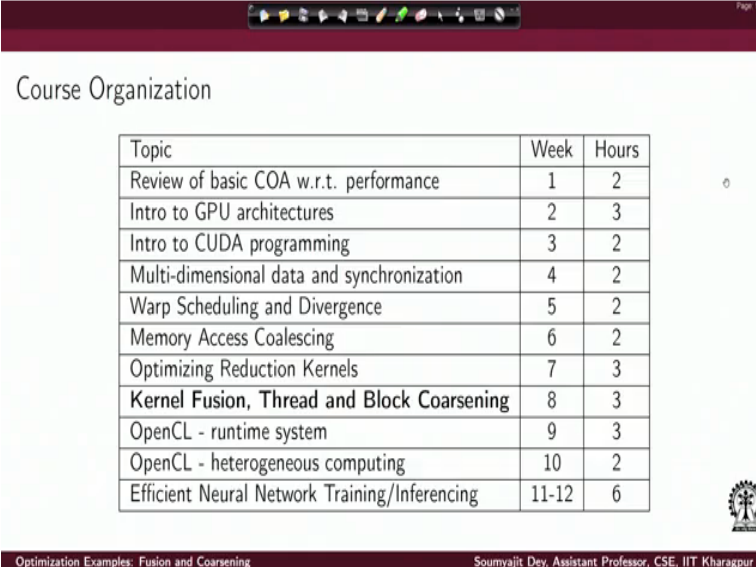


GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Lecture – 35
Optimizing Reduction Kernels(Contd.)

Hi, welcome to the lecture series on GPU architectures and programming. So today we will be moving over to a new topic which is some different other kinds of optimizations primarily the optimizations will be discussing are fusion or fusing multiple kernels and coarsening of threads in kernels. So they are popularly known as fusion and thread coarsening has two possible optimization techniques.

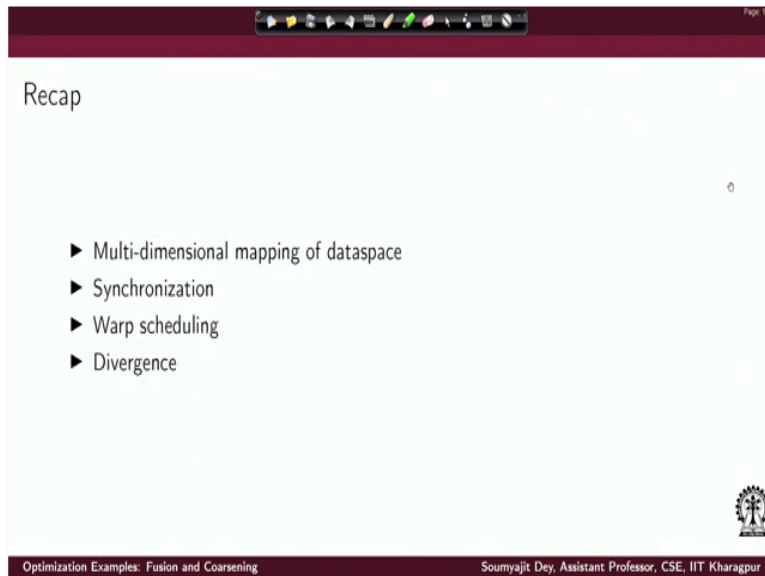
(Refer Slide Time: 00:54)



Topic	Week	Hours
Review of basic COA w.r.t. performance	1	2
Intro to GPU architectures	2	3
Intro to CUDA programming	3	2
Multi-dimensional data and synchronization	4	2
Warp Scheduling and Divergence	5	2
Memory Access Coalescing	6	2
Optimizing Reduction Kernels	7	3
Kernel Fusion, Thread and Block Coarsening	8	3
OpenCL - runtime system	9	3
OpenCL - heterogeneous computing	10	2
Efficient Neural Network Training/Inferencing	11-12	6

So coming to this topic on kernel fusion and thread and block coarsening.

(Refer Slide Time: 00:59)



Recap

- ▶ Multi-dimensional mapping of dataspace
- ▶ Synchronization
- ▶ Warp scheduling
- ▶ Divergence

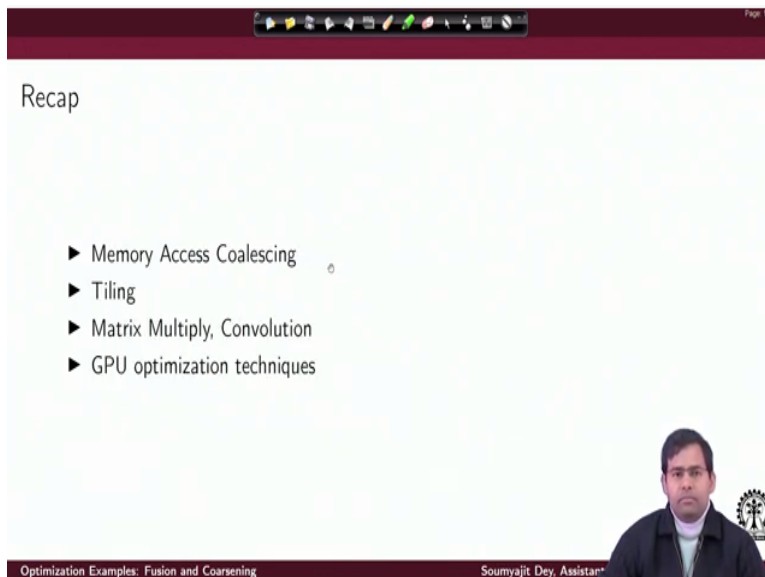
Optimization Examples: Fusion and Coarsening

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

This slide is a presentation slide with a dark red header and footer. The main content area is white. It features a list of four topics under the heading 'Recap'. The footer contains the text 'Optimization Examples: Fusion and Coarsening' and 'Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur'. There is a small logo in the bottom right corner of the slide area.

So this will actually require you to do a small recap of the following topics like how to do a multi-dimensional mapping of dataspace, concept of synchronization of threads concept of warp scheduling and divergence.

(Refer Slide Time: 01:17)



Recap

- ▶ Memory Access Coalescing
- ▶ Tiling
- ▶ Matrix Multiply, Convolution
- ▶ GPU optimization techniques

Optimization Examples: Fusion and Coarsening

Soumyajit Dey, Assistant

This slide is a presentation slide with a dark red header and footer. The main content area is white. It features a list of four topics under the heading 'Recap'. The footer contains the text 'Optimization Examples: Fusion and Coarsening' and 'Soumyajit Dey, Assistant'. There is a small logo in the bottom right corner of the slide area and a small video inset of a man in the bottom right corner.

And also how memory accesses get coalesced across I mean across different thread inside a warp and the concept of tiling and also we will see that the primary workloads will be talking about here would be like matrix multiplication convolution which are the workloads on which we maybe focusing a bit like that.

(Refer Slide Time: 01:43)

GPU Optimization techniques

Some examples of GPU Optimization techniques-

- ▶ Reduction
- ▶ Fusion
- ▶ Coarsening

Optimization Examples: Fusion and Coarsening Soumyajit Dey, Assistant*

(Refer Slide Time: 01:48)

Loop Fusion

- ▶ Classical compiler optimization in programming
- ▶ Improve performance by reducing off-chip memory traffic
 - ▶ reduction of cache miss
 - ▶ better control of multiple instruction
 - ▶ reduces branching condition
- ▶ Operates by fusing iterations of different loops when those iterations reference the same data

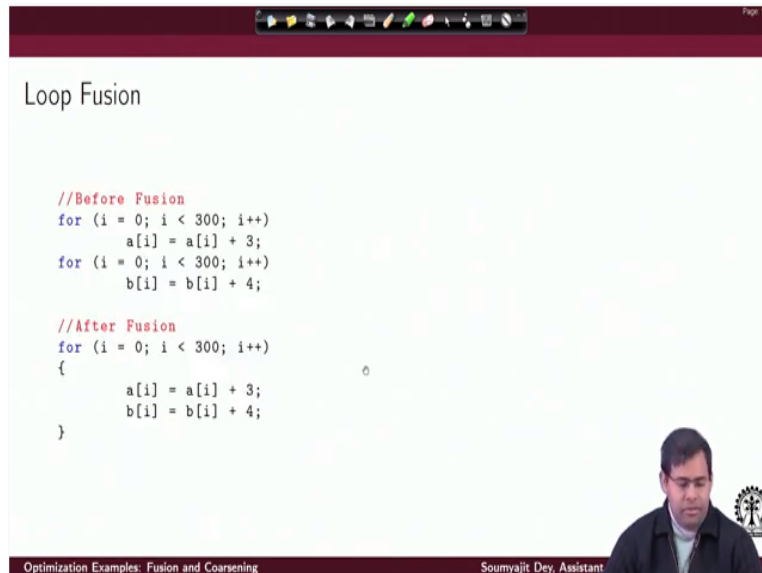
Optimization Examples: Fusion and Coarsening Soumyajit Dey, Assistant*

So getting into this topic of fusion let us just do a recap of what is classical loop fusion? So in this is essentially a classical compiler optimization which helps to improve performance of a program. The primary reason for the performance benefit is that if I do a loop fusion operation it may be possible to reduce the off-chip memory traffic. Essentially you reduce the number of accesses and you can reduce the cache miss.

You can also warp I mean obtain better control on multiple instructions. And also it helps in reducing the number of branch condition checks that are required. And we will see that why that is so we will just refuse it this idea of loop fusion in this case. So the way fusion of loops work is

that you have 2 independent loops. You fuse the iterations of these different loops and it helps in case the iterations are such that they are referencing the same data element.

(Refer Slide Time: 02:58)



```
Loop Fusion

//Before Fusion
for (i = 0; i < 300; i++)
    a[i] = a[i] + 3;
for (i = 0; i < 300; i++)
    b[i] = b[i] + 4;

//After Fusion
for (i = 0; i < 300; i++)
{
    a[i] = a[i] + 3;
    b[i] = b[i] + 4;
}
```

Optimization Examples: Fusion and Coarsening

Soumyajit Dey, Assistant

So if we look at this kind of an example. So here we are just trying to give an example of loop fusion. We are not saying that in this case I will obtain a usefulness which is exactly that we are referring the same data element or not. But let us see what is the advantage we get? So this is a simple loop fusion example. We have 2 loops here. As you can see in the first loop i iterate 0 to i mean i iterate a total of 300 times.

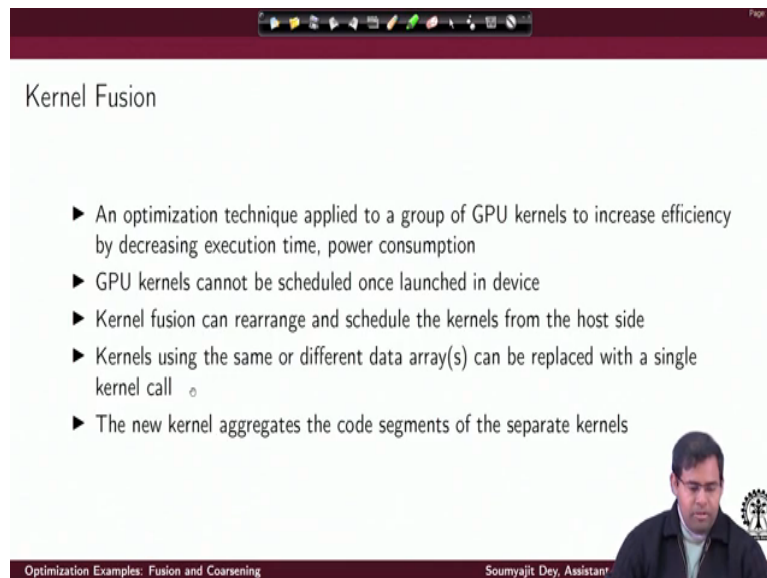
Again in the second loop i again iterate in total of 300 times while in each iteration in of the loops I am accessing the array elements of ai or bi. So alternatively I can fuse the loops and I can perform these updates of ai and bi together inside a single loop right. So what is the good thing about this fusion here. If you check the original code which was before fusion here we are having the condition checks in the first for loop as well as the condition check separately in the second for loop that gets avoided right.

So that is the advantage which we are leveraging in this case. Maybe in case of other examples which we will soon see that suppose I have 2 loops which are effectively I mean in the same iteration they are effectively referencing the same data then it does not make sense to bring to do

that in separate loops and we can just bring those computations inside the same loop provided there are no dependencies and the main thing that holds is in the same iteration same value of i .

I am accessing the same data a_i and maybe a_{i-1} stuff like that in the 2 loops right and that is what we fuse and we get an advantage. We just went to cache access which is fairly understood.

(Refer Slide Time: 04:47)



Kernel Fusion

- ▶ An optimization technique applied to a group of GPU kernels to increase efficiency by decreasing execution time, power consumption
- ▶ GPU kernels cannot be scheduled once launched in device
- ▶ Kernel fusion can rearrange and schedule the kernels from the host side
- ▶ Kernels using the same or different data array(s) can be replaced with a single kernel call ◦
- ▶ The new kernel aggregates the code segments of the separate kernels

Optimization Examples: Fusion and Coarsening Soumyajit Dey, Assistant

Now coming to our specific topic of fusion when we are talking about GPU kernels. So again before getting into this I would just like to tell you this is just a very simple representative example. It does not highlight all the advantages of loop fusion. It highlights a few. But you can do a study of compiler literature or program optimization literature which is classical and find there are lot of advantages that loop fusion actually brings into play okay.

In today's lecture we will primarily be studying this idea of loop fusion from the perspective of GPU kernels. We will study that how the fusions can be done. What are the different possible ways to do the fusions? I mean in terms of handling the fusion at the thread level or the block level and we will talk about architectural implications and other things in detail later on.

So kernel fusion is basically the kind of fusion where we are going to group a set of GPU kernels together and rewrite a single kernel. And the idea of using this is that if I fuse multiple kernels together it may potentially increase the efficiency by decreasing execution time, power

consumption etc. Now why is this advantages? Let us first discuss certain issues that we have with GPU kernels.

For example, whenever I am launching a GPU kernel after setting up suitable launch parameters. I have no control on how the different threads get scheduled right. It is all where I mean it is all decided by the hardware scheduler. So I have no control on how the threads gets schedule and typically when 1 kernel is running I wont have the flexibility of running launching another kernel as long as the kernel 1 which was running does not finish.

In modern days NVIDIA and other support concepts called concurrent kernel execution through which I can launch multiple kernels together. But still I would say that we are not having absolute control of how the different kernel threads, threads belonging to different GPU kernels gets schedule once launching the device right. Now this is something on which you can have final control if you perform a static optimization like kernel fusion on the input kernels.

So what can be done is inside the kernel fusion operation I can rearrange and schedule multiple kernels from the host side right. That I will be able to control that which threads in which kernel execute fast for which thread in which kernel later on like that. If I fuse them following some suitable semantics. So this is something we like to see that how such fusion of kernels can be done.

Also the other idea would be that if I am using kernels with same or different data arrays they can be replaced with a single kernel call that is the fundamental thing. Instead of having multiple kernel launches I am going to launch 1 carnal if I am fusing multiple kernels together.

(Refer Slide Time: 08:08)

Advantages

Increase efficiency by -

- ▶ data reuse using on-chip memory improves performance
- ▶ reducing off-chip memory data traffic
- ▶ reducing global memory data transfers
- ▶ reducing kernel launch overhead
- ▶ utilising maximum threads in GPU

Optimization Examples: Fusion and Coarsening Soumyajit Dey, Assistant

And we will first study what are the advantages out of fusing multiple kernels together and writing a fused kernel. It may happen that in case I have data reuse using on-chip memory and if that is a property of the individual kernels that they actually are accessing data from the same memory from the same data elements. Then by fusing them I am able to leverage the reuse of the data through the hierarchy of cash and shared memory on primarily cash as there is on-chip memory to the primary and also the shared memory which is there in the SM through the and through the memory hierarchy of the GPU and that would improve the performance right.

Now of course there are side effect what happens is that if I am making better use of the on-chip memory I reduced the off-chip data traffic and that is going to provide me essentially that is what we mean by reducing the global memory data transfers. The number of data transfer from the global memory will reduce and that reduces the kernel launch over it right. Because I instead of launching multiple kernels as we have repeated that I am launching less number of kernels.

Because each time when you are launching a kernel you have to set up launch parameters. You have to set up the GPU system run time system of CUDA will set up internally suitable data structures and that also consumes space and time. So that is the kernel launch overhead which you can bypass if you are launching a single kernel by fusing 2 or more number of individual kernels which the original programmer may have written right.

Now the other advantage that may happen is you can utilize maximum threads in the GPU that means suppose you are launching single kernels at a time right and each of the kernels are not completely occupying all the SPs in the GPU. Then you are not really using the full parallelism offered by the GPU instead you can perform suitable fusion and utilize maximum threads in the GPU.

(Refer Slide Time: 10:21)

Limitations

Kernel fusion does not always result in performance improvement:

- ▶ architectural resources like the capacity of on-chip memory and registers are limited
- ▶ reduction of off-chip memory data traffic is feasible upto a certain limit
- ▶ overhead in identifying fusible kernels
- ▶ overhead in defining a scalable method to search for the optimal rearrangement of fusible kernels
- ▶ may introduce divergence

Optimization Examples: Fusion and Coarsening Soumyajit Dey, Assistant

Now what are the limitations? Now this kernel fusion may or may not result in performance improvement right. So when does it not result in performance improvement for example let us also understand that architectural resources like the capacity of on-chip memory and registers inside each of the SMs they are limited in number right? If I fuse kernels, then essentially each kernel would place the higher requirement of on-chip memory and registers right.

That is good that does not help performance as long as I have enough on-chip memory and registers to support the execution of a fused kernel. But in case the requirement becomes too high then the fusion is not really advantageous and it may lead to further overheads and reduction in speeder right. Now again in a similar way we will have all the other points. For example, this idea that I would get reduction in off-chip memory traffic that is also feasible up to a certain limit.

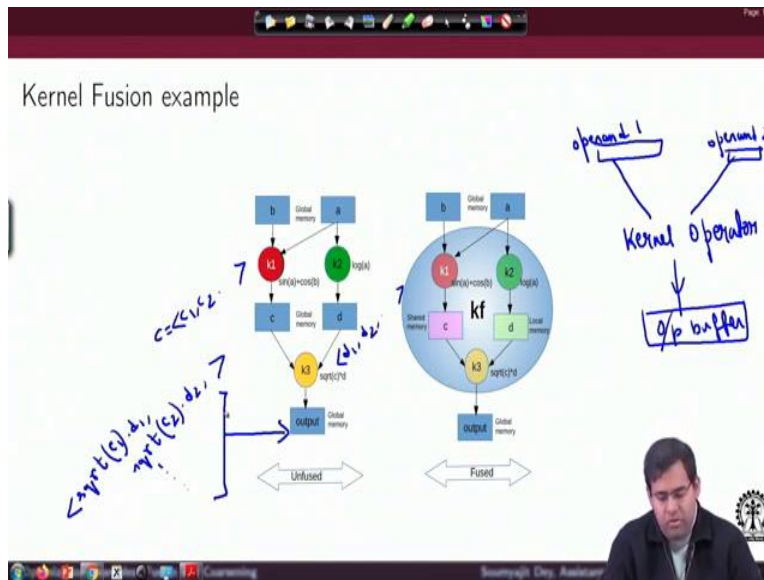
Beyond which that advantage will also go I mean that advantage will also be lost and the other thing is when you are fusing kernels there may be an overhead in case the kernels do not have an identical data I mean data access patterns. There may be overheads in terms of fusing the kernels. These are the few things which will also study. Also in general it may be difficult to search for optimal rearrangement of fusible kernels.

So essentially first thing is I have to identify which are the kernels which can be fused to our advantages. Then the next problem can be that what should be the if there are many kernels which I am deciding to fuse and the next problem is more related to scheduling of the kernels that can there be a suitable arrangement of the threads which use me the best possible fusion. Now that can also be a decision which may be quite complex in general.

Let us not get to that. And also in certain cases if I am fused in process of doing the fusion I am incorporating too many conditionals inside the fused kernel code and it becomes way complex that may also introduce divergence which was not the case earlier right. So we have to understand that essentially this is an optimization where we are trying to exploit the entire parallelising the entire compute and memory bandwidth available with the GPU with respect to doing the fusion.

But if we overdo it, it can actually create some issues and speed of gains may not happen. So one has to decide on doing the kernel fusion based on his architectural knowledge and the usability of the GPU parallelism.

(Refer Slide Time: 13:15)



So what we will do is we will provide some examples and we will classify this idea of kernel fusion into several formal classes and figure out in which cases which kind of fusion is advantageous. In this figure we are just providing a basic simple example of kernel fusion. So we are just trying to show that in the global memory we have some data segments labelled with b and a .

So let us consider that it is a big data chunk sitting in b and also similarly in a . And we are considering 2 kernels k_1 and k_2 . So this is more like a task graph right. So k_1 is a kernel task graph or I would say it is a DAG right representing the partial order relationships between the operators and the operand. The kernels are the operators. So the k_1 and k_2 are the kernels which are the operators. They are operating on the data operands.

Like in this case k_1 is a kernel an operator which is going to perform a $\sin a + \cos b$ transformation on data inputs from b and a right. So these are the buffers containing that data. Sequential data of b and sequential set of data essentially a big a large buffer or containing the data in a . k_1 will launch multiple threads in parallel and for each data item in b and a it will simply compute $\sin a + \cos b$ right.

And this sequence of values k_1 is going to write in an output way for c right. So this entire dependency is highlighted here in the form of our task graph. So you have I would let us just

define the task graph here you have operand 1, you have operand 2, you have a kernel operation and here you have the output buffer. These are the input buffers right. So follow I have this k1 which is following these kinds of a structure.

And then similarly I have k2 which is working on a and providing me as an output buffer d containing the sequence of log a right. All the components of a and in that way in the global memory once k1 and k2 have executed we are saying that we will have 2 output buffers c and d filled with the outputs of k1 and k2. And then I want to execute k3 which is essentially going to taking the contents of c and d and is going to perform a further complex operation is going to do a square root of c and multiply by d and output that in the global memory.

So c let us say c is containing the values like that and this is containing the values like that. So here so this is what you get in output buffer sequence of this kind of values after k3 executes. So this is like a task graph. We are trying to show that we have kernels k1 and k2 processing input data buffers b and a and producing the output in the global memory output buffer right. This is the situation when we are doing an unfused execution.

What happens when I fuse these kernels? So essentially we can replace these 3 independent kernels k1, k2 and k3 and create 1 kernel called kf. So then what would happen is? Kf would be a single kernel whose inputs would be the global memory buffers b and a. So it will copy data from b and a, it would compute $\sin a + \cos b$ and $\log a$. And it would have the $\log a$ in the local memory and it is just 1 possible implementation.

It would be storing the $\sin a + \cos b$ data elements in the shared memory. This again I am just saying this is an example and then it will perform all the operations of k3 on these data points and then do the global right okay.

(Refer Slide Time: 18:18)

```
Code snippet for Kernel 1

__global__ void
process_kernel1(float *a, float *b, float *c, int datasize) {
int blockNum=blockIdx.z*(gridDim.x*gridDim.y)+blockIdx.y*gridDim.x+blockIdx.x;
int threadNum=threadIdx.z*(blockDim.x*blockDim.y)+threadIdx.y*blockDim.x+
threadIdx.x;
int i=blockNum*(blockDim.x*blockDim.y*blockDim.z)+threadNum;
if (i<datasize)
    c[i]=sin(a[i])+cos(b[i]);
}
```

Optimization Examples: Fusion and Coarsening

Soumyajit Dey, Assistant

So this is the code snippet I have for kernel 1. As you can see that you first find out what is the total number of blocks by this statement, total number of threads by this statement then you use this total number of blocks and total number of threads to create i which would be the total I mean exactly the total number of threads that have been launched right. So I hope this is clear. So I am finding out what is the block number right sorry.

Let me just repeat this part. So block number so essentially what you are doing is we are multiplying this block Id z with grid dimension on x and y right. And then you are doing an addition with block Id x dot y times the grid dimension in x direction and then you are adding block Id x. So in an effect all you are doing is you are considering for part thread. What is the corresponding block right?

I mean if I just linearize the total set of blocks then what is the block and which it belongs? And then I am finding out what is the thread number for it inside the corresponding block and then if you just execute this kind of a statement you figured out what is the global idea of the thread? Essentially the relative position of the thread across the entire input data space.

And of course we require that considering that I have these 2 input buffers a and b which are 1 dimensional arrays containing the data ai's and bi's. I mean of course they can be in other I mean in other setting fundamentally they would all be linear and that is their primary reason why you

would like to know the global thread Ids. So once you identify the global thread Id. You use that global thread Id to have a linearized access to a.

You just access a_i , you just access b_i . I mean when is that a valid access? As long as i is less than a data size which is a variable that tells you up to what position in both the buffers a and b . You are supposed to contain I mean they are supposed to contain a valid data right. So you compute the global thread Id, you use the global thread Id to compute the linearized position or offset inside the array similarly the location from b right.

And then you are simply doing $a \sin a_i + \cos b_i$ to figure out what should be c_i right from the with respect to the input buffers.

(Refer Slide Time: 21:03)

```
Code snippet for Kernel 2

__global__ void
process_kernel2(float *a, float *d, int datasize) {

    int blockNum=...
    int threadNum=...
    int i=...

    if (i<datasize)
        d[i]=log(a[i]);
}
```

Optimization Examples: Fusion and Coarsening
Soumyajit Dey, Assistant

And similarly if I can just do it for kernel 2. I just can find our block number, thread number and then again compute the global Id. I will just see that whether it is going to access something valid inside the data size and then I will just consider the corresponding position in the input buffer a for using a linearized access and stored the data into the output buffer d_i right. So I do I provide a code snippet for kernel 1 a code snippet for kernel 2.


(Refer Slide Time: 21:34)

Code snippet for Kernel 3

```
__global__ void
process_kernel3(float *c, float *d, float *output, int datasize){

    int blockDim=...
    int threadIdx=...
    int i=...

    if (i<datasize)
        output[i]=sqrt(c[i])*d[i];
}
```



Optimization Examples: Fusion and Coarsening Soumyajit Dey, Assistant


And then I also have a code snippet for kernel 3 where I am using i as the global thread Id and using that I am doing this computation of sqrt ci multiplied by di and producing the result to output i.

(Refer Slide Time: 21:48)

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



Optimization Examples: Fusion and Coarsening Soumyajit Dey, Assistant

So of course it is very simple to fuse all these sequence of instructions to a fused kernel. All we need to do is we will have a similar implementation compute block number thread number compute the global thread Id. If it is less than the data size you just use it to compute c, you just use it to compute d and then use c and d and apply the last operation to compute the output i value.

So this is kind of an example of fusion of kernels and as you can see that what is the advantage out of it? Now once we understand what how a fused code should look like you can see that is just a sequence of the kernel operations that have been performed. Now what is the control I have here? I can control this sequence right. So this is important. Let us observe what is the sequence we are doing? So we are first computing c then we are computing d and then we are computing output right.

Now when we look at the DAG it is not serializing k1 or k2. It is just saying that okay you can you have to execute k1 you have to execute k2. Once both finished then you have to execute k3. So both are valid execution orders here. When I am fusing the kernels I can do. Since if you look at the dependency structure of k1, k2 and k3 it is something like this which means if I just serialize I can have k1s operations followed by k2 followed by k3 or k2s operations considering that I have serialize them in the kernel code and this right.

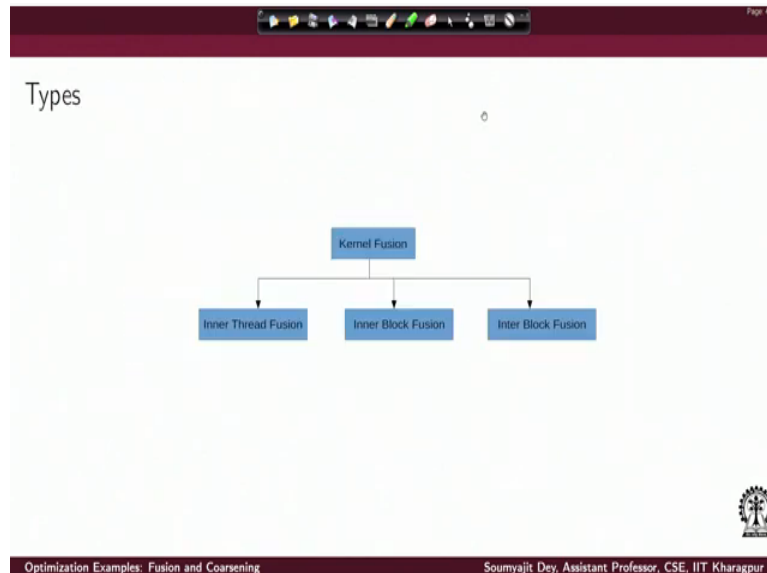
So when you look at the fused kernel I believe the log is in k2. So what we are doing is here? For each thread I am doing k1s operations followed by k2 followed by k3s operations. This is also a valid order. So if we just execute like this the original kernel 3 arrangement of 3 kernels essentially I leave it to the system to figure out how to execute the kernels? What should be the sequence followed by each thread and all that.

But when I am fusing that sequence is set by the programmer right. That is one possible observation here. That observation is let us identify what is the advantage in this case out of fusion? Well in this case it is a big advantage because each of the kernels are not doing that bigger computation but there is lot of global memory reads and writes right. So k1 and k2 are reading from b and a. Then they are writing back to c and d and then k3 is reading from global memory again and writing back to the global memory right.

If I fuse k1, k2 and k3 then what happens is? I have a global memory read from b and a and I have a global memory write which is at the final output. In between I just will have I will just have shared memory and local memory writes and reads. So that would significantly increase my execution time sorry significantly decrease my execution time and provide a significant speed up

here just because I have avoided lot of unnecessary global memory transactions. So in this case the kernel fusion operation was found to be advantageous.

(Refer Slide Time: 25:26)



Now while that was simple example and earlier we have discussed that it there are certain advantages of the fusion operation. There are also certain disadvantage of the fusion operation. The first thing we will do at this point is we will classify the different types of fusions we will formalize this classification different types of fusions possible among GPU kernels and then we will speak about what are the different advantages and what are the advantages and disadvantages of each type of fusion.

So kernel fusion primarily can be classified into 3 types. One is inner thread fusion, the other is inner block fusion, and the third one is inter block fusion. So maybe we will end this lecture with this introduction of this classification and in the next lecture we will go into the detail of each of these type of fusion optimizations. Thank you for your attention.