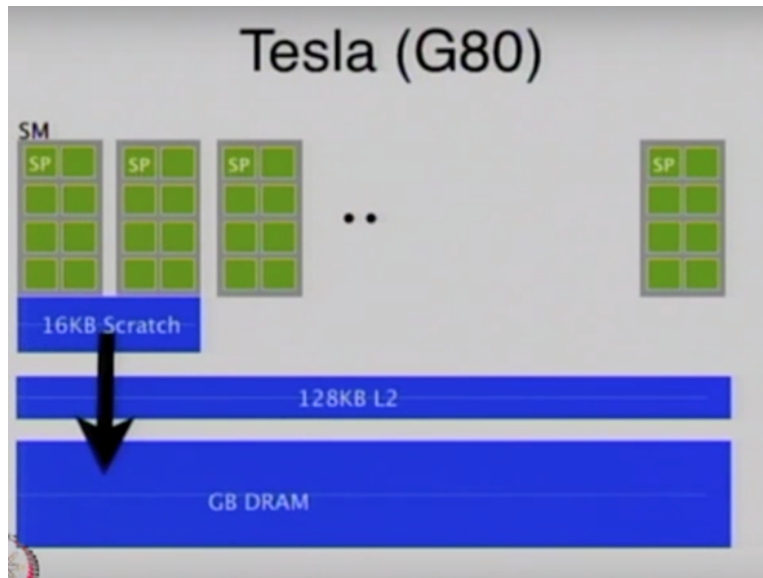


Parallel Computing
Prof. Subodh Kumar
Department of Computer Science & Engineering
Indian Institute of Technology – Delhi

Module No # 05
Lecture No # 21
CUDA

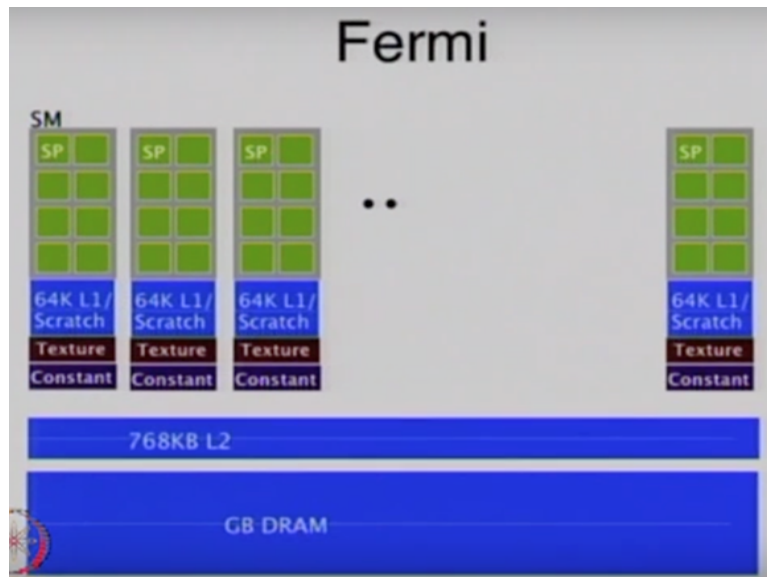
(Refer Slide Time: 00:32)



There are a couple of slides I am just trying to repeat from earlier. There are the machines that you will probably working on will be G 80. Although now slightly modified architecture (()) (00:42) is also available. And we have i think a limited number of those. And essentially you got a number of processors clubbed into groups right.

And for each group you have got some amount of scratch memory. A cache that is shared across all of the processors, and then the main memory, the GPUs main memory. And beyond that remains CPUs main memory which is not shown in this picture. So system memory is outside right, which sits beyond. And in fact when you being when you read things from the disc you typically put them in the system memory first, then copy as necessary into this memory ok.

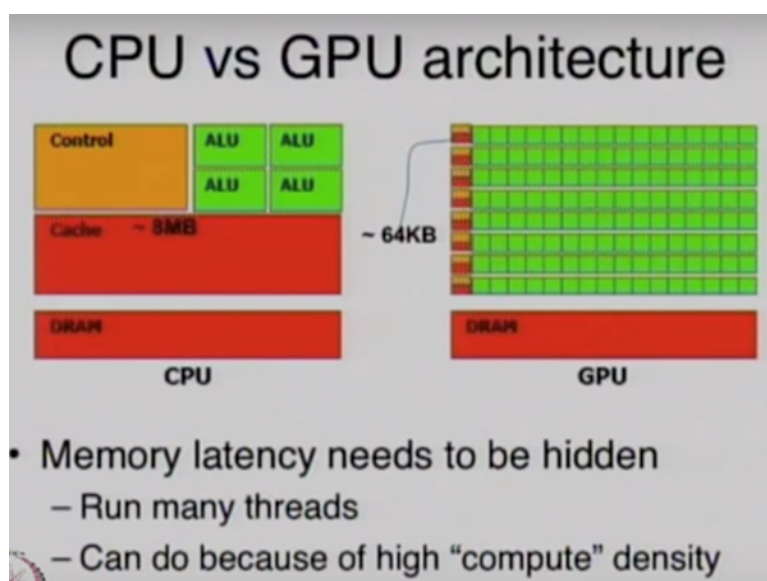
(Refer Slide Time: 01:31)



And this is the fermi version. And then really the only thing and although I have drawn texture and constant as two blocks here, they go in the other version also. It just this is a little bit more detailed. The size of scratch is slightly bigger. The number of processors is typically bigger.

And the L2 size is bigger and actually gets used. If you recall in the previous architecture, L2 was not actually being used. And the rest of it primarily remains the same right. Although I am going to talk about the programming model that is about both, not going to focus on the differences because mainly you will be programming on the tesla, the previous architecture ok.

(Refer Slide Time: 02:31)



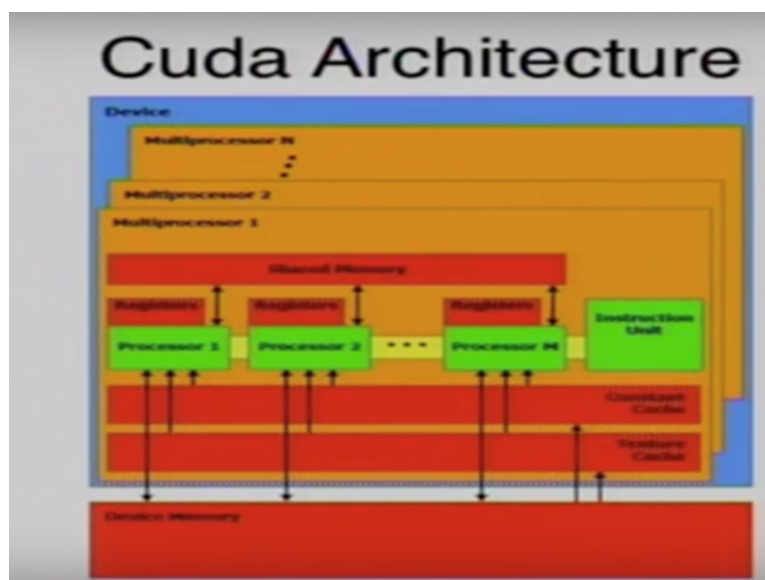
And just to understand why the programming model is slightly different from what you are used to on the CPU side. Look at the architecture. One is that a stark difference between the CPU architecture on the left-hand side and the GPU architecture on the right-hand side is the cache right. CPU has a big cache, GPU has a small cache, as well as the computation. The green is the computation.

And the red is memory, which is divided into the cache and the main memory so to speak. And the cache on the GPU side is this little red that you see with the oranges which are the controls. And all the green stuff is the computation, which means there is a lot more area on the chip being devoted to computation at the cost of cache. Which means you kind of become responsible for using the limited amount cache wisely.

“Professor – student conversation starts.” So if you to finalize that probably GPU is not (()) (03:46). It depends; it depends. You definitely have a big chunk of data which needs to be processed. And the processing is not that the data size which is large. And because we have a smaller cache and larger. But remember that for a large size the CPUs cache is not particularly useful right.

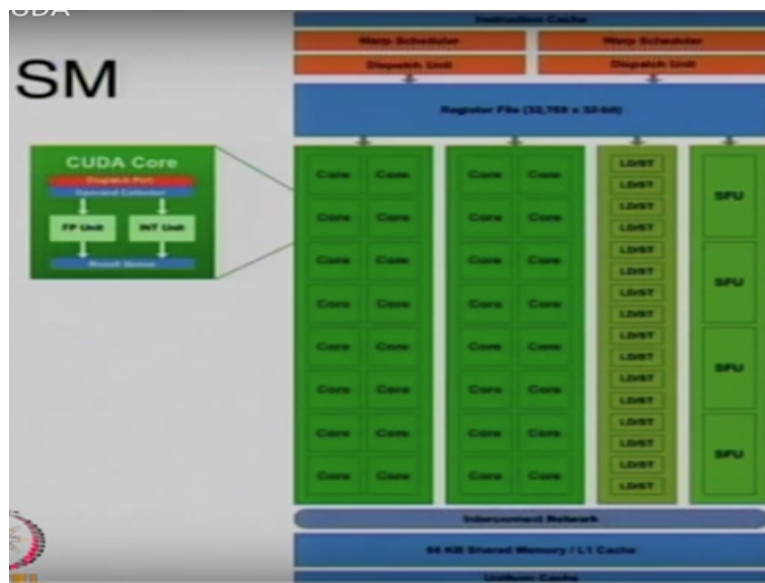
You go there, then you jump here, then you jump there. Its jumping all over is it not? “Professor – student conversation ends.” So CPUs cache is not particularly effective for widely searched item ok.

(Refer Slide Time: 04:26)



And there are some number of these multiprocessors, and each multiprocessor is which are independent right. It is like having multiple cores. But each of the multiprocessor actually is several SIMD processors which are not independent right. They kind of work together their in lock step effectively. And they are working on same instruction for multiple pieces of data ok. Here is a slightly more detailed version of one of these multiprocessors.

(Refer Slide Time: 05:06)



There is a bit of instruction cache. There is this notion or warp which is things that get executed together on a SIMD unit. There are two schedulers so two different warps can get scheduled in the same flock. And then the instruction gets dispatched. And then there is 32 kilobytes of registers available thru these processors combined ok.

The register space is a lot which means that we can get by running many threads at the same time without worrying about context switching. You can assign some processor some register to this thread, some register to that thread, some to that, some to that and everybody's register is live. The core are divided into two groups of sixteen. So that two warps can be done. There is special function unit which does things like sine cos, higher order function which can be done less fast because there are only four of them instead of sixteen.

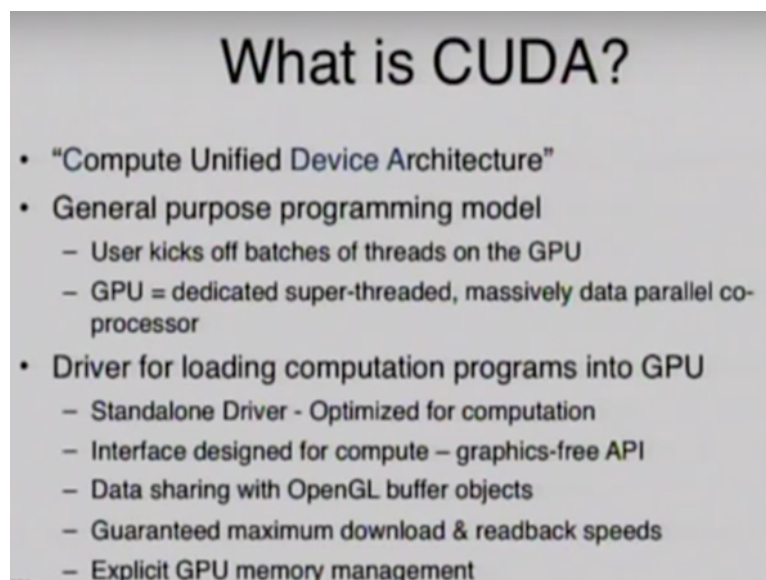
Of the basic operation course. And then that column which is none of them are readable. But they have different colors on my monitor. The third column from the left is the load store. So there are 16 load store units which can be loading or storing at sixteen different times. Fermi there is 64 kilobytes of local space, the scratch space which was 16 kilobytes in the machines you will be using. Fermi also we can partition that into a cache and a scratch which is directly

user control so you decide what lives in that cache rather than some predetermined cache replacement policy. And then there is L2 and all that ok. “Professor – student conversation starts.”

So how many bits for addressing. How many? Bits for addressing? The load stores can support 64 bits of addresses. But the current software level allows for 40 bits of addresses ok. Professor – student conversation ends. Professor – student conversation starts. This thing was, what part of that? So it was one of those multiprocessors; ok so we slices show here. Each slice is a multiprocessor. One of these slices is what was in the other picture. Idea one was just a rotation to show how the GPU has a lot more computation than CPU?

Yes, right that is basically showing the approximate die configuration. Ok. Professor – student conversation ends.

(Refer Slide Time: 08:04)



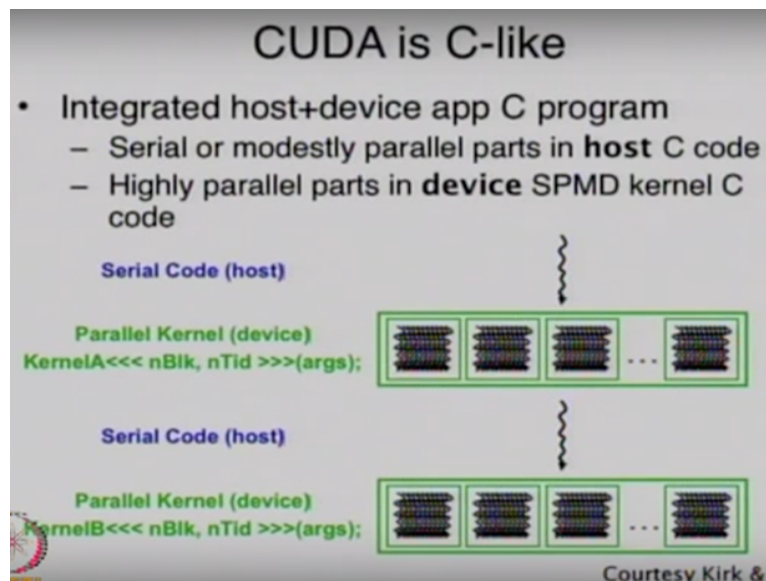
CUDA is compute unified device architecture. The programming model is essentially C centric, C based, with a few more things. Just like open MPA open MP you added a few things to the C to make it parallel. It is kind of similar in principle. Except there you were saying run this many times on same CPU.

Here you are saying run this many times on something else, on the GPU. Some of this will probably not make too much sense because they are related to graphics and open GL. But essentially the and in face I will probably show you the software stag that is necessary. You

are going to need a variant of the compiler, that can compile the CUDA specific stuff inside ok.

And typically it is a wrapper. Which preprocess it to extract the CPU parts uses the standard GCC, CL, whatever you have to compile those things. And specific part that have to be done on CUDA gets compiled separately. And they all gets compiled into one executable or one loaded unit I should say ok.

(Refer Slide Time: 09:41)



And in some ways this is basically it, although there are a lot more details that we will talk about. you have got serial code or whatever, multiprocessor code if you if you wish running on one of the cores. And at one point you say I want to do this a thousand things on the GPU ok. And it is a data parallel model which means you say I want to one this function on this data a thousand times ok.

And so you cannot see the mouse on the parallel kernel divide incantation you see is basically saying run a given function so many times with these arguments. These arguments are basically do all of those instances on that function ok. And so in in this example parallel kernel would be a function that you have written ok. And it is going to be invoked by saying kernel there are these less than and greater than brackets which are those things that are added to C which is going to get started by the CUDA compiler.

It says how often should this be run. We will we will get to details shortly. And then the bracket says, these are the arguments sent to be sent to that function ok. And then you go

back and start continue your serial execution. And later on you can do another kernel and so on ok. And the picture in in in the right side says that there is a single thread of control then this massive multi-threaded thing got executed in parallel.

And then your thread of control continues on from there ok. It is possible to overlap them. But here you are waiting for CUDA two, CUDAs computation to compute. You take its result. Do some more processing on it. And then say do some more. “Professor – student conversation starts.” So you cannot do nesting right? You cannot do? Nesting I mean in parallel. Inside a kernel? Yeah.

You can actually but not in this fashion. It as far as the compiler will be concerned it will still be one kernel. One kernel with only that size. And you have to indirectly, in each one of them do multiple things. So in the definition of kernel we can write this kind of lines. Which kind of lines? (12:38) Yes, not in the definition of the kernel, in the call to the kernel right.

There will be the function dot E, which be will be the definition of the kernel and somewhere else there will be a call to it. So the three things is for at the call time. So in the definition of a kernel we can call other kernel for nesting? No the compiler does not allow that. You will have to manage that internally inside the kernel. Inside the kernel, you have to do something like that.

Is even spanning is not allowed. I mean in a value. Yeah, effectively that is what you will have to do. You will have to manage it ok. “Professor – student conversation ends.” And at a high level its it its designed to be a coprocessor. That means you do your main program and at some point some function you want to execute its an expensive function like H two long to compute on the CPU you say now do this function.

Only this function on the GPU. And if I were continuing to do that on the CPU I would have to wait a long time before that gets done. And if I submit to the CPU, the result will probably come out faster ok. There is a separate memory which GPU has. It cannot get data directly from the system memory, which probably will change overtime. But right now it cannot, which mean every piece of data needs to be transferred copied by you into the GPUs memory before the function will start.

Professor – student conversation starts. there is no other way for you to keep it? Sorry? Combine the CPU with GPU they have they have that architecture; I do not know how it works but. Yeah nobody does. this programming model is not very clear at this at this moment. They are trying to see open CL which is very CUDA like is going to work on it.

But its still work in progress. (14:50) are a complete operating system and after the initial boot and loading of the drivers, all the user space programs, they are called thru these kind of like, they are called thru CUDA. Then it is it possible to have like running multiple instances of operating systems, CUDA each signal processor, particular instance, something of that.

No, it is bad with context. it getting better. So for me, contexts, you can have multiple applications independent applications be using for me at the same time. but the context which is expensive at least from the CPUs standards right. i think for me its around twenty-five microseconds and for the previous generation it needs to be at the range of half a millisecond or so.

Professor – student conversation ends. so as far as usage is concerned, the intended usage is still is that you will be using one application which will be at one time using this CUDA device. so if you have lots of different applications using the CUDA device your parallelism will be effective by all these overheads. “Professor – student conversation starts.” We can partition the device.

One way is to partition it in like no this times dot in one previous end users the complete device for a certain time. can you do like these many number of co-driving used by this particular. that is just a software issue and its currently cannot be done. but without real change to the hardware this can actually be done. Somebody who is working on the software side will have to find it valuable to do this or it is possible to do. “Professor – student conversation ends.”

(Refer Slide Time: 17:17)

CUDA Devices and Threads

- A compute **device**
 - Is a coprocessor to the CPU or **host**
 - Has its own DRAM (**device memory**)
 - Runs many **threads in parallel**
 - Is typically a **GPU** but can also be another type of parallel processing device
- Data-parallel portions of an application are expressed as **device kernels** which run on many threads
- Differences between GPU and CPU threads
 - GPU threads are extremely lightweight
 - Very little creation overhead
 - GPU needs 1000s of threads for full efficiency
 - Multi-core CPU needs only a few

Alright so the main thing is that it is basically offloading your entire computation. And it is going to be running typically many threads hundreds, thousands, something of that sort. And the a typical CUDA type application will at many places be launching several kernels ok. And so the copying of some memory to CUDA as well as launching beginning of a kernel onto CUDA is supposed to be as efficient as possible.

Because otherwise if it takes too long to send this and do this on CUDA might as well spend the time doing the computation on the CPU itself.

(Refer Slide Time: 18:12)

Extended C

• Declspecs	<code>__device__ float filter[N];</code>
– global, device, shared, local, constant	<code>__global__ void convolve (float *image) {</code>
	<code>__shared__ float region[M];</code>
	<code>...</code>
• Built-in variables	<code>region[threadIdx] = image[i];</code>
– threadIdx, blockIdx	
• Intrinsics	<code>__syncthreads();</code>
– __syncthreads	<code>...</code>
	<code>image[j] = result;</code>
• Runtime API	<code>}</code>
– Memory, symbol, execution management	<code>// Allocate GPU memory</code>
	<code>void *myimage = cudaMalloc(bytes)</code>
• Function launch	<code>// 100 blocks, 10 threads per block</code>
	<code>convolve<<<100, 10>>> (myimage);</code>

And here is a quick glimpse into the changes to C that CUDA introduces ok. there are function type specifiers ok. This is called declaration specifications. And a function can be at global, a device, and this is not function. Actually a piece of memory can be global, device,

shared, local and constant ok. Remember those two blocks that I had added (18:55). One said texture, one said constant.

Those are specific partitions of memory the same D-RAM, the global space of memory that is accessible to all of the processors. That gets partitioned into a global memory, a textured memory and constant memory. The paths of each of these memories to the multiprocessors are different ok. The path from the texture and constant memory is speeded up, much faster. It has got its own data path.

As a result of which you cannot write to that (19:42). It is a read only (19:44). Once you initialize it your CUDA program or your multiprocessors individually read it. It cannot write to it. Viewable memory is something you can write to. it is so to speak the main memory on the CUDA side. Shared memory is the cache right. The scratch pad we saw in the architecture. It is much faster.

It is almost for all practical purposes, it is as faster as reading a register. So you can think of 64 kilobytes of additional register space being available essentially, in principle. so what you see is an example on the right-hand side. It says device float filter some N right. So float filter is regular C. You can add a declaration specification to it. It says that this variable must be allocated on the device ok.

It is not a CPU variable, it is a GPU variable. when you see the global, and by the way the although on the left side just the keywords that are being added to C are listed. When they are used in a program, there are two underscores in the beginning and two underscores in the end. So underscore underscore device underscore underscore. To say that it is a device variable. global void convolve says that it is a function which is visible on both the GPU as well as the CPU.

What that means is that you can call it on the CPU. You can run it on the GPU. you call it in one of those kernel invocations alright. The third example which says shared float region N is saying that variable needs to be allocated in the shade memory L1. there are some built in variable. One built in variable you always need is your ID. Either function or a variable. Here instead of function it is a variable.

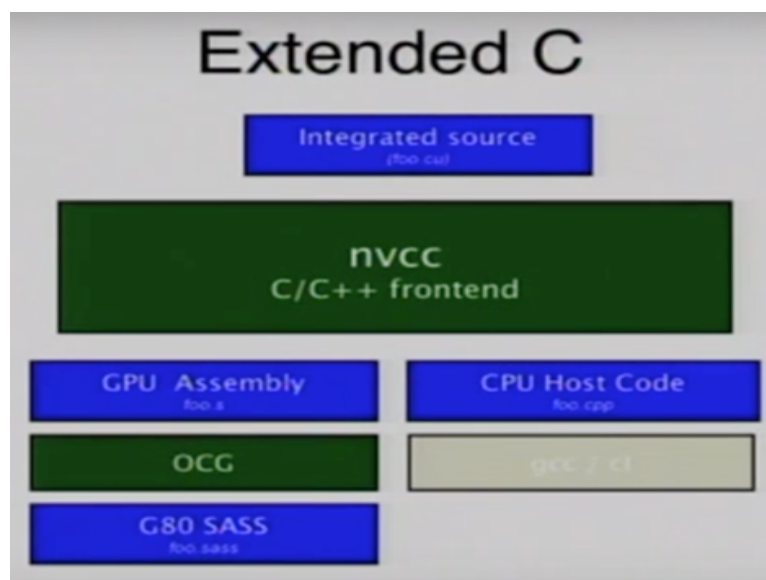
And so every thread has to know its ID so that you can if there are hundred things to do you are number three, so you do the third thing ok. the I will get to more detail on the ID. And so it is not a single ID that is provided to you but it is kind of your location in the space of configurations. And it is not just a thousand things to do but it maybe I have got ten by ten things to do.

Each thing is a twenty by twenty thing ok. So then your ID is slightly more complicated right. It is not just number in in (())(23:09). There are special functions. For example underscore underscore sync threads. It is a barrier. Barrier in the context of things running on a single multiprocessor. There is at the moment no barrier across multiprocessors except by involving the CPU.

Alright and the last line, before the last line there is allocate of memory. So you can do Malloc to get memory on CPU. You can do cudaMalloc to get memory on the GPU ok. And you will get some handle to it so that you can then copy from CPU to GPU or make the copy (())(24:02) make the copy a request on the CPU side. and to launch a function which is what we call a kernel in CUDA terminology.

Which will be of global variety when you have declared a function to be of type global you can call that function with these three brackets with some information that says how to start this function ok. And then some parameter, that the that function takes.

(Refer Slide Time: 24:44)



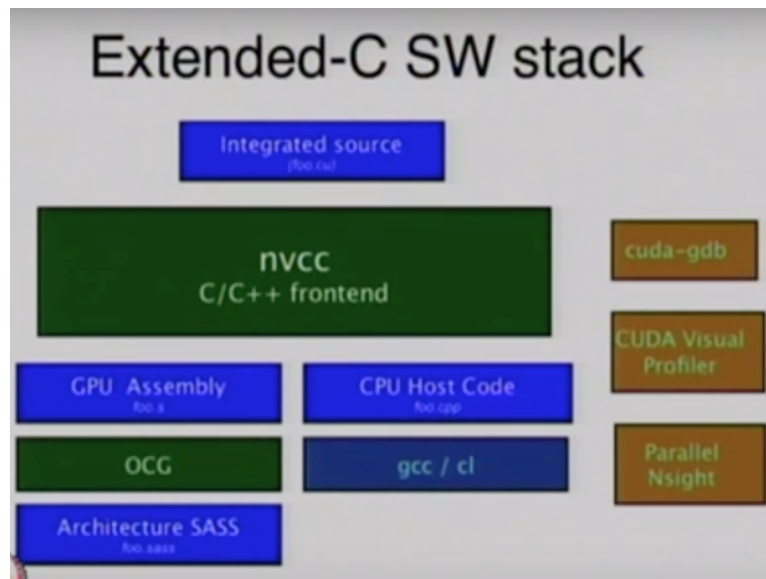
Let us look at let we will stop here. this is the software stack of things that that you are going to need. you have your CUDA program right. Which is really C with all these things we saw in the last slide added in. And you run something called NVCC on it. It is the wrapper for the compiler but CUDA programs. or sometimes also called the CUDA compiler. It is going to separate out the host code.

Host code being the basic C. And the GPU code GPU code it does something like what java does right. It has got bytecode that java virtual machine is run. It has got something called PTX that it generates, which all CUDA devices are supposed to run. Or there is a layer at the bottom that will convert PTX code to the given CUDA device. So PTX code is a bit generic architecture independent assembly language ok.

And at the end there is this SASS compiler which is going to take the PTX code, generate the machine code for a given architecture that you know at the moment that you are running on. and so if you were to say write software and distribute it and you compile it to PTX, it can be run on all the CUDA machines that people may have. But if you compile it to SASS or SASS compile it then it will only run on that specific type of CUDA device on which it has been compiled ok.

So that is the difference. and at the bottom of it is the driver, the CUDA driver. Which is going to take this piece of code that SASS generates and send it to GPU in a form that can then be executed, loaded, executed and so on and so forth. So we are going to stop here. We will look at a few more of the details and then look at a few of the examples next time.

(Refer Slide Time: 27:34)



Ok so we were talking about this the CUDA software stack and I will briefly repeat. So you write, I have given you a high level picture here. I we will we will get to so me of the nitty gritty soon enough. but there is a source code you are going to write. probably in a file. Probably with an extension dot CU to signify that it is a CUDA program and that will have stuff that gets executed on the CPU as well as stuff that gets executed on the GPU right.

So the general structure is that you have you have got a CPU program and in the middle of program which may be running in a multithreaded way or a single threaded way on the GUP on the CPU side. Somewhere along the way you can say run this thing on the GPU. And then the result will come back and you continue on from there ok. And so you are, the same program has code that will execute on GPU as well as code that will execute on CPU.

And this NVCC is the compiler that is going to separate out those two parts ok. It is actually a piece of software that has lots of modules or other sub software that are even individually available. So it calls several things in the pipeline. so it is really a wrapper. that NVCCs main job is to separate out the CPU code, what is the GPU code and for the CPU code call your visual studio GCC whatever compiler you may have.

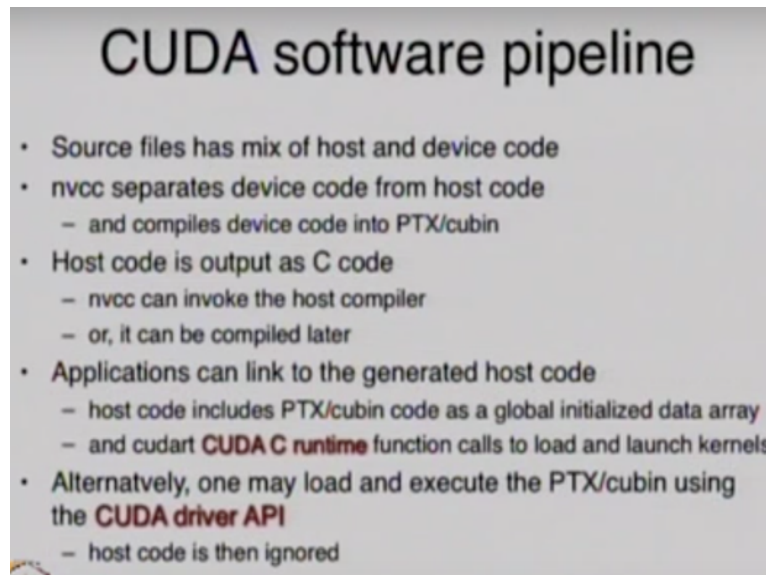
Of course it has to be compatible with the product of NVCC and most compilers that you use typically are. and then it is going to separate out what is going to execute on GPU call proper compilers. There is an intermediate language called PTX that is going to compile it so to speak into this intermediate architecture independent assembly language. So that is what is

indicated here with some foo dot S, which can learn in a variety of on a variety of GPU platforms.

And then once you know what GPU platform you are going to run on you are going to compile further using OCG. Which is essentially optimized code generator. It is going to take PTX code, optimize it at that point. It will figure out how many registers you have. How to schedule it among the different number of different multiprocessors and then so on. And then generate the actual assembly language for a given architecture for a given GPU that you may have.

And in the meantime the CPU had produced the CPU side ok. Now you have got two pieces of code. How does how do they interact? I will I will show you shortly. But there is also support for some debugging. cuda-GDB exists it is extension to GDB. and CUDA has a profiler so it is going to tell you what function is taking how long. it there is recently also a release of something called parallel insight. Which plugs into visual studio. So you can directly from your visual studio debug it, program it, run it, execute it and so on ok.

(Refer Slide Time: 31:24)



CUDA software pipeline

- Source files has mix of host and device code
- nvcc separates device code from host code
 - and compiles device code into PTX/cubin
- Host code is output as C code
 - nvcc can invoke the host compiler
 - or, it can be compiled later
- Applications can link to the generated host code
 - host code includes PTX/cubin code as a global initialized data array
 - and cudart **CUDA C runtime** function calls to load and launch kernels
- Alternatively, one may load and execute the PTX/cubin using the **CUDA driver API**
 - host code is then ignored

And so this is what is going to happen. the source file is going to have some CPU, some GPU. So it is GPU is what is being referred to here are the device. CPU is a host. and we will see we will separate it out. Depending on the options we give to NVCC it may compile it all the way to what is called cubin right. It is the object code for a given GPU instance or to PTX which is an architecture independent assembly language.

And then it is going to also generate host code on which it will either evoke the compiler or simply generate the source code which you may then at later stage compile. And the application, so there are two modes in which you can do this. the output of NVCC can be in one encapsulated library, application whatever you want to call it. depends on how you what flags you give to the compiler.

And from the application you see this GPU piece of code as data. It has got an array filled with some GPU program. And there are function that CUDA provides. It is it is CUDA runtime when you install CUDA you automatically install what is known as CUDA runtime. And CUDA runtime will load this memory array. and you can directly using CUDA runtime functions do something with these GPU pieces of code ok.

Alternatively you can directly interface with the driver. So every device will typically come with a device driver which will have some interface which CUDA drivers also do. And thru the driver interface you can from your main program read the encapsulated piece of GPU code and send it to the driver saying execute this ok. So you can do it from an application directly talking to the driver at runtime or compile into the application using CUDA runtime functions, call appropriate functions to load the GPU function ok.

When you create a full GPU CPU program, you typically do not have to do anything right. You just have to execute something dot exe and the proper CUDA runtime calls have been made by the library calls you made to make sure that something that gets execute that is destined for targeted for GPU will automatically get sent to GPU. It is a get executed and the synchronization between CPU and GPU will happen.

Professor – student conversation starts. Suppose (())(34:34) driver cannot does not exist in some system they also (())(34:38). CUDA runtime uses the driver right. Either you can directly talk to the driver or you make function calls that talk to the driver. Sir after all that driver is going to (())(34:49). Yes in in both cases. Can we take the particularly wait for the (())(34:56) architectures it is not below.

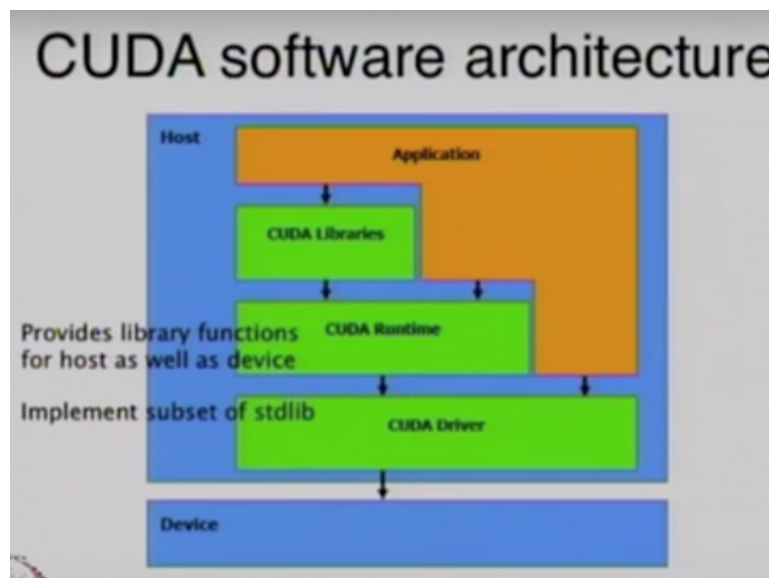
Cubin is just a term. Just like you say dot OBJ, it is dot cubin. Maybe some difference between object past PTX and past cubin. Absolutely PTX and cubin are two different things. PTX is architecture independent and cubin is architecture dependent. So you will generate

different cubin for different hardware right. If you have a lower end laptop GPU your cubin will be different from a few running on a high end tesla for example.

Ok but PTX also has two versions. they have because fermi architecture has some more innovations that tesla did not have. And the PTX was not designed to take that into account. So the now PTX is more general more powerful than the old PTX. but the new software that will take PTX and generate cubin will take old PTX also right. So if you have a legacy PTX code, it will run on new GPUs. It will just not be using the advanced features.

So what you mean by that global initialized data (())(36:10). Global initialized data is as far as CPU id concerned. every piece of GPU code is an array right. It is data, it is never to be executed on CPU. So it is an initialized data and you can go read thru it in your some CPU program. You have access to that pointer. But it those instructions would not mean anything on CPU drive. So in that sense it is a data, data array.

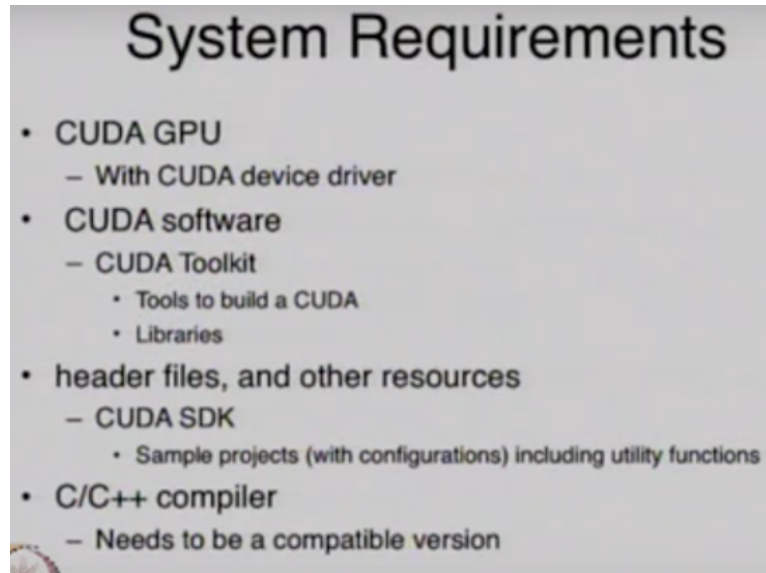
(Refer Slide Time: 36:40)



The software architecture basically somewhat of the rehash of the same thing I told you about. The application sitting at the top can talk to CUDA libraries which would be most common usage which in turn will talk to the runtime. And then runtime will be talking to the driver. And driver knows how to talk and interact and synchronize with the GPU ok. Or the application can directly talk with the CPU with the driver.

Which will then eventually talk with the GPU. so all the examples that I am going to show you will just be the library version. You make library function calls which the magic is happening in the background.

(Refer Slide Time: 37:28)



Alright so you are going to need a GPU to run these programs. So the exe that you will be generating will have the GPU code built in. And that GPU code as far as CPU is concerned is just data. However as long as you have enough things around that GPU code will run right. There is a driver and there is a device sitting on the other side of the driver is going to work. There is it is possible that that GPU code is a PTX code because that is what you instructed NVCC to do.

Device driver will compile it to the given device ok. It knows how to call the compiler. and that is that is called just in time compiling ok. the other thing other than the device and the driver you really do not need anything to run it ok. Which means somebody gives you a program. They programmed it they compiled it. You have got the driver and the device. You can run that program on your machine right. That is how people will distribute code.

And as long you have got two driver installed and your machine is up, it will run if it is PTX complied. If it is compiled to a specific cubin, specific architecture. Then then the driver will be able to do nothing right. Driver can only say I have go cubin, I can run it on the GPU. But the GPU will not understand that cubin. So then it will not work ok.

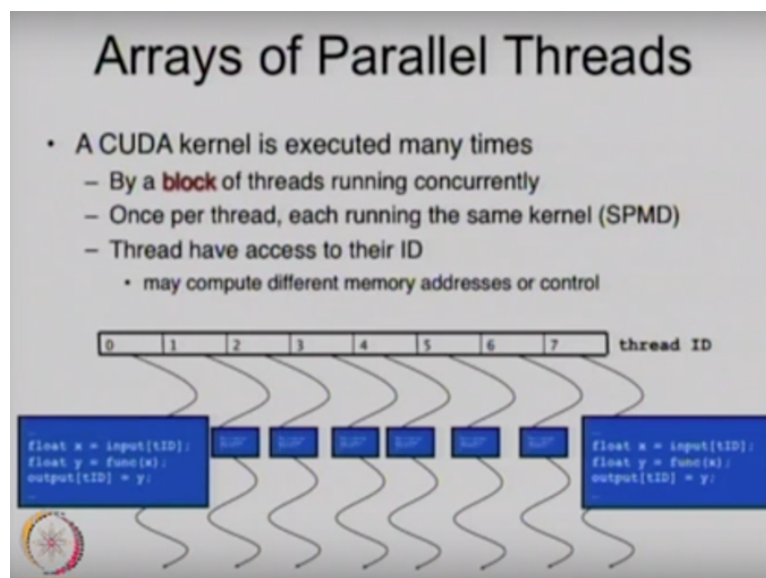
So this just in time compilation business is necessary for especially because unlike CPU, most GPUs are there are many GPUs in the market today which all have slightly different architecture. More than that on a CPU you will say I have got a thread to run and then that thread will run on a core. Here you will say I have got this code to run and I want to run it as well as possible right.

I want to run it on as many processors are available. I have broken it down into some number of units and if you have got one GPU processor you run it on one place all the units. If you have ten then you however you schedule. It is typically around robin. But you can take the first run it on the first unit, second run it on the second and so on. All of that can be done only if you know how many units there are.

Alright when you are compiling you need a little bit more. When you are developing you are programming you need the software tool kit. There are some utility type functions built in. so there is something called cutil that you will get used to. We will use often the library will also have the standard STD lib style stuff. You will not have input output because that is not allowed on CUDA directly.

But most of the basic stuff, string app, string libraries and stuff will be there. And then you include the proper C header files as well as CUDAs header files. And you if you are writing GPU code you will have access to the same function on the CUDA side. If you are writing CPU code you will have access to the same function on the CUDA side ok.

(Refer Slide Time: 40:58)



And so this is the basic way in which it gets done. You write a function. pretty much it looks like a CPU function. We will see examples shortly. And you say run this function so many times. That means in this example you have got a block of eight threads running the same function ok. That those blue blocks although they seem smaller in the middle I just scaled down the version of the big blue blocks. So they are running the same function.

That is why they are called; this style is called single program multiple data ok. they can they will have their own ID right, thread ID. And based on that thread ID they can do different things. They can fetch different pieces of data. They can do different things if they want to. That is why it is not strictly SIMD, but as we have discussed before also that some of the tightly bound one multiprocessor is designed as SIMD.

So all the SPs of a single SM run in a SIMD fashion meaning that if they have some threads wanting to do X, some other threads wanting to do Y, then there is some magic that will make X happen first then Y happen for the subsets of threads that you want to do X or Y ok. you can and this is what is called a block ok. So this is what I was saying that the scope for sync threads is a block.

Here I am only showing that block is a 1 D array of threads but you are allowed to have 2 D and 3 D array of threads also. What that means is that, only thing that it means at the end of the day is that your index will have I and J as opposed to just I or I, J and k as opposed to just i ok. The idea is that in many applications, specially in scientific computations you are dealing with a grid of sorts right.

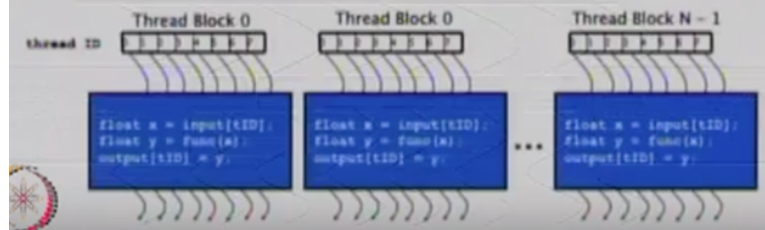
You have you have sampled a 2 D or a 3 D space and you have said here do something with my neighbor's data, do something with my left neighbor's data, do something with my top neighbor's data. You can compute the address of you left, right, top, bottom neighbor using some row major addressing that that we are used to. Or we can simply say if I am IJK then get $I + 1$ JK, $IJ + K$, $IJK + 1$ right. So this is this is I, this is J, this is K.

You will see that blocks have a very tightly integrated set of synchronization tools. So they can do a lot together cooperatively.

(Refer Slide Time: 44:02)

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



But there are also notions of blocks ok. so you have got an array of blocks now. And the term you use for that is a grid. So you do not necessarily have an array of threads working on a kernel. You have a grid of blocks of threads working on a kernel. You have got a grid of blocks meaning there is a block and again the grid can be 1 D or 2 D.

And each block in that grid can in itself be a group of threads which can be organized in 1 D, 2 D or 3 D fashion. And then each thread runs that kernel ok. so now when you in your pro in you kernel you will be thinking of what does a thread do right. Because every thread is executing the same kernel. So now you have to think in terms of, I have to do this and there are other members in my block doing whatever they are doing.

And then there are other blocks in the grid that are also doing something else ok. And you can talk to as I said other member in you block in a more tightly integrated fashion. So there are these usages of there is something called shared memory. it is similar to share shared memory that you know of in from open MP days. there is something called global memory.

Which is also shared memory from the point of view of open MP ok. the difference is that shared memory is really shared by the block ok. Its entire life and context is a block. Whereas global memory is shared across the block. It is a grid different blocks of the grid see the global memory as shared memory from again not using (())(46:25) terminology but general shared memory terminology.

But we will we will just so there is no confusion we will from now on refer to shared memory as that that is shared threads of a block. And global memory as that that is shared between threads of different blocks ok. Professor – student conversation starts. So this blocks can be arbitrarily large right? No, they cannot be. Because last time when we defined shared memory we talked about the scratch place. That is shared memory.

That scratch space is where shared memory lives. That is limited; so that memory is limited? Yes. But block sized can be limited by the size of. Not necessarily right. you can imagine blocks that do not use any shared memory at all. We will be working sequentially. Will there be enough stuff for all the (47:34). You already have. As much as possible right.

So if you have got only a block always runs on a single SM ok. We are further dividing that block into grid? No. We are dividing grid into blocks ok. So block is a tightly integrated group of threads and that tightly integrated group of threads must run on a single SM ok. Single SM is just 16, 32, 16 wide SIMD processor ok. The different blocks of the same application or the same kernel can execute on different SM.

And that is typically scheduled around robin or something of that sort ok. So if you say that I want to run a grid which has 16 blocks. Each having 32 threads then the 32 thread group will be on one SM, but if you have got 16 SMs then all the 16 blocks will be scheduled in parallel. But if you had only eight SMs, then first eight will be scheduled and then as they get done the other eight will get scheduled on the same SM.

So each block can do different things? There are no SPMT restrictions on that block? There is there is. The idea so SPMT do not think of it as a restriction. It is more of a programming framework right that you took a kernel you provide a single function. And that kernel is shared by the entire grid. Now within it you can check your block ID within a grid, or your thread ID within a block.

You can do whatever you wish to do. And in fact there is really no penalty to be paid if you have different blocks of a grid doing completely different things. Because they are running on different processors. But there is some penalty to be paid if within a block you are trying to do different things. Why do why do we have these connotations. I mean so if I just go ahead

and (49:51) function then why do I need to say that? The differentiation among grid and blocks.

Why cannot I simply say this is block of one thousand or ten thousand threads. it is because you have limitations the entire block is live in context. You have only sixteen processors and sixteen kilobytes of register space. And so if you have a program that needs much more than that each thread needs its own register space right. And so the more the number of threads more the size that you have to keep live in context.

And hence there is a maximum size to the block. But sometimes you want to do bigger things more things. And so then you say I will take however many I can fit in one multiprocessor then I will have as many of these as necessary depending on my problem size. There is no compensation between blocks? Whatever block (50:55) finish and then the other block will come in or?

No, they are all live. So if two blocks that are scheduled on the same SM can both execute ok. they are concurrent. then this block will be executing independently without affecting any other block. We can design our blocks in such a manner? They can affect in the sense that they share the global memory. If I write a one block writes, a thread of a block writes something to the global memory a thread of another block shares it so they can communicate that way.

But there is very limited guarantees in terms of memory consistency model across blocks and so on and so forth right. You cannot even recognize across blocks. Except within a block. So you can you can have you have stringent guarantees for memory accesses within a block because you can. Hardware allows you to do that so the programming model essentially is fitted onto the hardware.

That is that is the way it was designed. there are also atomic operations meaning that when you say add something to global memory or shared memory or wherever that entire operation. Or add one to shared memory or read off the memory, add one to the data, write the result into the memory becomes atomic. So no this cannot be interrupted by another read or seen by interfering with another read. there is also barriers.

Again barrier is really only within a block. There is notion of sync threads. And you can say barrier within a block. But you cannot have barriers across blocks. And again it comes from the same reason that all the members of a block are live in context and if you have many blocks which do not fit on your GPU, that cannot that can never execute right. A grid has one thousand blocks, you can only fit three blocks on a give GPU right.

So those three blocks will run but they cannot get switched out. They are taking up the entire GPU. But if in the middle of that block you say barrier across all of the grid the other ninety-seven do not get a chance to run. So that grid is done right. I mean that barrier is done you can never come out of it. You can say that block are in context with each other, entire block can be put out if multiple blocks are sharing the same memory.

No context which is only across applications. And there is a significant penalty for that. Right two applications can both be using CUDA and that is also only for the recent architecture. So one block can finish and then the other block. Yes, no. Multiple blocks can be live but they do not con they do not switch context right. They are all live. So I have got ten blocks that can fit in at the same time.

So the scheduler now says I am free. My adder is free, my ALUs is free. Which of those ten blocks has the next instruction that can be execute scheduled on that day? Ok they all have their registers ready. All the ten blocks. So there is no context switching among them. (()) (54:48) Because there are that many registers when you. in that example. There are only three live blocks at time? 3 SMs there are only three SMs and there are a hundred blocks.

So in that case the rest of the ninety-seven blocks do they get to execute? You said no right? They do not get to execute even a single exertion unless and until the first three full blocks go away. Exactly, so those three blocks are live. So the ninety-seven blocks are not live? They are not live, they are not going to be, there is no switching across from these 3 to 97.

These three will complete, or one of them will complete and then one of those 97 will be scheduled ok. But these three, there is no switching between them. Because there is switching of scheduling right; there is no switching of context. everybody is live and at clock speed the scheduler is saying whose instruction is ready. Whose operands are already in the register in their respective registers.

And then some K number of the blocks will be ready and it is going to execute one of them. No but ALU, how many ALUs are? We have 16 ALUs. that is it. Yes, and there are two groups of 16 each. But one of one warp which is a subset of a block executes in 16. Why does it need to ask? Because somebody may have read something from memory. the register is not filled with the data yet. But each ALU, each SM has its own ALU?

Each SM has its own ALU and each SIMD unit on the SM, there are 16 of them, has its own ALU. Right. And one in in a given clock cycle on one of those 16 SIMD units you can execute one instruction for 16 threads right. let me stop because I think we have come to the end of half an hour. And I will explain it shortly after.