocks, so that there is a difference in the data size that introduces one "if" statement to check whether I am inside the common blocks or I am in the blocks only relevant to k2. The more I am going into this mess of the differences in threads per block as well as differences in number of blocks, it is only going to introduce more and more of conditionals, even for dependent kernels and the total program will get much more complicated with more primitives required in case of dependence.

Sorry, this is already sufficiently complex in case of independent kernels and that would require additional primitives, if I am going to use this concept in case of dependent kernels where I also have variations in threads per block as well as block size. So from this, the point to take home would be that when I am trying to fuse independent kernels, we should take care that at least there is similarity of threads per block size, even if there are differences in block size.

But if we have differences in number of blocks as well as threads per block, there may be significant absence of balance in the workload in the fused version and if we bring in the added complication of dependency among kernels, it would require more primitives, which would

complicate things even further. So we understand now when inner thread fusion is absolutely fine, as long as there is full identical arrangement of the data space across the different kernels.

Even if that is not the case, if at least threads per block remains same, still things are okay for the case of independent kernels, and when I have variation in Sth as well as Sbk, then it is very difficult to achieve workload balance among threads for inner thread fusion even for independent kernels and it is better in those cases to look for other optimizations in respect to fusion, like inner block fusion or inter block fusion.

So with this, we will be ending this lecture and in the next lecture, we will be looking into more details of inner block fusion and that would be all for this lecture. Thank you.