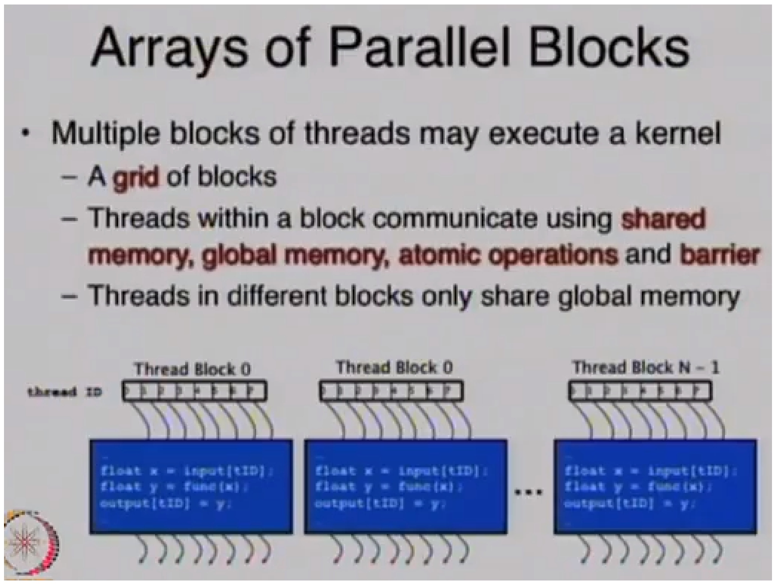
Parallel Computing

Prof. Subodh Kumar
Department of Computer Science & Engineering
Indian Institute of Technology – Delhi

Module No # 05
Lecture No # 22
CUDA (Contd.)

(Refer Slide Time: 00:35)



Arrays of Parallel Blocks

- Multiple blocks of threads may execute a kernel
  - A grid of blocks
  - Threads within a block communicate using shared memory, global memory, atomic operations and barrier
  - Threads in different blocks only share global memory

Thread Block 0
Thread Block 0
Thread Block N - 1
thread ID
...

```
float x = input[tID];
float y = func(x);
output[tID] = y;
```

Continue that discussion so we have a kernel it is organized as grid of blocks many blocks and each blocks as some certain number of threads okay. The entire block meaning every thread of the block whatever the number may be we limit on the recent architecture is 1024 limit on once on probably using its file later okay. So up to 5, 12 threads of a block will all be like sometimes if a block does not use all the resources on the SM.

Multiple such blocks can be live okay so now your grid had 1000 blocks and you have let us say 2 SM's each SM can fit let us say 16 blocks. So among the 1000 16 blocks will be assign to the first SM 16 blocks will be assign to the second SM okay and until a block is done other blocks will wait now at a given time in one of this SM's there are 16 live blocks meaning that everybody has subset of the register allocated to it and everybody is able to the read things from the memory shared memory global memory wherever and each clock scheduler will ask I have the 16 blocks running which block instructions are ready and that is somewhat of an information.

Because actually a block need not be only 16 threads right you can only execute 16 at a time so there is there are some details that are not critically important are time being let us say that a blog is schedule meaning that when a block is scheduled whatever the number of instruction says will run one after the other okay that is not the exactly the wait is but let for instance think that way it not the going to change anything in the general understanding.

So suppose my block as 32 thread so I have got 16 such blocks every block as 32 threads and I will take the first block and run all it is 32 threads on the SIMD the 16 processors that I have okay. Once I have taken the 16 processors and run one instruction on that group of 32. I will have ask which next block whose instruction can I run okay. So I have got 16 blocks to choose from it can be the instruction that I ran the instructions of it will actually not be the same again for some detailed reasons but any of the blocks which has its operands in the register is ready to execute its next instruction okay.

And its next instruction means it is 32 thread blocks next instructions so when that block is ready you take its 32 threads run it on one of the SM's and next instruction has the same question. And so as far as the scheduler is concerned it is just a number ready instruction to run okay it does not care about which blocks they came from which blocks are ready with other things with just says here is the group of 32 there data has arrived that means some hardware that you track of data has arrived and their operands in the register and so on and hence so forth.

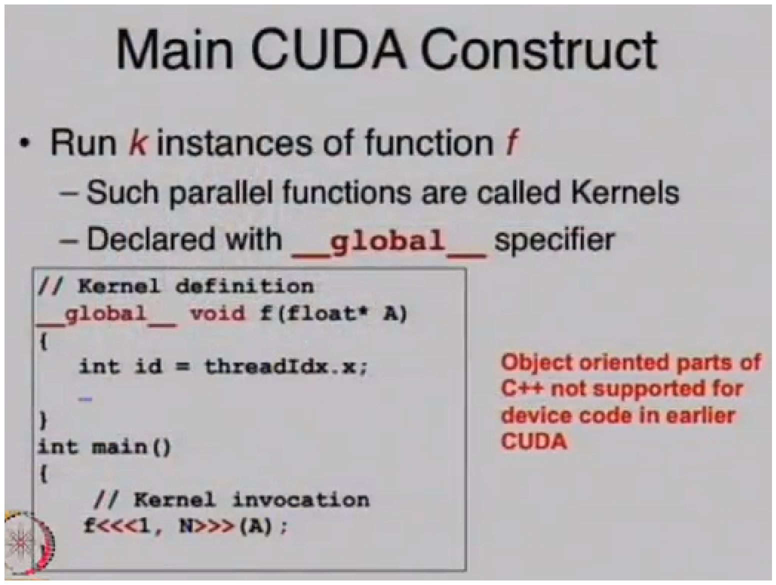
So that flag is one meaning its operands are there it will execute it next clock cycle it will find something else to execute and it is as if you are executing one big program okay because as far as the (()) (05:13) is concerned it is just every clock executing one instruction in the CPU that instruction typically all those instruction come from one thread. Here that instruction comes from one block okay one block of threads and then the next instruction comes from the block of threads and those instructions may be different depends on whose data arrived first whose getting data compared compassion and so on and hence so forth.

And it is not unusual it is quite usual that you can fit more than one block in one SM okay if your blocks are big for example 1000 thread blocks it is possible that only one blocks in SM at a time but then even that block as many groups of 32 and you can only execute only 32 at a time and so you will say the first 32 executed second 32 executed third 32 executed and so on and hence so forth okay.

And they need not be round robin order either even if it is one block you it is possible that one of the 32 threads data they can be reading from wherever they want to comes in early and it is ready for execution whereas in another group of 32 will take several clock cycles before the data arrives and so in the meantime 3 other groups of which by the way called warps have executed 3 at each and that guys data have not arrived it.

So it is only within in a work that there is SIMD steps not in the entire block however if you want to lock in entire lock steps then you do same threads right that you do a barrier you say that I want everybody to get to this instructions the barrier instruction and then they all start proceeding from there okay.

(Refer Slide Time: 07:58)



Main CUDA Construct

- Run *k* instances of function *f*
  - Such parallel functions are called Kernels
  - Declared with `__global__` specifier

```
// Kernel definition
__global__ void f(float* A)
{
   int id = threadIdx.x;
   …
}
int main()
{
    // Kernel invocation
   f<<<1, N>>>(A);
```

Object oriented parts of C++ not supported for device code in earlier CUDA

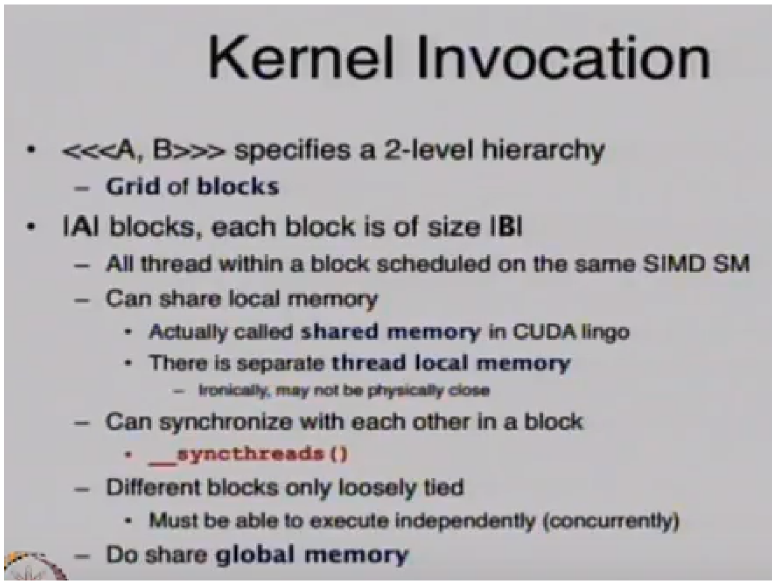
So here is how you would this is a very simple definition of a global function which is how you define a kernel is underscore global underscore void F it takes argument that is the flow okay. And inside it says give me my thread ID and the thread ID is off the tight called dim 3 dimension 3 you can if you have declared 1 DRA you will be only checking the thread ID DX pre-defined variable and you can access IDX dot X dot Y and dot Z.

If you have only one dimensional array then you only accessing contacts an from the collars for example in this case the main so male if the host function whereas F is a device function technically it is a global function it runs on a device but its address is available on the host okay because the host has to call it. So if you look at the main function it is main F open close brackets some information about how many instances of this to be run in the kernel and the parameter is A okay.

A is the parameter that is going to be available to all the threads which are going to participate in this class okay. 1N here says that we are executing 1 block 1 is the dimension of the grid and the block size is N threads and it is define somewhere one grid having you always have one grid right. How many blocks in the grid is the first thing in this is case there is only one block there is only one grid the kernel can be thought off as the grid this kernel is the associated with the one grid.

And how many blocks you want to run or independent units some more independent units and then size of the blocks which is more tightly coupled set of threads and its N threads in the block one block in the grid. Now all the object oriented parts of C++ are well supportive but on earlier version there was some bug and issues with virtual functions and instancing in a lot of steps. So you may want to just say with basic C style program you can take classes do not get to innovative.

(Refer Slide Time: 11:26)



Kernel Invocation

- <<<A, B>>> specifies a 2-level hierarchy
  - **Grid of blocks**
- |A| blocks, each block is of size |B|
  - All thread within a block scheduled on the same SIMD SM
  - Can share local memory
    - Actually called **shared memory** in CUDA lingo
    - There is separate **thread local memory**
      - Ironically, may not be physically close
  - Can synchronize with each other in a block
    - `__syncthreads()`
  - Different blocks only loosely tied
    - Must be able to execute independently (concurrently)
  - Do share **global memory**

Okay so the configuration is how you say how many times how many blocks times what is how many threads in each blocks no the global is not even visible after it has been compiled okay something that is processed by the high level NVCC compiler okay. S when I am saying that it is accessible when I said it is a data everything that function that compiles to is the set of instruction basically data as far as CPU is concerned.

Yes but not from your program right yeah there are we will come to that there are synchronize and A synchronize calls just like MPI if you recall and for synchronize call you have to wait for that function to return okay. It turns out that when you invoke a kernel function that synchronous call and so we have to wait for the kind of a time there is a way to stream this kernel and you can it into a Asynchronous calls we hopefully we will talk about that is not necessary to understand the basics of good you can read that one up later okay.

So among the most important changes if we see is this configuration three less than 3 greater than and you have got 2 variables constant separated by an among these each can be an integer just like this was in the previous example but when it is an integer that means it is a one deal array. If you want a higher dimensional grid or blocks configuration then you have to use one of those variable predefined dim 2 or dim 3 for 2D and 3D arrays okay.

What you can do is have different blocks of a grid with different size in terms of number of threads this basically a cross B type definition what you cannot do is in the middle the kernel say now create so many more threads so that there are some of the limitations as I was mentioned all thread which are within a block are scheduled I groups of 32 on the same SIMD they will have access to the same shared memory but two blocks.

Because you do not now because you do not know whether they are going to be scheduled on the same SM on different SM’s cannot share each other share memory cannot see each other shared memory and that is filtered or that becomes visible in the programming order is where you cannot allocate shared memory for multiple blocks. So from a thread point of view what is something that is totally owned by the thread registers but no other thread even within your block can see what is in your register.

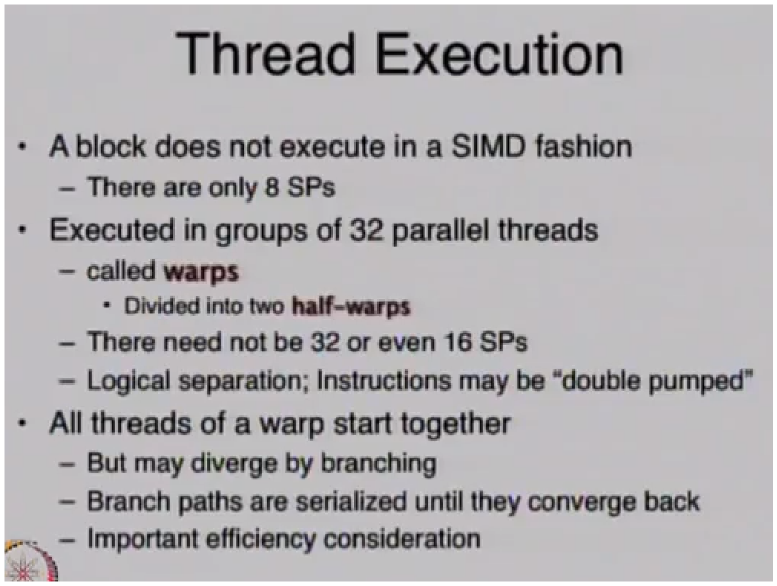
Or even knows about the register cannot address your register there is also non register space called thread local memory is like local variable of a thread which resides in local memory okay. So once understand it is not confusing but the names are not unfortunate is called thread local memory and lives in local memory but it is thread local in the sense that scope is local only that threads knows what is in there and that thread can address it okay.

Then there is shared memory beyond the thread and you can see other thread are writing of the block are writing into their shared memory and when you write into the shared memory of the thread can see it and then beyond that is global memory which is the new write something t the global memory any thread of any block can see it eventually and there is no guarantee of any consistency except that when you write something and somebody else write something either your data or somebody else data else will get written.

It is never that half of other data and half of somebody else data written so if you write 69 and somebody else write 7 never be that data in the memory is going to be one okay it is either 69 or 37 but there is no way of knowing unless you do some synchronization which is going to get written okay. There is no guarantee that different thread of different blocks will see anything in the same order okay.

You have to ensure that is going to that must happens there are guarantees that with in the block they are very clear guarantees within a war because they are executing in lock step fashion but overall there is no guarantees okay.

(Refer Slide Time: 18:31)



Thread Execution

- A block does not execute in a SIMD fashion
  - There are only 8 SPs
- Executed in groups of 32 parallel threads
  - called **warps**
    - Divided into two **half-warps**
  - There need not be 32 or even 16 SPs
  - Logical separation; Instructions may be "double pumped"
- All threads of a warp start together
  - But may diverge by branching
  - Branch paths are serialized until they converge back
  - Important efficiency consideration

So this is some more of that discussion that we are having and this says that there are 8 SIMD processors that is the previous generation and next current generation has 16 of them but in both generations the notion of the warps is 32 wide. So there are 32 threads that are going to execute in SIMD meaning that if some instruction of one of these 32 executes you can be sure that other 31 have executed that instruction alright.

This is now guarantee that must warps of even one block and there is also notion of half block which come from the fact that 15 actual processors right half warps is actually physically being executed together and the full warps is executing one after another precisely one after another right so you are why can we guarantee that if your half if your warps fraction 5 I have seen given thread of seen instructions 5 and is going to instruction 6 then every other thread of that warp has finished execution of warp instruction 5 right.

In spite of the fact only half warp execute in real SIMD fashion strictly right once you are executed 5 your 6 cannot begin until other half is not executed 5 then your 6 begins then your 6 beings then that guy 6 begins okay. So still in lock step fashions there is a notion of branching within a warps as you mentioned also meaning if you say my ID is even they something with the ID or do something else and half of them will be doing on something the other half will be doing the other.

There is significant overhead to this branching one is clearly that when one is live other so suppose you say if my ID is hard do something then the SP on which the thread is turning out to be even those blocks will be doing nothing. So those instructions those blocks ID have been wasted on those processors but there is also significant overhead in collecting the data together when you have this divergence of execution and at some point they are going to probably converge.

So you can diverge and execution unit have to keep track of which are the live ones that are going to execute now and which are the ones that are going to execute the else part there is one right. So that keeping track of divergence and the eventually convergence has significant overhead and as a result the new architecture have this notion of each instruction or rather each thread of each warp being separately enabled or disabled which is not exactly the same thing as convergence and divergence they all are basically not diverging at all.

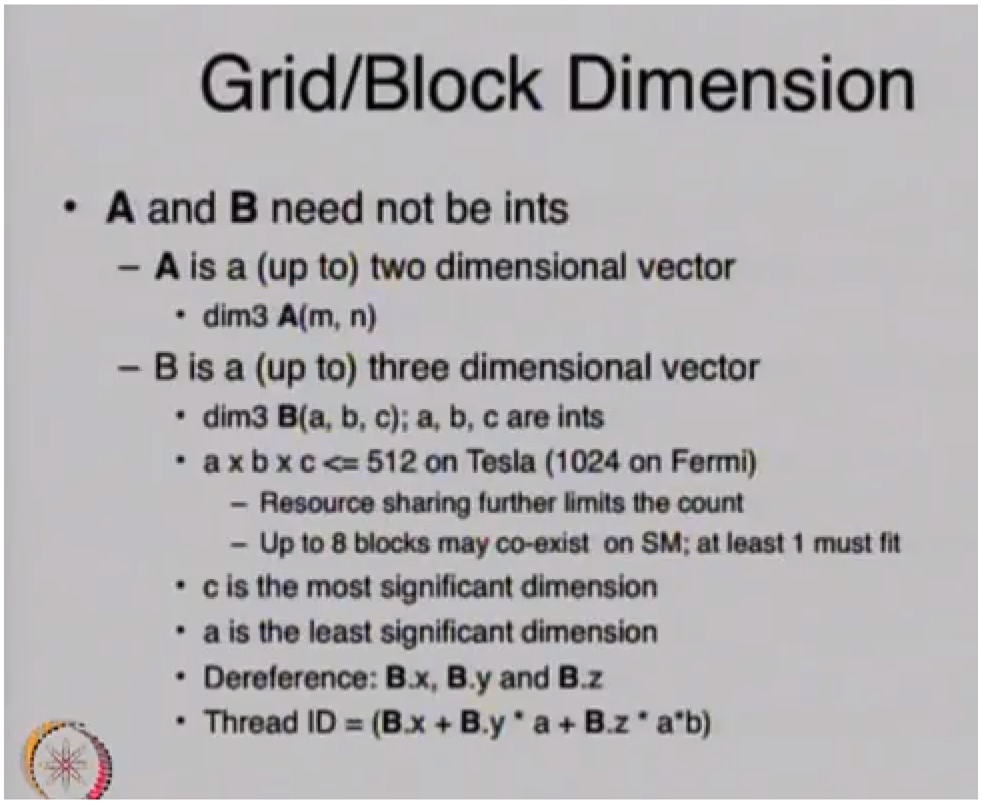
That every instruction is being executed for all of them but something of them happen to do nothing nobody is keeping track of if part has been executed now let us execute else part on the ones that were in active in the previous phase.

So in divergence the hardware is keeping track of I have executed the if part thread number 1, 7 and 9 where had the condition true others had false right it has to keep track of now I have to go to the false and execute them and that is the way instructions get scheduled but in this case you keep every instructions as a complete full instruction if there is no if else it is simply a mask that says which of the SP's are going to execute this instructions right.

So now the compiler this is typically done for simple loops where you say if X do something else do something right so it will generate two pieces of code and in once piece of code its going to get keep everything that is going to execute to true enable the other piece of code is going to get everything disabled okay. So the hardware is relived the extra overhead of keeping track of this okay condition check can happen at runtime right.

So that once you have that condition check that masks can be set and then there is instruction to get the mass and then you do the other things. So switching is needed the mask so that is also an instruction that is negate the masks so these instruction did not execute earlier or did not exist earlier now they do 16 processor executing 32 things.

(Refer Slide Time: 24:25)



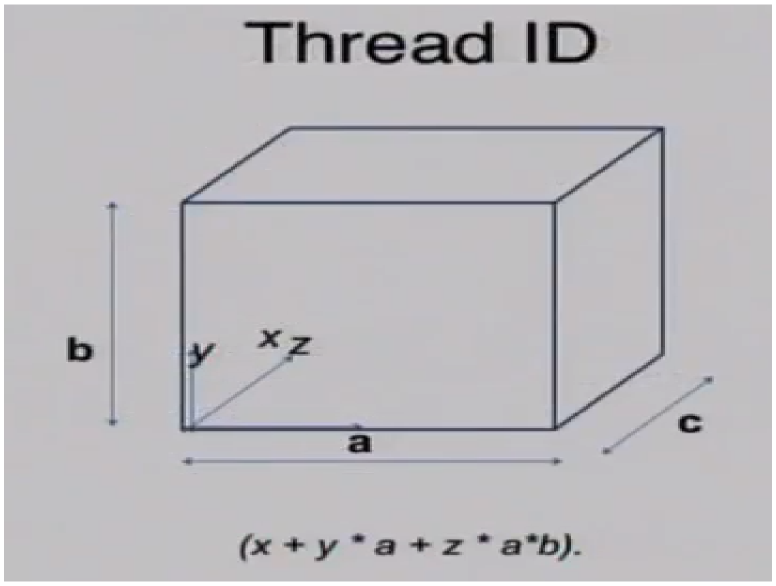
Grid/Block Dimension

- **A** and **B** need not be ints
  - **A** is a (up to) two dimensional vector
    - dim3 **A**(m, n)
  - B is a (up to) three dimensional vector
    - dim3 **B**(a, b, c); a, b, c are ints
    - a x b x c <= 512 on Tesla (1024 on Fermi)
      - Resource sharing further limits the count
      - Up to 8 blocks may co-exist on SM; at least 1 must fit
    - c is the most significant dimension
    - a is the least significant dimension
    - Dereference: **B**.x, **B**.y and **B**.z
    - Thread ID = (**B**.x + **B**.y * a + **B**.z * a*b)

We have talked about this dimension business the only thing that they need to be aware of it is important to note is that defectively turns out to be collimation when it comes from the XY where X is on the horizontal axis Y is on the vertical axis and you access something by XY co-ordinates. So X become column and Y becomes a row X becomes column right so how far the X = 0, 1 , 2, 3, 4, 5, 6 and Y is = 0, 1 , 2, 3, 4, 5, 6.

And so it so far example when you have dimension that is what does it say A cross B then it has A cross B cross C then the indexing will work out to be B dot X so usually it would be K would be the fastest varying variable I times one 2D array J times the row number and then +K and here is the opposite it is B time B dot X + Y times the X dimension and Z times X dimension Y dimensions right.

(Refer Slide Time: 26:04)

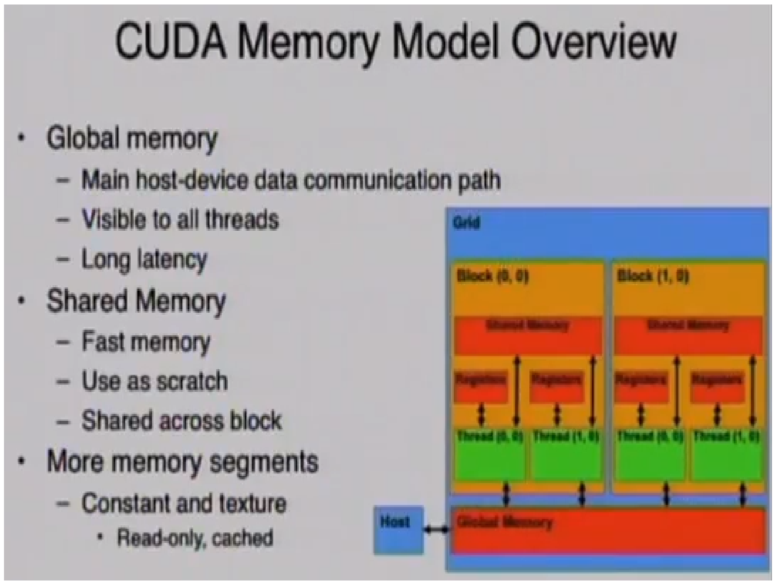


Thread ID

$(x + y * a + z * a*b).$

So it is just the opposite of a row major so that is what demonstrated pictorial okay.

(Refer Slide Time: 26:10)



CUDA Memory Model Overview

- Global memory
  - Main host-device data communication path
  - Visible to all threads
  - Long latency
- Shared Memory
  - Fast memory
  - Use as scratch
  - Shared across block
- More memory segments
  - Constant and texture
    - Read-only, cached

Grid
Block (0, 0)
Block (1, 0)
Shared Memory
Shared Memory
Registers
Registers
Registers
Registers
Thread (0, 0)
Thread (1, 0)
Thread (0, 0)
Thread (1, 0)
Host
Global Memory

X is horizontal axis written in the wrong place it is actually move to the wrong place it was written in the right place. X is what is been drawn here is horizontal Y is what is been drawn here as vertical and Z is one going into alright other than global ensured memory there is constant and texture map okay both of them are cached it turned out in previous architecture shared memory and global memory were not cached meaning that every time you read some data if actually needed to come from the actual memory DRAM.

Now everything is cached the only difference is that constant and texture memory is redone and it is paths are read paths are optimized and you cannot write to it is possible because constant and texture memory are both actually physically in the global memory it is possible to alias this address and so somebody else sees the same address as global and can write to it. If you do that there is no guarantee over what is read by the read only part or somebody else is reading okay.

Global memory is not say that again global memory is not a cache global memory is the memory which sits off the chip right it is got it is own DRAM chip and there is some but that needs from the main CPU to this global memory shared memory is kind of like cache it is on chip cache except it does not behave like cache in that it does not have cache lines defined it does not have that's defined to figure out what is on cache what is not in cache does not have automatic fill and snooping and discard all these.

Scratch pad shared memory there is cache separate outside of the scratch pad am going to since we are anyway at end of the time.