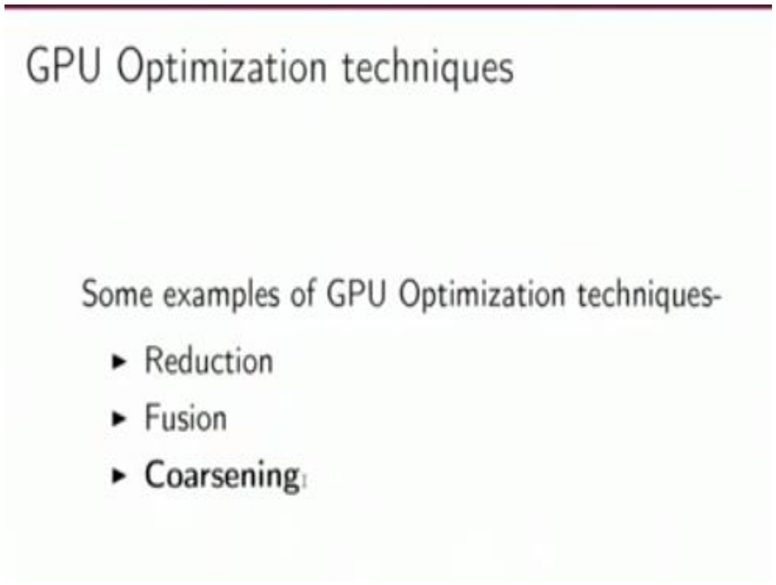
GPU Architectures and Programming

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

Lecture 38
Kernel Fusion, Thread and Block Coarsening (Contd.)

Hi, welcome back to the lectures on GPU Architectures and Programming. So in the previous lectures, we have been discussing certain GPU optimizations.

(Refer Slide Time: 00:32)



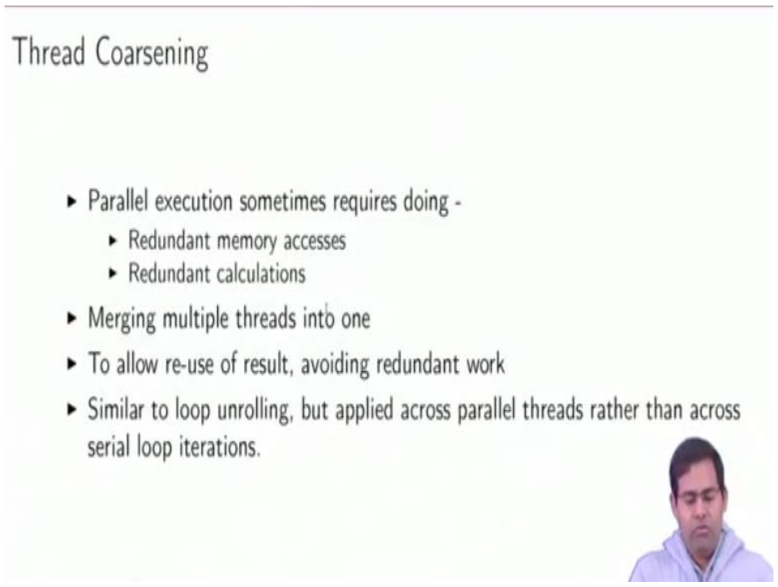
GPU Optimization techniques

Some examples of GPU Optimization techniques-

- Reduction
- Fusion
- Coarsening

GPU based parallel program optimizations, like reduction, like fusion, and we will now start with our last discussion on program optimizations in GPU more specifically CUDA based program optimizations and this would be the idea of thread coarsening.

(Refer Slide Time: 00:48)



Thread Coarsening

- Parallel execution sometimes requires doing -
  - Redundant memory accesses
  - Redundant calculations
- Merging multiple threads into one
- To allow re-use of result, avoiding redundant work
- Similar to loop unrolling, but applied across parallel threads rather than across serial loop iterations.

So speaking in layman's terms, so when we are doing parallel execution of threads, they sometime require to do redundant memory accesses and we also require to do redundant calculations and also we often launch multiple threads and the threads are pretty much light weight and launching so many threads in such cases do not make sense, because launching a thread also means creating a threads context in the hardware and keeping track of that thread.

And if for that thread, I do not really have too much of activity, then simply the launch of the thread does not make sense. So the idea of thread coarsening is that you do some kind of load balancing, that means you launch significant number of threads, but also make every thread bear some cost. That means make every thread do some meaningful activity and also the threads should make re-use of results and avoid redundant work.

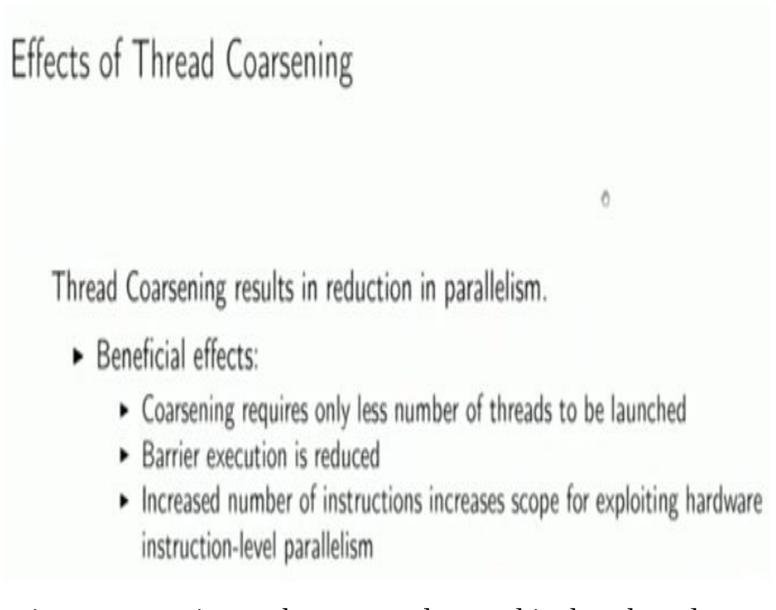
What we mean here is, when we are doing a GPU based computation, every thread has got its own part thread private local memory. Every thread gets its part of share in the register file. So these are things which we have discussed earlier also, like let us say we are using an video GPU, so while you can actually set that statically that what should be, whether you are going to allow register speeding or such things to happen or not.

By default, they are prevented. So when you are going to launch your kernel, the system is automatically assigning a set of registers for part thread activity and that would also mean when the thread is doing its computation, the results are available in the registers, which are fixed for that thread, but those are also the registers, which are concurrently running thread cannot see. Maybe it is doing some activity, for which it is not available.

But it may be the case that this adjoining threads, they have some locality in terms of computation. They do some activity, which could have been re-used. So if I make one thread to do activity in this case, for both threads together, then maybe some computations which the thread 1 has made can be used again by thread 1, if it is doing thread 2's activity without original thread 2 duplicating the same activity inside its own set of registers or the set of register that may allow that way.

We will make these things clear that how this re-use of results can happen and all that and how I can make threads have a redundant work. So the concept is quite similar to loop unrolling, but here instead of applying it on sequential execution of a thread and avoiding multiple executions of the loop, we are actually applying it across parallel threads rather than on the serial loop iterations of a single thread.

(Refer Slide Time: 04:18)



Effects of Thread Coarsening

Thread Coarsening results in reduction in parallelism.

- Beneficial effects:
  - Coarsening requires only less number of threads to be launched
  - Barrier execution is reduced
  - Increased number of instructions increases scope for exploiting hardware instruction-level parallelism

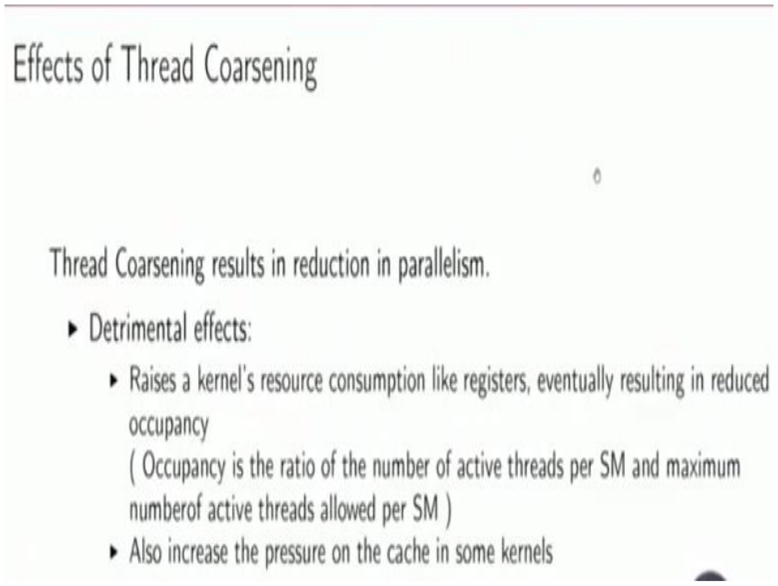
Now, of course, one important point we have to understand is that thread coarsening would result in a reduction in parallelism, but there are good effects of that also. First of all, why does it result in reduction in parallelism, because if I let one thread do the job of multiple threads, then my thread block size can decrease or I can keep the thread block size same, but the oral activity can be done by a lesser number of blocks.

So if the total number of threads I launch is less than the number of threads that the GPU can execute concurrently, then there would be a reduction also in parallelism, but otherwise, what can happen is, if the original threads were very light weight, now each of the threads have got some significant activity. So anyway, if there is available parallelism, then the kernels would be launched and all the parallel execution units will be engaged.

So the beneficial effects, if we can summarize is that, the coarsening would require only less number of threads to be launched. So due to that, as we said that lot of redundant execution can be minimized. For example, if there are barriers for thread, now since I have less number of threads in total, the number of barrier execution is reduced, so that would also mean an overall gain in the end-to-end execution time of the kernel.

Since now I have more number of instructions to execute per thread, but that would also mean, that now I can exploit hardware level instruction level parallelizing, which maybe earlier for each of the threads being very light weight, there may not be too much parallelizing inside the threads.

(Refer Slide Time: 06:06)



Effects of Thread Coarsening

Thread Coarsening results in reduction in parallelism.

- Detrimental effects:
  - Raises a kernel's resource consumption like registers, eventually resulting in reduced occupancy
    (Occupancy is the ratio of the number of active threads per SM and maximum numberof active threads allowed per SM)
  - Also increase the pressure on the cache in some kernels

Now, this may also at certain points be detrimental. Of course, here we are just giving some (()) (06:17) definitions like we have said that okay, since the threads have own local share of register file, there may be redundant computation going on, if threads are light weight. So if we merge activities of threads to a single, then whatever was the computation by thread 1 and earlier it was done, maybe sometimes the thread 2 was unable to re-use it.

But since now one thread is doing the job of two parallel threads, it can make a better use of the register file. Those were the arguments that we are trying to make in terms of thread coarsening advantages. At the same time, if we try too much of that, it can also be disadvantageous. Why, because suppose I am now coarsening the threads making each thread do lot of activity, so then observe, this is the most important point here.

That when the program is getting compiled, the compiler is figuring out the part thread activity and it is allocating a part of the runtime system. The runtime system is going to allocate significant amount of part of the register file, now to part thread. So that is a finite resource. Register file, in total it is a finite resource, shared memory in totality is a finite resource. So when the runtime system is going to allocate it, it is figuring out that, okay, now the part thread register file usage has increased or the part block share memory usage has increased.

So since the resource increase part thread and part blocking effect has increased, that total number of thread that the system or more specifically parts streaming multi processor can execute will be reduced. Now this is a notion that is known as occupancy. So occupancy is the ratio of number of active threads per SM and maximum number of active threads allowed per SM. So let us say we have the number of maximum threads that are allowed per SM is 2048.

Let us say that is the maximum thread context that I can have and then it is not required that the occupancy that the SM will support in the runtime is same as that maximum limit. Why, because as you have figured out, that whenever I am launching the threads, the threads will be allocated local resources in terms of registered files. The blocks will be allocated local resources in terms of shared memory.

If I coarsen the threads, there will be point that these values will be significantly high, that is the part thread resource requirement is high. The hardware in that case will not really launch that number of blocks into the SM as is allowed for the SM, simply because that many number of thread blocks cannot execute parallelly, because the lesser number or subset of them is good enough to completely engage the local shared memory and the register file resources.

So that would eventually result in a reduced occupancy value. This is a finer point, but I hope it is clear. We need to understand the SM can run as many threads as there are SPs and Sf is there and as many thread context that SM can remember, but at the same time, when it is running the threads, the threads will be occupying its own share of local resources. So once I increase that share of local resources, the hardware will not really run that many active threads, which it can and this ratio is the occupancy factor.

If I coarsen the threads too much, the occupancy factor is going to decrease. So that is the effect that the kernels increased resource consumption will create in terms of occupancy and that may be detrimental with respect to the overall kernels execution and the second thing is that if coarsening the threads or increasing the part thread activity, as we have discussed earlier, that if the part thread activity increase, I may be removing some redundancy in terms of thread computation.

Because earlier with two threads, we were not able to share their computations in terms of registers, I mean computed values to the registers, but now that can. At the same time, I am increasing cache activity. This would increase the cache pressure on the kernel and that is also a resource. Now at this point, I would like to also talk about something, like with respect to cache that is present in the kernel.

Earlier we have discussed that for in the GPUs, we have L1 and L2 cache. L2 is unified and L1 is present inside the SM. As a shared memory plus L1 setting, which can be configured. Now while that is too up to the Kepler case of architecture from Maxwell case of architecture, there are instances of GPUs with separate share on L1 caches.

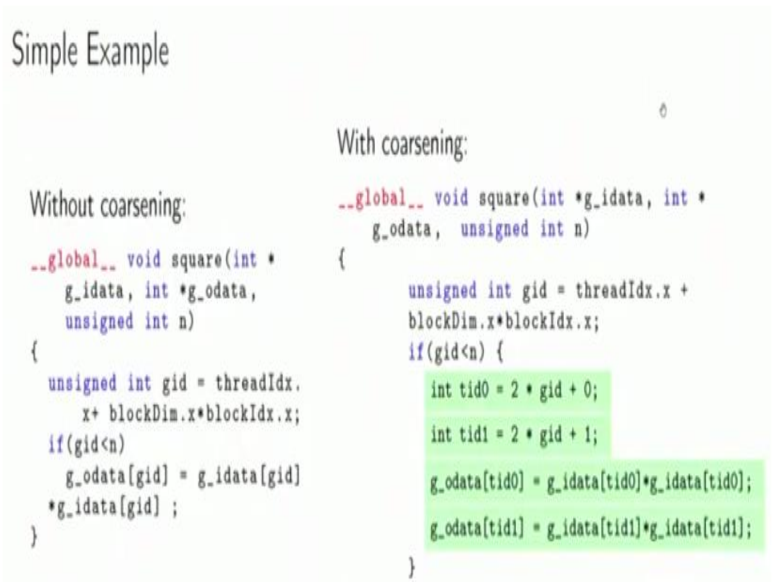
So when the threads are going to do memory reads and writes, they are also consuming the resources that is cache and with part thread memory reads and writes increasing, the cache pressure also increases and that also plays a role in deciding what is the total speed up gain by thread coarsening and with the too much of coarsening, there may be increased cache pressure, there may be increased latency of the instruction execution, due to this cache pressure and that would be a detrimental effect.

It may finally lead to a loss of speed up. So the point we are trying to make here is for every kernel, it is not the cache that you keep on coarsening the threads, reducing the number of threads as much as you want and because if I think it on nice way, if I give more activity per thread, well then I would have less number of, I am doing more amount of work per thread, so I am really reducing the total grid size for the kernel, but that may not always be the case.

As we are finding, that there is a sweet spot with respect to what is a good coarsening factor. The optimum coarsening factor should be such that it will make maximal use of the GPU parallelism. We have to understand that the parallelism is in length and breadth, because I have a set of resources. I want to consume these resources that is the execution resources and the memory resources as much as possible in parallel, but also in length.

Because I want to use them as much as possible in parallel for as much time as possible. So there is this sweet spot in parallelism. So if we go too much with respect to coarsening, then finally it will lead to sequential execution and we will lose parallelism. So this sweet spot is something that as a programmer, you have to understand by looking at the activities and the algorithm.

(Refer Slide Time: 13:25)



Simple Example

Without coarsening:

```
__global__ void square(int *
    g_idata, int *g_odata,
    unsigned int n)
{
  unsigned int gid = threadIdx.
      x+ blockDim.x*blockIdx.x;
  if(gid<n)
    g_odata[gid] = g_idata[gid]
  *g_idata[gid] ;
}
```

With coarsening:

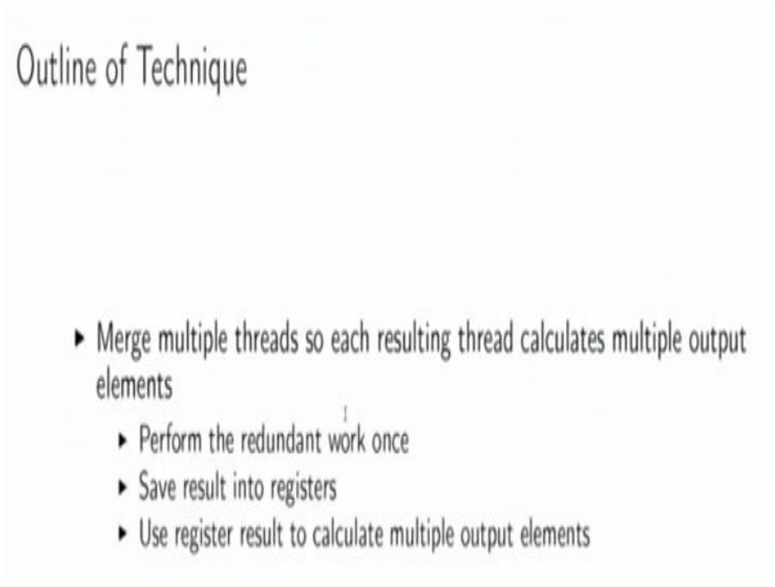
```
__global__ void square(int *g_idata, int *
    g_odata,  unsigned int n)
{
        unsigned int gid = threadIdx.x +
        blockDim.x*blockIdx.x;
        if(gid<n) {
          int tid0 = 2 * gid + 0;
          int tid1 = 2 * gid + 1;
          g_odata[tid0] = g_idata[tid0]*g_idata[tid0];
          g_odata[tid1] = g_idata[tid1]*g_idata[tid1];
        }
```

So let us start with some simple examples of coarsening and we will get back to this issue of speed up that one can achieve with coarsening later one. First, let us start without coarsening code. So on the left hand side, we have the code which is without coarsening. As you can see, it is a simple program, which is just revealing what is the global thread ID and then for the content in the memory location for the thread ID, we are just executing a square operation.

All we do with coarsened version of the kernel is that make every thread do that operation for two consecutive locations. So the initial case is, you launch thread and they all perform squaring of the data and then in the modified version, this is the without coarsening case. So this is how, with half the threads, we are doing the activity and as you can see that first we find the global ID and we use this global ID, we just multiply it to create the offset at which it will work.

Because we know that now for all the previous thread IDs, the work from two data points. So we have to then bring in this multiplication factor by 2 and the next element, multiply by 2 and plus 1. So these are two consecutive locations. So for anywhere in between, you take the thread ID, compute the global ID and then go to the next consecutive location. So these are the two consecutive locations, for which we are going to do the work.

(Refer Slide Time: 16:01)



Outline of Technique

- Merge multiple threads so each resulting thread calculates multiple output elements
  - Perform the redundant work once
  - Save result into registers
  - Use register result to calculate multiple output elements

So if we just make a small outline of that technique, what we are doing is, we are merging multiple threads, so that each resulting thread calculates multiple output elements. So they perform the redundant work once. So why shall we really say that? Essentially you can see that each thread earlier, every thread would have some launch over it and they will do this computation.

Now we have reduced the number of thread context by 2, so that removes the product and we have given significant activity for each of the threads. For example, we can just say that when this has been fetched, the other data is also being used, is being computed in another register by the same thread. So the work of loading the data is being reduced and the results are getting saved into the register and use the register results to calculate multiple output elements.

While that is not exemplified here, but it is easy to understand that when we have the multiple threads merged, then suppose each thread was having some initial activity, now that I can do the initial activity together. So let us say the initial each of the threads had some common multiplication or some common constant evaluation to do. Now that is not specific to a thread ID or corresponding data space. If that is the case, then that is the definition of the redundant work.

Now that it would be done in a smaller number of times. That is point 1. Second point is intermediate results can be saved to the registers and they can be used for doing some computation, which were originally done by the other threads and then the register results can be used to calculate multiple output elements, maybe let us consider the situation, that earlier there were some part thread activity and finally this part thread activities output, which are computed by which of the threads, were going to go through some transformation.

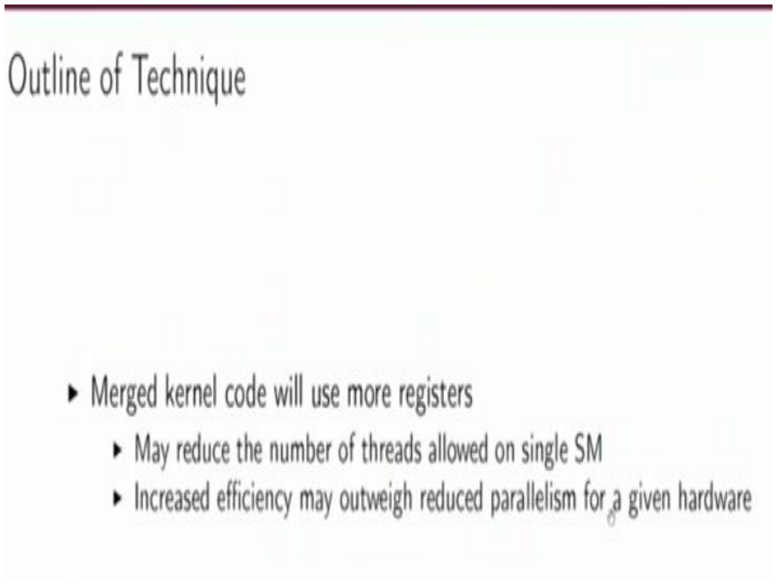
Now since two adjoining elements are already computed by a single thread, if that transformation was associative, then some part of it can be done right here. I hope that I am making sense here. Suppose in the original kernel, whatever the outputs they are going to be computed, so let us say this is the final output of all the threads here, they are going to go through some transformation function f.

So let us say these values are a1, a2, an, so I am going to apply if on a1 and then a2 and then like that up to an. Now that many operations are required, but now inside this function only, I can compute if a1, a2 and similarly when these threads get done, then this job is some of it is already done here. So final label, I have to work with half of the values like considering this (()) (19:28) operation, this is fine here.

So just because each of the threads are now working on more data points, I am again making a lot of savings here with respect to achieving some more parallelism right at the thread level, before the synchronization point. So we can understand that fundamentally what is happening, since I am increasing the part thread activity. I can reduce the redundant work, which each thread was doing as a common activity.

I can make use of the intermittent more number of registers to store intermediate values, which can be used and which can actually help me to calculate a final output value, which can actually help me to reduce a final step, which the threads are suppose to do, once each of them compute their output value and then synchronize and then perform. Now some of the value, if there is an (()) (20:26) operations, some of that can be performed right now when the threads are executing on different data points.

(Refer Slide Time: 20:31)



Outline of Technique

- Merged kernel code will use more registers
  - May reduce the number of threads allowed on single SM
  - Increased efficiency may outweigh reduced parallelism for a given hardware

So the point we are trying to make here is also that the marched kernel code will use more registers. Now what is the issue here, that this may reduce the number of threads allowed on single SM, like we were speaking earlier. This may reduce the occupancy. Now increased efficiency that we are getting may outwardly reduce parallelizing the given hardware. So this is an important point, reducing occupancy is not the bad thing.

So I may have reduced number of threads on the single SM, but since I am also having fine grain parallelizing inside each of the threads, I am having a reduction in the local computations of the threads due to lack of redundant activity. I am gaining by doing some parallel computation, on the outputs computed by the original threads now, due to the coarsened activity. Initially, the reduction in occupancy of the SM may not hurt, because it may be too more efficient.

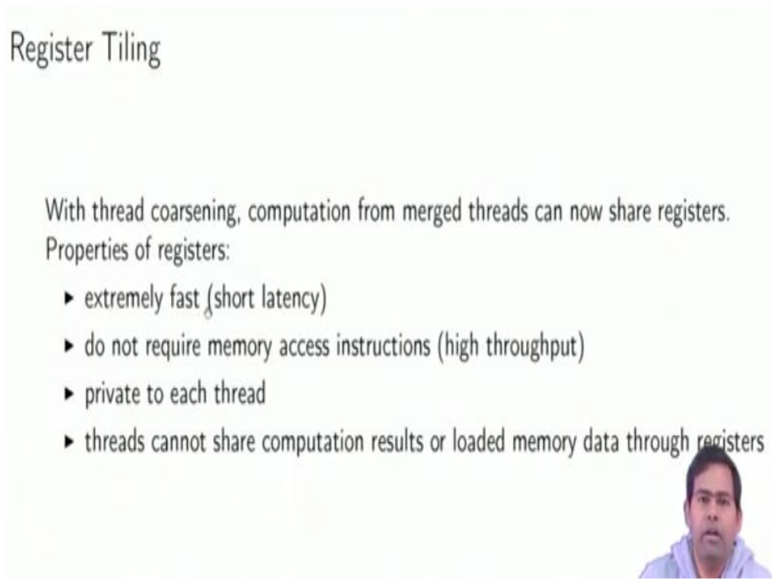
Again I will repeat this part. Each thread is consuming more local resources. Due to that, up to some point, it does not hurt, because although they are consuming more resources, the SMs are full, where they are going to operate with the maximum number of threads they are allowed. After that point, if I coarsen more, then the SMs will execute a lesser number of threads that they are allowed to execute, but still this reduced parallelism may not be bad enough.

Because inspite of the reduced parallelism, inside each thread I am now exploiting some instruction level parallelism and I am doing some redundant work, less amount of time, but after some point of time, it will start hurting the global execution view. So up to some coarsening factor, I may have increased efficiency even with reduced occupancy, that beyond a coarsening factor, the occupancy will go so much down that the reduced parallelism disadvantage will outweigh the effect of increased efficiency due to thread coarsening.

So again I will repeat up to some point, the efficiency factor in terms of coarsening that we gain in terms of efficient execution outweighs the reduced parallelism, but after that point it starts hurting. If we lost too much parallelism, then it will start hurting. So again to summarize, that up to some point, if you coarsen the threads, you may not lose occupancy, because still the SM is able to execute threads up to its maximal limit.

After some point, it will have reduced occupancy, reduced parallelism, but still the instruction level parallelism and increased efficiency will outweigh the reduction in parallelism, but again after some point the occupancy reduction is so high, that coarsening will be detrimental.

(Refer Slide Time: 23:37)



Register Tiling

With thread coarsening, computation from merged threads can now share registers.
Properties of registers:

- extremely fast (short latency)
- do not require memory access instructions (high throughput)
- private to each thread
- threads cannot share computation results or loaded memory data through registers

Something important here, that with thread coarsening, the computation of the merged threads starts sharing registers. This is something we have discussed earlier also, multiple times. So why is this a good thing. First of all, we all know that register file is extremely fast. So if I increase part thread activity without decreasing the occupancy significantly, then still it is a good thing, because if the thread brings more data and loads into its part thread available register file.

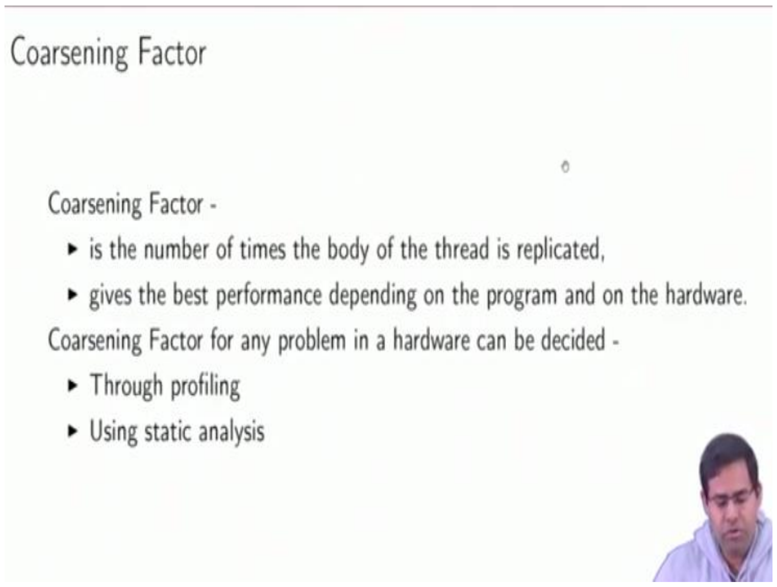
Then does not the threads while doing their local computation, they reduce execution of memory access instructions and in that way the gain in high throughput and this is a good thing, because earlier when the threads were not coarsened, each of the register file portions that were available to each of the threads at the lower ground level, they are private and suppose the thread 1 gets S1 of registers available to it, thread 2 gets S2 of registers available to it, so they cannot share the computation results, because they are private to their threads.

But now, when I coarsen the threads, the set of available registers may be S1 + S2. So the threads get more registers to work with and now they can share the intermediate results. They can decrease the number of memory access instructions in between. Why? Suppose earlier when some register was storing some local result and then for part thread activity, maybe the register size was not good enough, that is register spilling, by default it would be deactivated in the GPU.

So then, there will be some access to the local memory, sorry access to the global memory. Memory access instructions will be executed, but now if I have more number of registers available, it may happen, then due to this sharing of context registers, I have less number of accesses required. I can avoid the register spilling a bit and since the threads are able to share the computation results, they will make a more efficient utilization of the larger set of registers that would be available to them.

That will actually lead to the efficient computation, that we have been discussing in the previous slide.

(Refer Slide Time: 26:33)



Coarsening Factor

Coarsening Factor -

- is the number of times the body of the thread is replicated,
- gives the best performance depending on the program and on the hardware.

Coarsening Factor for any problem in a hardware can be decided -

- Through profiling
- Using static analysis

So the other thing, when we coarsen the threads, there has to be a choice of coarsening factor. So if we look into the example we give in terms of the program, so this is simply a coarsening factor of two. We are employing a coarsening factor of two. You are increasing the part thread activity by two. What is a good coarsening factor is also an important issue. The definition of coarsening factor is it is the number of times the body of the thread is replicated.

In that example, the replication is twice. Which coarsening factor gives the better performance, the best performance will precisely depends on the program and the hardware. Coarsening factor for any problem in the hardware can be decided in two ways. I mean, we can get an estimate of the coarsening factor by performing a static analysis of the program and the dynamic way would be to profile the program, you run the program with different possible coarsening factors.

Also for each run, you try different possible inputs of the program and you get the execution statistics and figure out, okay for what is the good coarsening deficient for this kernel. So just to summarize here, let us understand that for each state, which we increase the part thread activity, part thread memory requirement, thread starts sharing registers. In that case, they can store intermediate results inside the available registers.

This may reduce memory access for the global memory, because if it can make a good use of the larger set of registers, which are now available to it for sharing purpose and that would actually ensure shorter latency for the memory accesses, because you have more data in the registers and you execute, if you do not require memory access instructions, then you can attain high throughput in terms of the overall kernels execution.

So this is the advantage. We are not going too deep here. In the next lecture, we will take some more examples, when we really take, we will see that how the coarsening factor affects the kernel's execution. With this, we will end this lecture. Thank you.