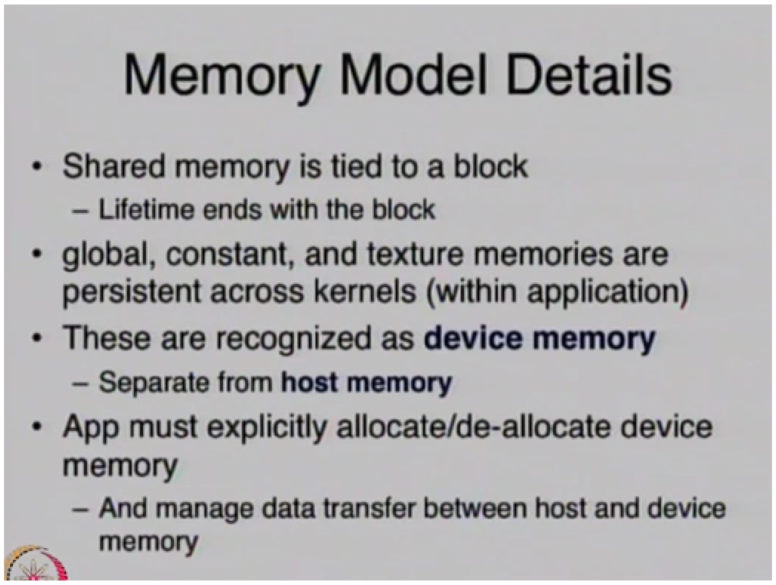
Parallel Computing
Prof. Subodh Kumar
Department of Computer Science & Engineering
Indian Institute Technology – Delhi

Module No # 05
Lecture No # 23
CUDA (Contd)

(Refer Slide Time: 00:28)



Memory Model Details

- Shared memory is tied to a block
  - Lifetime ends with the block
- global, constant, and texture memories are persistent across kernels (within application)
- These are recognized as **device memory**
  - Separate from **host memory**
- App must explicitly allocate/de-allocate device memory
  - And manage data transfer between host and device memory

Let us continue a bid more on the memory. So we are kind of mixing how the hardware is organized and what the programming model is. Because they really are very closely sync and tied together. So the shade memory as I mentioned earlier also is completely tied to a block right. So it starts when a block begins ofcourse hardware wise the shared memory is always there.

But the data the access to the data that visibly of the data begins when the block begins ends when the block ends ok. So if a three block beginning then each is going to have some piece of shared memory based upon to it. And the blocks will not be able to know about other block shared memory. The global constant and texture memories are as even the name suggests it's not globally visible to all the blocks.

It is persist meaning that it has a context from kernel to kernel it has the life of the entire application. So if you start a block end it. Start another block you cannot expect that the shared memory has the same data that the previous memory ended with. Because there is no notion of previous block however in the in this global memory where also texture and constant memories reside you can have ten kernels and the previous kernels output can be read by the next kernel.

So that data is alive all the way until the end of the program just like system manner. When you write something into the memory until your program quits. That data is there once if you re-begin it. You have no of knowing what data is that the thread local memory. Yes it is in global memory and it is slower than scratch memory meaning slower than shared memory which means that you should not be using.

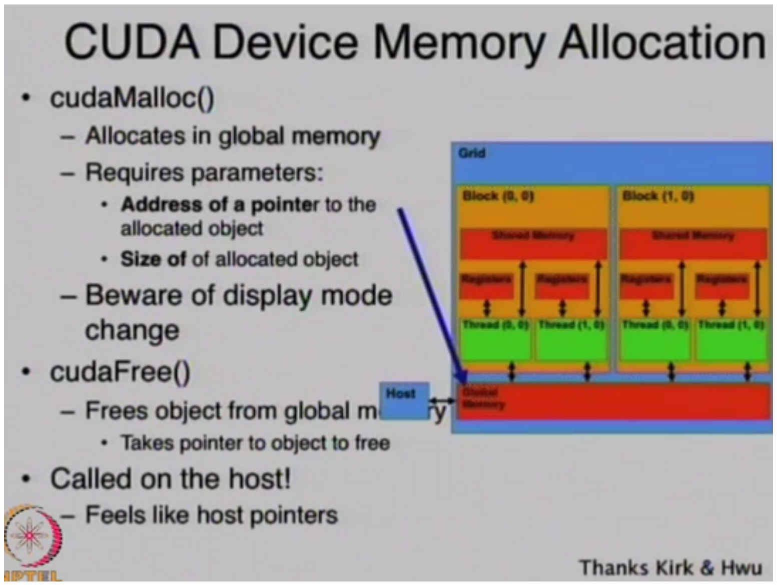
So in practice what that is what does it says do not be used to many local variables as long as those local variables can be fit in the registered space. It is fine but if you have a big array right you declare a local array of one thousand that you can expect it to go to the global memory regular than the global memory. The host memory it is a is physically separate right different from the GPU memory which means typically whatever data you have on host especially when you are beginning the computation.

We will have to copy it to the GPU. There are also notions of mapping the memories. So some sections of the host we can mapped to the GPU memory. And then you can write on host as if it is a local memory to you and on the GPU you can read ok. So that is also possible but one way another you will be explicitly allocating space for whatever you want to fill in the global MEM in the GPU in the global memory and filling it by either physically doing MEM copy or assigning things to the variables.

But you do assign only when you are able to map and if you take some CUDA memory on GPU memory and map it locally as a memory address on the CPU side. Then you can assign it a variable assign it a value and it is going to stop there on the GPU side.

There are some limitations to how much of that you can do and what it does to the rest of your system memory. A more general approach which is going to work all the time which is does not have some of those instructions that the mapping thing does is you allocate whatever instead of MALLOC you would have calling CUDA MALLOC I will show you couple of examples

(Refer Slide Time: 05:24)



CUDA Device Memory Allocation

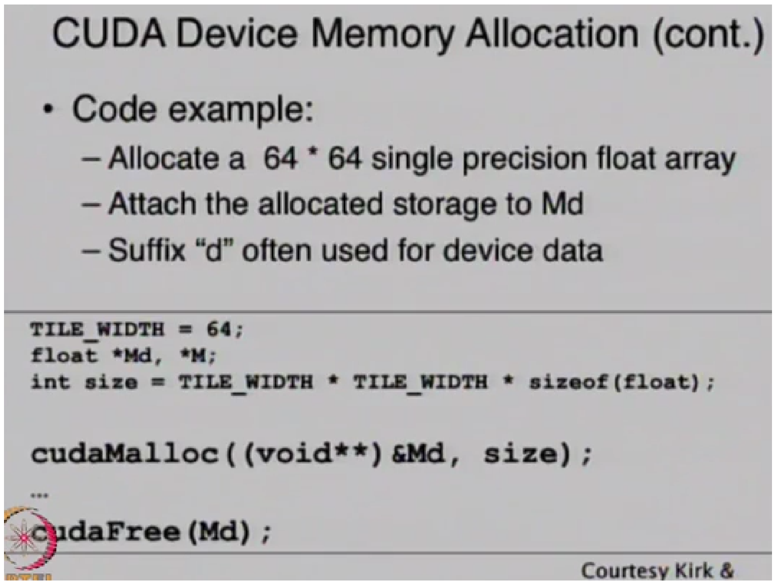
- cudaMalloc()
  - Allocates in global memory
  - Requires parameters:
    - **Address of a pointer** to the allocated object
    - **Size of** of allocated object
  - Beware of display mode change
- cudaFree()
  - Frees object from global memory
    - Takes pointer to object to free
- Called on the host!
  - Feels like host pointers

Thanks Kirk & Hwu

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

I think right here you will call CUDA MALLOC you will get the address and on into the address. You will do CUDA MEM city ok MEM copy and then at the end you do CUDA free if you do not need that space.

(Refer Slide Time: 05:42)



CUDA Device Memory Allocation (cont.)

- Code example:
  - Allocate a 64 * 64 single precision float array
  - Attach the allocated storage to Md
  - Suffix "d" often used for device data

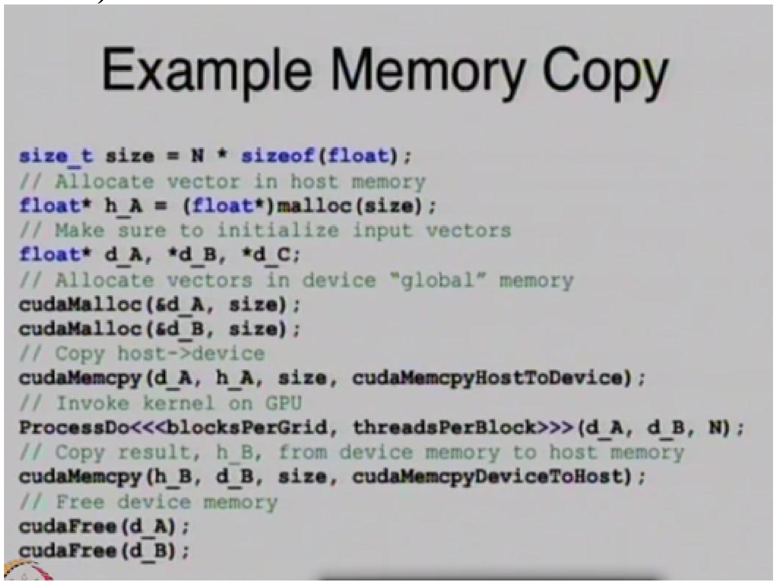
```
TILE_WIDTH = 64;
float *Md, *M;
int size = TILE_WIDTH * TILE_WIDTH * sizeof(float);

cudaMalloc((void**)&Md, size);
...
cudaFree(Md);
```

Courtesy Kirk &

Let us look up a couple of examples that will make this much clearer. In this case we are so in MALLOC what you do. You say in MALLOC some size and it turns to a pointer ok. CUDA MALLOC is slightly different in that. The pointer that turns to you is taken as one of the parameters so two CUDA MALLOC. You send size even to MALLOC as well as the host pointer the address of the host pointer.

Because some value will get rid in that and later on you can and you basically think of it as a handle. It is on a real address that means something on the CPU. It's some handle that you will be using it through CUDA functions. And you cannot simply now say MP is = 63. And then an example of how you might use it is CUDA free ok. Because you cannot directly do anything with it all the usages will be based on the on CUDA functions.

(Refer Slide Time: 07:03)



Example Memory Copy

```
size_t size = N * sizeof(float);
// Allocate vector in host memory
float* h_A = (float*)malloc(size);
// Make sure to initialize input vectors
float* d_A, *d_B, *d_C;
// Allocate vectors in device "global" memory
cudaMalloc(&d_A, size);
cudaMalloc(&d_B, size);
// Copy host->device
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
// Invoke kernel on GPU
ProcessDo<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, N);
// Copy result, h_B, from device memory to host memory
cudaMemcpy(h_B, d_B, size, cudaMemcpyDeviceToHost);
// Free device memory
cudaFree(d_A);
cudaFree(d_B);
```

And here is the basic example of how you might do it. So what you are seeing here focus is not much at the beginning. At some commons that you can read if you declared float pointers on which you will do will be doing MALLOC right. So you have for example CUDA MALLOC D and your score A the center being a float pointer. And then address to a float pointer and asking for some size bites to the MALLOC to be allocated on CUDA.

And so it has two it allocates two arrays of the same size one with the handle D underscore A the other with the handle in D under the score B. So wherever you see D it is a device address so it cannot be directly accessed on the CPU. And then it has a CUDA MEM copy into DA from HA DA is device A HA is host A and it is has how many things to be copied you know we have allocated size that is how much you are expecting to put that and in this CUDA MEM copy style

You have to say which is the direction of copy that is to say something is up more descent methods do not need for you to tell them that. And then there is a call right some process do with some configuration. How many blocks for rid, how many threads for block it takes three parameters the address of the three of the two GPU arrays that you have allocated.

And the size number of alignments you put there and then once the job is done inside the kernel we assume that there are some processing that happens on DA. The results have been stored in DB and we want to now see the result. So we copy the DB back into the host variant of that same variable HB. And then CUDA free D and D so this is basically that the common setup that you will probably be doing for many CUDA application is now so at the beginning of the kernel.

You have got the data in the corresponding arrays which are been said as parameters. And you do processing just like you would do on any CPU meaning that the code is going to look like any CPU code. There are a couple things to keep track of you may or may not actually use it in your in your programs. But there are somewhat more optimized way ways of accessing and initializing on one D arrays right 2 D and 3 D arrays.

And there the usage of this or the need for this will become clearer or bit later when we talk about how the memory accesses. Because ultimately all accesses are parallel liars like when you say read something. You are 16 people saying read something and 16 different addresses which are to be read in parallel as much as possible. And so there is some optimization that can be is done on there and those optimizations can be well taken advantage of instead.

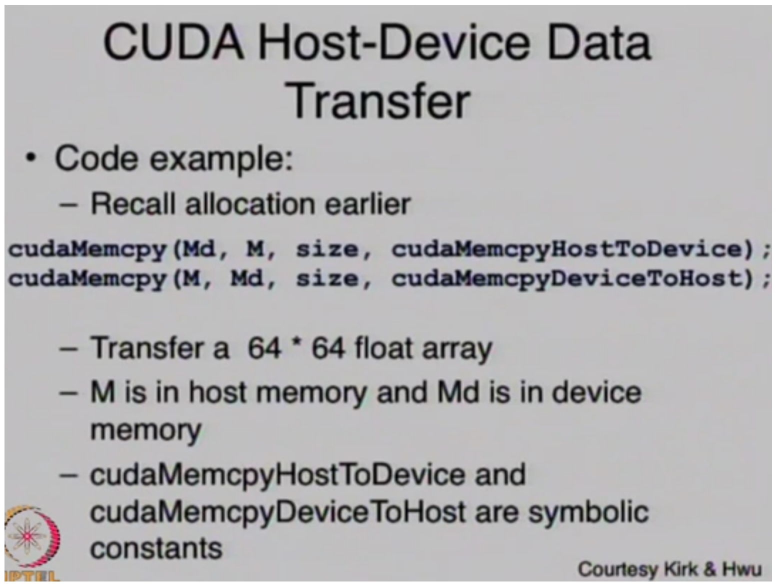
When you are needing two D or three D arrays instead of directly doing CUDA MALLOC you do CUDA MALLOC pitch or CUDA MALLOC three D and there is corresponding MEM copy variant. There is also a notion. There is also a notion of asynchronous transfer which I am going to leave for the timing not talk about it may be. You have already seen the parameters to MEM copy the type of transfer that you can do is not just device to host but also device to device right.

You can do from the host you can instruct that this given array may be replicated. It is not necessary but you can do also host to host what so you are making this call on the host right. So CUDA MEM copy is a CPU function rights its CUDA library right which inturn will probably do something on the CUDA run time. But you making this call on the CPU and that side there are the ways in which you can initiate asynchronous transfers.

Such that when you say CUDA MEM copy there is some time to make that copy happen in the mean time you can do go some may write your code or do something else if other kernel begins. And automatically when the kernel begins or when you reach the launch of kernel.

It would not act physically get called on the GPU until the MEM copy is happen right. Because kernel input depends on the MEM copy it cannot begin until the MEM copy is done so that is synchrony is needed and it will be automatically meant even if you are doing asynchronous transfer.

(Refer Slide Time: 12:53)



CUDA Host-Device Data Transfer

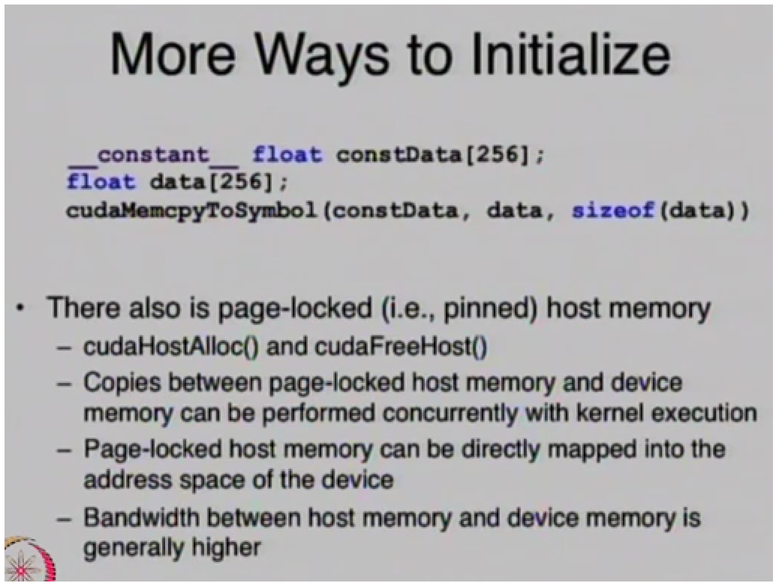
- Code example:
  - Recall allocation earlier

```
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
cudaMemcpy(M, Md, size, cudaMemcpyDeviceToHost);
```

  - Transfer a 64 * 64 float array
  - M is in host memory and Md is in device memory
  - cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost are symbolic constants

Courtesy Kirk & Hwu

(Refer Slide Time: 13:02)



More Ways to Initialize

```
__constant__ float constData[256];
float data[256];
cudaMemcpyToSymbol(constData, data, sizeof(data))
```

- There also is page-locked (i.e., pinned) host memory
  - cudaHostAlloc() and cudaFreeHost()
  - Copies between page-locked host memory and device memory can be performed concurrently with kernel execution
  - Page-locked host memory can be directly mapped into the address space of the device
  - Bandwidth between host memory and device memory is generally higher

There is another example you can MEM copy from host to device and device to host. You can also do simple thing like this and here is variable called COS data. Because you have refixed it with underscore underscore constant underscore under score in the X going to be allocated in constant memory. And you can't assign to it and its address does not mean anything on the CPU.

It does not exist on the CPU but using CUDA functions you can set some values to it. For example you can say CUDA MEM city to symbol and provide a local CPU float data. 56 is the CPU array and you can say there is the size of the data make it is copy and so it is another variant slightly sugar coated variant of CUDA MEM copy. There is also a notion of page locked memory hind memory meaning is not page able it's not going to go the desk.

It will stay on the system memory on the GPU side is no paging this is only on the CPU side. So if you want your copies it is somewhat faster you copy from a page lock memory into GPU not from any general memory. Because then there is no need for the copying urgent to worry about whether that memory address is paste able or not it is actually allocated in the memory or not and all that.

And you take simply the physical memory address and starts copying from there and the functions that you used to allocate on the host side and paged are pint memory or CUDA host ALLOC or CUDA free host. There are there is limit to how much first of all every time you lock given area of memory into physical memory. You are saying that now virtual paging system does not have access to that area of memory right.

So you are effectively reduced your physical memory that for another application circumstances. So they will see some slow down might see some slow down the system also has some limit on how much at given process can lock. So that varies from system to system. But yes if you have a small amount of memory that you have to transfer. Then you make sense to allocate a bit of hind memory keep reusing that for different transfers that you have to sent to GPU and every time you know need to send another block of data to GPU used the hind memory.

Same thing reused same thing like that there is a slightly more optimized way of transferring which is called right combining. And now it is locked as well as uncashed right and so you can instruct the CPU to let go of l one l two usages for this memory. So that part of memory gets deleted from the cash and not cash tax similarly. And as a result there is no snooping and all that needed that data is being modified.

But first system results it is infact useful only when first of all you are not cashing you are basically reading from one into the other. And it is no reading back and forth and which is common right you read from left to right and you copy it to the GPU. So for that kind of applications thus right combining make sense but still it really works faster only if you have a one way transfer from the CPU to the GPU ok.

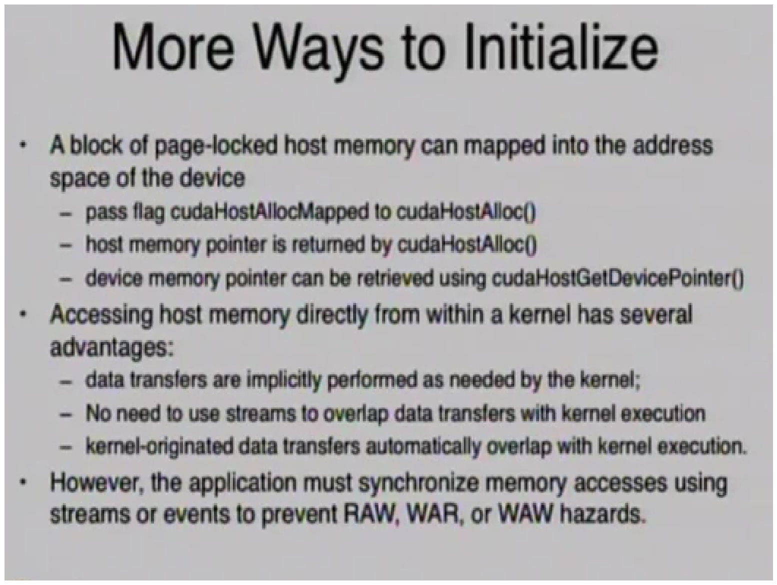
For that if you want to write something into this memory on the CPU side then it is not a good usage meaning that it is not efficient not right now not on the current hardware. There infact have been graphics starts in the past which extended to system memory but it is not available on CUDA the CUDA gives another. Yes, because right now there is no way of extending although hiring CUDA cards. You can go upto high memory sizes page not host memory it can be device yes that is what memory mapping.

We are talking about earlier this is highly doing it you can only map if it is page locked. It is actually the other way around meaning on the CPU you keep writing to it and you do not have to MEM copy anything. And the same address that is the binding of address so that it is going to be more like MPE where there is some local buffer being used to send that address.

Anything you write to that address to the GPU yeah that is very similar if you done in OS seen or used memory map IO and you have it basically. You map an IO into the address that's you memory which actually happens to do an output. So if you want to say print off you simply write something to memory and are going to show upon the screen. If you have map that IO address to memory and it is basically resign ID yes it implicit it implicitly copies in the GPU.

So you would have to know that copy that you do CUDA MEM copy from host memory to device memory. We do pin host memory to device memory we do right combining host right combining pin host memory to device memory. And we can directly do using symbols right you say copy to symbol and typically this is done if you have few variables. But generally it is more efficient to do it in a single go to do CUDA MEM copy by arranging your data to be in one block.

(Refer Slide Time: 20:57)



More Ways to Initialize

- A block of page-locked host memory can mapped into the address space of the device
  - pass flag cudaHostAllocMapped to cudaHostAlloc()
  - host memory pointer is returned by cudaHostAlloc()
  - device memory pointer can be retrieved using cudaHostGetDevicePointer()
- Accessing host memory directly from within a kernel has several advantages:
  - data transfers are implicitly performed as needed by the kernel;
  - No need to use streams to overlap data transfers with kernel execution
  - kernel-originated data transfers automatically overlap with kernel execution.
- However, the application must synchronize memory accesses using streams or events to prevent RAW, WAR, or WAW hazards.

So you also talk about the fact that you can do at this mapping. And so there is some memory here on the host side that becomes approximately for some memory on the device side right. So you write to the host memory on the host CPU and the data appears on device memory on the device CPU on the GPU there also.

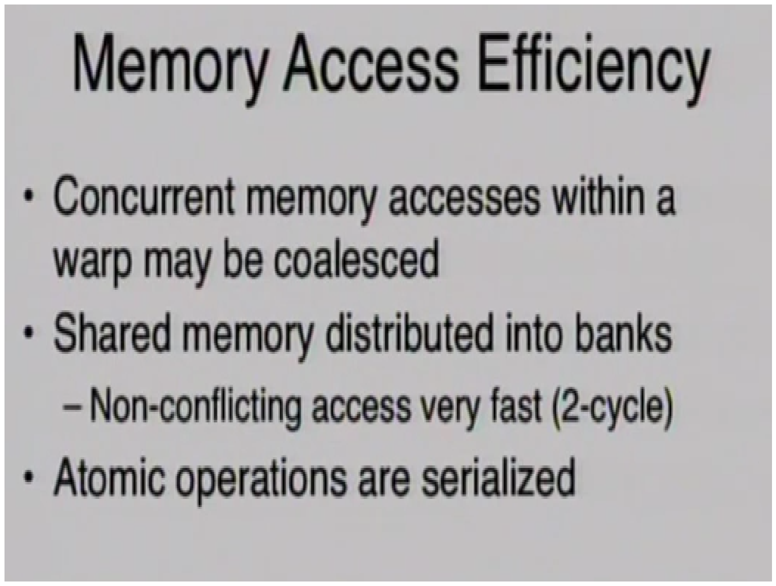
Because you are essentially thinking of the shared memory and you have also thinking of physically shared memory. There is an abstraction that provides you the notion of shared memory right. Although when you write something here it is getting copied to something over there which is then read over there as if there is a common shared memory. And you write something some that thing is being read over there.

And so if you want the synchronization between the writes and the read so that you want to make sure that the writes happen after the read or the writes happen before the read. Then you have to ensure the synchronization ok. So you have to be careful about these read write hazards. We also talked about the fact that and we will look at a couple of slides on streams also.

That when you have several things to do and you want that those things to happen concurrently and you want x to be going on the device. But there is more to be done because device has a lot of compute power then you can do y in parallel right. So you can say do on the device we have talked in terms of kernels. So kernel 1 and kernel 2 and kernel 3 we talked about the fact that we will typically have to do that in streams.

We will setup three streams one stream have the kernel 1 and the other stream will have kernel 2 the third stream will have kernel 3. And they will automatically determine whether there is an upstairs through them all the kernels at the same time ok. Atleast in the case of memory transfers you can overlap memory transfers of multiple streams without synchronization. It can also overlap kernel execution with memory transfers as long as they are through this host mapped memories ok.

(Refer Slide Time: 23:32)



Memory Access Efficiency

- Concurrent memory accesses within a warp may be coalesced
- Shared memory distributed into banks
  - Non-conflicting access very fast (2-cycle)
- Atomic operations are serialized

This is in some stance are precursor in something. We can talk in bit more detail at the end just the high level destruction of CUDA which is where a lot of program designed effort is devoted for CUDA programming. And this is generally should be devoted for any parallel programming which is to recognize that when you read something. And you say read this variable X = Y so you read y and you return into X you recognize that this is happening in parallel at many places right.

Especially when you have a single instruction multiple data been used for that instruction being run on that instruction. You know that all that data pieces are being read that instruction to happen. So it is not just a read of y or a write of X but in read of N array of Y is not necessarily physically continuous array. But set of values and the memory band with is going to be fixed right you can read one bite from the memory at certain speed.

And if you want to read them together so in our example there is a 32 is the size of the work we need 32 sets reading at the same time. I also said that actually half work plans at a time so 16 threats reading at the same time. So I need 16 addresses to read it. So parallel IO is happening now the memory band width may become an issue if there are more reads and typically hardware designers will separate memory into multiple backs. I will say either generates a memory that has 16 ports.

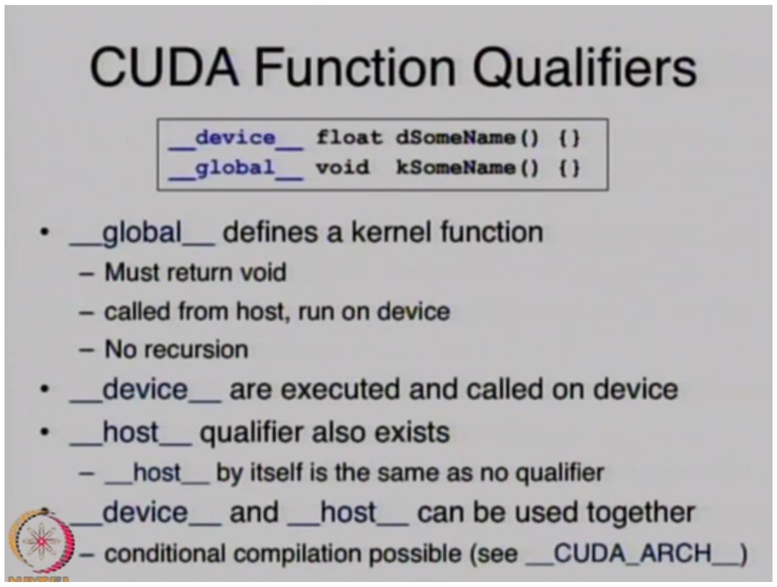
So you can read from 16 different addresses in one clock that is highly expensive to do or you have many banks I am say as long as you read from this bank. You read from this bank you read from this bank you read from that bank there are 16 ports and everybody it is like 16 independent CPU is reading from 16 independent memories in it.

So that happens if it can really do. So we will talk a little bit in more detail about this business of (()) (26:04) meaning that you are doing parallel reads and writes. And what happens to that set of addresses that you have regenerated or what should the pattern or the set of addresses that you are generate in b. So that accesses are faster ok. Typically in CPU programming we do not worry about much of this primarily.

It is a single threaded work for the time so you are reading one thing and that's the only thing you need to worry about. And also the big cashes ahead you would typically be worried about the cash size when you are making the CPU program right. But you were aware of it you can make it more efficient for example if you know what the line size is then we discuss the business of false and true sharing. Infact we saw an example we organize if you are stuck in a one way you keep reading one bite.

And the entire cash line keeps coming in you organize your stuck in your different way. Several reads get collapsed into one cash line and so you read less from the memory. So if you know those things your efficiency will be increased and will look at some of those issues especially in the context of CUDA ok.

(Refer Slide Time: 27:37)



CUDA Function Qualifiers

```
__device__ float dSomeName() {}
__global__ void  kSomeName() {}
```

- __global__ defines a kernel function
  - Must return void
  - called from host, run on device
  - No recursion
- __device__ are executed and called on device
- __host__ qualifier also exists
  - __host__ by itself is the same as no qualifier
- __device__ and __host__ can be used together
  - conditional compilation possible (see __CUDA_ARCH__)

The functions we have talked about are global which are run on the device call from the host functions can be devices which are run on the device call from the device. And function can also be host ok what is it mean to say is the host function normal C function right. It is essentially ignored by the combined sources but you can do though this is device and host it use multiple qualifiers for the same function that’s really an overloading right.

That is short hand for saying I want this to become piled on the CPU as well as for the GPU. So there will be you are basically write in two functions in the same body. And as a matter fact you can even use conditional compilation. So there is some variable called CUDA arch which is set. This will identify that you are running this on CPU if you want few lines of CPU specific code or GPU specific code. In the then you can test for this variable say it is identifying then I must be running on the GPU. So if it is defined the I must be running on CPU.

And your fact is it when it is defined something about the version of the hardware or the architecture not the hardware. But there are I may mention this earlier that there are two classes now 1 point something 1. 0 onwards I think it went upto 1.4 or 3 or 1.3 and then 2.0 onwards and there is a big jump in capabilities of the hardware from 1 point 3 to 2 .0.

Sometimes you may want whether it is older or newer style hardware whether it is subversion within 1 point something or 2 point something. You may not want to do in significant about that.

And as I had said that the once the hardware are accessible in the lab are all 1. X for you and so you would be capable about using the new features no it is different it is slightly different.

Device host means that this function is being compiled by the CPU ok and another part of this function is being made by a VCC right. And compilers the one that generates PTX ok no such things happen for global. It is still being generated for the CPU for the GPU at with DTX assembly being generated from it.

But there is a handle for it being generated. For the CPU to make the call so that call can be generated somewhere on the host function you will be making a call to this global function. So that call is going to get translated on the CPU or translated for the CPU and it only needs to figure out what address to send to the GPU to make that call on the GPU. Eventually that call is made on the GPU right but there is some packaging that needs to happen. So that the GPU can tell that CPU to start execute that no not exactly right.

When you are making the function that is the device host you are making two functions ok. One is the entire function with host your reboot one is again the copy of the entire function with the device keyboard reboot ok. It is not the same thing one thing I want to do on CPU and one thing on GPU that is something that you make you had the choice of doing but you do not have to write.

For example sometimes when you test something global absolutely does not give you this choice because global will only execute on the GPU. So that definitely it does not make chance you cannot make a function call to the device function from the CPU or a device host function. Yeah, because you had a version for the function of the GPU right and the GPU that the GPU name are not even visible and even for the global function.

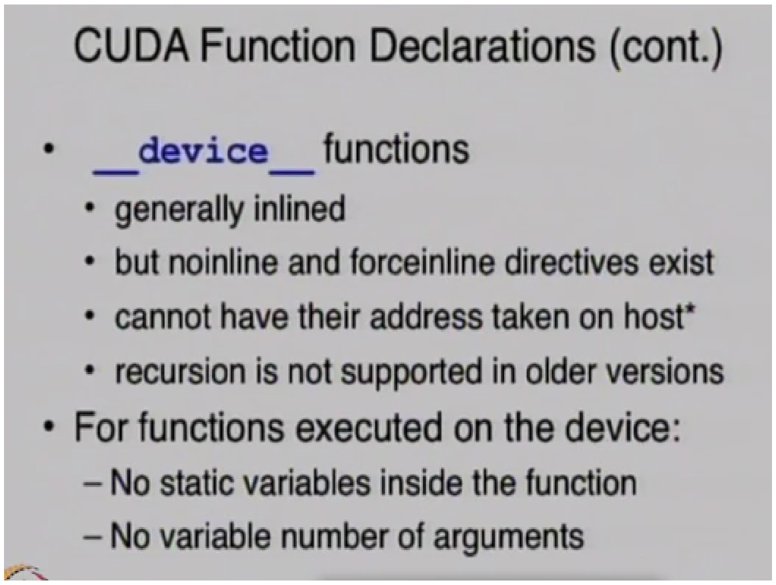
It that name may not be visible right handle for it which had that name inbuilt. Somewhere host bite cell does not mean very much they are very different. Let me repeat the host device right so the host device is like writing two functions one host one device global is not writing two functions global is still a device function. There is no you cannot execute the global function on the host not strictly right. Because CPU generate some command so that it may be entered on the GPU.

But you are not directly calling that function on the CPU right you are not taking the address of that function when putting on in program counter and jumping to it that can be done. I do not think so but it could have been possible it is no real reason you it must be disappointed you would so that for any global call you have to specify a configuration.

That is how many threads to ram and when you do not specify that configuration it could assume itself it is a host call right. So it could be done I do not think it is and it might happen to do with this configuration right. Thread X variable thread ID yeah thread ID is a variable which is predefined only on the GPU side does not exist on the CPU side right. So if it were a global host a device function and you wanted to do something based on thread ID it would have to be protected inside the say would have inside host there would no thread ID.

Yes, the global functions can call other device function they would not call other device function ok.

(Refer Slide Time: 37:00)



CUDA Function Declarations (cont.)

- `__device__` functions
  - generally inlined
  - but noinline and forceinline directives exist
  - cannot have their address taken on host*
  - recursion is not supported in older versions
- For functions executed on the device:
  - No static variables inside the function
  - No variable number of arguments

I should probably explain that only in 2 point X in 1 point X recursion is not allowed ok. And that you see not in this slide some hint about that here which is the device functions are generally inlined ok. Aggressively inlined especially in 1 point X in 2 point X it is somewhat similar to GCC. Might to and you do have where are these underscore underscore no inline underscore underscore directs.

You can use you say this function should be in inline or their force inline that say if you really want inline just like you do in CPU plus plus. There is a force inline function the keyword id different even if you force inline. It sounds like you are instructing the compiler to do certain thing but it is really a suggested. So there are situations in which your force inline will be ignored and they are more likely to be ignored.

Wait no inline is more likely to be ignored in 1 dot X because it really wants to do aggressively inline. You just not very good in this the stag size on 2 dot X is still limited. Even that the total amount of memory is limited but enough that think I put on that slide. I could not find what the stag sizes in the two versions are. As you would expect you cannot take and a percent of the device function on the host if it is 1 dot X you cannot take a device function at all.

The address of the function at all if it is 2 dots X you can take this address but only on the device right. I would not make to take this address on the host if it is a device it does not exist on the host ok. Here it is recursion is not supported in (()) (39:25) but I thought it is more about that yes older version means 1 dot X global function is separate thing. It is not a regular function so you cannot nobody said neither the old nor the new support recursion in global functions device function.

Yeah global function calls something that is that can execute recursion right. Again typically you learn out of this sooner than you expect to run out of stats space on the CPU that stat space is somewhat smaller even on the newer versions. For all functions that executed on the device like device function or a global function you cannot have static variables inside. Although the specific types of variable behave as their static although you cannot declare anything to be static.

You can declare something to be resident in shared memory. For example it automatically becomes static and the shared is there if you reenter it still there. It is kind of like a global if you think in terms of C scopes variable number of arguments are not supported on either new or the older versions. I will get to how you call the kernels but I think we need to stop we have run longer than half an hour. So let me stop here and change the slide number.

Ok let us continue we are just talking about the different types of functions are there any questions on their restrictions. Start on the newer no not on the shared memory I believe it keeps the stat in either the constant memory or the global memory but not the shared memory ok infact constant memory was practically unused in the 1 dot X. Even the parameters that you send to functions where through shared memory and it did not have stats to store anything in memory but the newer ones use constant memory bit more.

So either the stat is in constant memory or in the global memory from your program actually that is right. It a hardware limitation not the software limitation so it will must be in the global sorry so there is two issues right. You cannot change because the software module does not allow you to change it or you cannot change because the hardware does not allow you to change right. So I think there is some limitation I think hardware does allow you to change it.

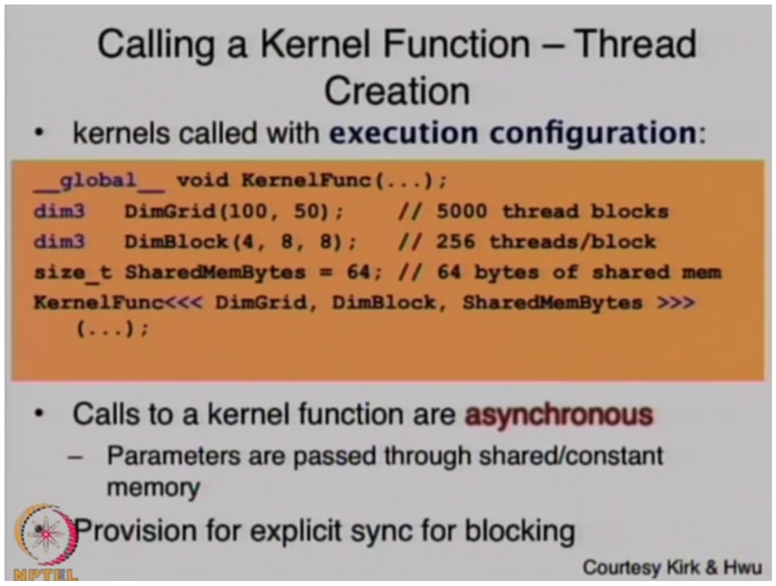
But it is extremely inefficient to do so and so that made them because it is efficient inefficient on the hardware. I would not be a good place to put this exactly yeah so global memory is the mean memory of the GPU. You do not refer to the stock almost always right. So you will bring it back into register and so at the beginning and at the end of the function call.

No it tries to fit in as many local variables as possible on the register itself it depends there are 16 kilobytes of registers for each assembly. But they are allocated to multiple blocks and so you will get if you get 32 registers for your program then you are doing quite well. Probably more like 16 is what you would expect again it depends totally on your kernel right. Your kernel doesn't or here there is a trade of the kernel and the block size.

If your kernel requires many registers and you can even control you can instruct the compiler to use more registers as you want everything to be in register. But then you must make sure that your kernels had fewer threads that are all concurrent. No shared keyboard is not used for function yes on the shared memory yes so it is possible to get the run time error in recursion when you run out of shred memory it is not work 1 dot X.

Yeah 1 dot X you would not have run compiled time ok. So kernels which are the global functions get call using a configuration ok.

(Refer Slide Time: 46:05)



Calling a Kernel Function – Thread Creation

- kernels called with **execution configuration**:

```
__global__ void KernelFunc(...);
dim3   DimGrid(100, 50);    // 5000 thread blocks
dim3   DimBlock(4, 8, 8);   // 256 threads/block
size_t SharedMemBytes = 64; // 64 bytes of shared mem
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>
   (...);
```

- Calls to a kernel function are asynchronous
  - Parameters are passed through shared/constant memory
- Provision for explicit sync for blocking

Courtesy Kirk & Hwu

So if its regular host functions you will say the kernel the parameters list right. Whatever is in the parenthesis and also I have written dot dot dot I am not meaning a variable number argument but this is a call. So it clear it is whatever the arguments are it takes three parameters to fully specify a configuration. And you have to have atleast one of them the other two are optional may be two of them I think.

You have to provide both dimension of the grid which is how many blocks you are trying to run and the dimensions of the block which is how many threads per block ok. Each one of them can be 1D, 2D no dim grade is 1D, 2D SIM block is 1D, 2D or 3D. And because it uses in constructors internally either you can had something of type dim three for the block size or even type two in which case the third dimension does not exist or even in it. Usually the second and third dimension does not exist ok.

And the third there is a third argument which is the size of the shared memory that this kernel is going to use ok. Including any recursion this is not the shared declaration that you have you do not want to count up how many all the times you have used the shared memory. This is on top of it and this is I am going to declare some variables which I want to reside in the shared memory. I want you to give me a block of 64 bites for me to use as scrod space ok.

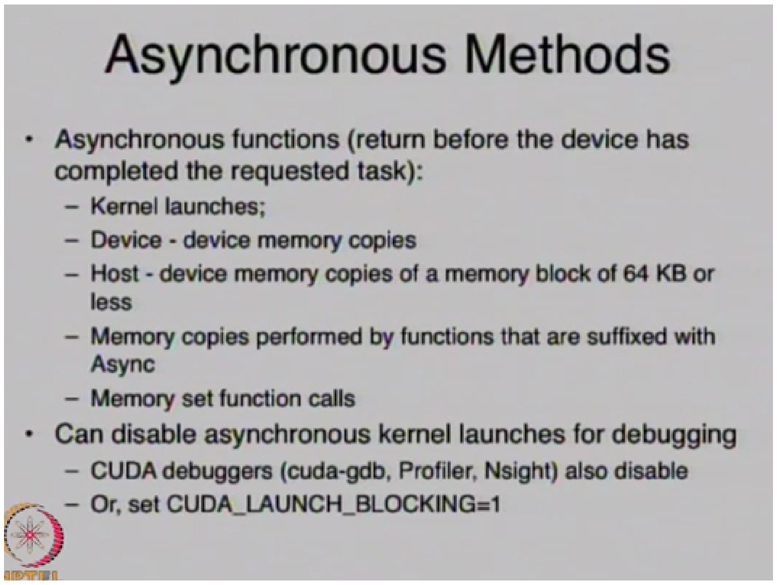
So it is an best essentially it is an array of whatever size you are saying it is it must be determined it can be determined at run time. And when this representative of this kernel invocation will get called on the CPU it is going to figure out how many hosts how many shared memory bites are required by the kernel. And that a requisite amount of control commands will get sent to the GPU to allocate that many shared memory elements ok.

I will tell you shortly how you would the shared memory this is separate edition to all the variables. You will declare you have to declare a variable so you can access this also but that is done using extend rather than the once you put inside the function. The call when you when make this configuration call it is going to be return as soon as it is able to figure out what is the configuration what is the command that is to be sent.

All of the data is sent to the GPU and then it returns right. So it is not going to wait here for the device to say that I have completed the execution of this function. However there are other things you may do later like calling another kernel or making a non overlap able memory transfer to device. Its going too stuck there so that cannot be proceed until this thing is completed ok. Yes, that is on the GPU side there is expenses but there exists and the parameters I think already mentioned are past using shared memory on 1 dot X.

And constant memory on two dot x as I said there exist provision for expressive syncing. So you can say that do not return to me until you have finished execution of this kernel and you would do that when you debugging. You would not necessarily want to do that in code that you are publishing right. In code that you want to be running you want to major the performance of and as a matter of attitude do CUDA GB or one of the other profilers they will also decide it for you ok.

(Refer Slide Time: 51:44)


Asynchronous Methods

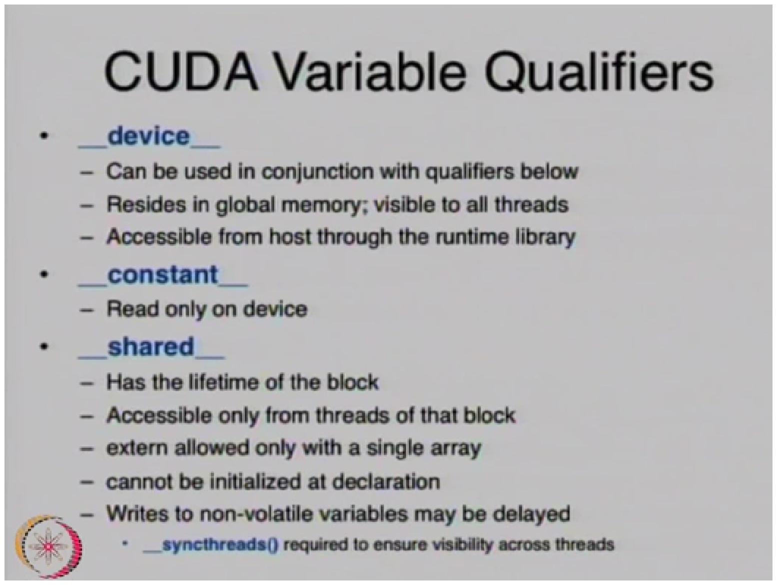
- Asynchronous functions (return before the device has completed the requested task):
  - Kernel launches;
  - Device - device memory copies
  - Host - device memory copies of a memory block of 64 KB or less
  - Memory copies performed by functions that are suffixed with Async
  - Memory set function calls
- Can disable asynchronous kernel launches for debugging
  - CUDA debuggers (cuda-gdb, Profiler, Nsight) also disable
  - Or, set CUDA_LAUNCH_BLOCKING=1

As I said the shared memory will talk about short time there are exist asynchronous versions or some function calls which make when the synchronous calls asynchronous ok. So you can say CUDA MEM copy sync if you want to return right. It does not mean that to finish as soon as you call it but it will return before it finishes ok. There are these five function calls odd events which are returning as soon as you as soon as it figures out what it has to do without actually completing the task can do launches.

We just talked about that when you on the host launch a kernel it will return as soon as it is figured out what to do you can do device to device memory copies ok. You can do host to device memory copies of limited size. And because there is enough buffer to make that happen it will return right away right if you to bigger chunk of data to get transferred then you have to wait for it to complete.

And if it is a sync as I said there are many of these functions have an underscore a sync version. So if you are calling a sync version then they will be done. And there is MEM set that you can do on the CUDA device which will also return without having already set memory ok. And I have assigned you can disable all of these asynchronous calls by setting an element variable called CUDA launch blocking and again you should not do that unless you just test debugging.

(Refer Slide Time: 53:42)



CUDA Variable Qualifiers

- __device__
  - Can be used in conjunction with qualifiers below
  - Resides in global memory; visible to all threads
  - Accessible from host through the runtime library
- __constant__
  - Read only on device
- __shared__
  - Has the lifetime of the block
  - Accessible only from threads of that block
  - extern allowed only with a single array
  - cannot be initialized at declaration
  - Writes to non-volatile variables may be delayed
    - __syncthreads() required to ensure visibility across threads

So these are the variable qualifiers as supposed to the function qualifier. We just looking at so variables you can say are device constant or shared ok there also text variable. We will stay away from that because they have a very graphic centric understanding of the memory should be accessed right. You can access memory address 2.5 instead of getting the data upto 2 and 3 get the average of the data 2 and data 3. So that the all of that is required in graphics and sop it is focused on graphics.

I do not expect if you are doing a non graphics application. I do not expect you to need that kind of access somewhat going to discuss the texture memory infact their organization is there a different you declare arise in a certain fashion with certain 2D, 3D patterns. And all that so device memory is one that is going to be on the device in the global memory.

And it can be accessed from the host and because you said CUDA get symbol and you can get an address to that device function and you can send something to that symbol. So there are provisions for you to be able to say something on the GPU. Set of variable exist on the device from the host by taking its address in an indirect way. There is constant memory which has its own path complete path to get from did I say constant is readable but highly inefficient.

I think I was talking about texture constant may not be readable at all even in the hardware sorry writable at all. Yeah ofcourse it is readable otherwise going to be much used even if you are able to write it then there are shared memory which will be allocated in the shared space. It is visible only within a block right. So if you got multiple blocks in a grid they all have their own versions of the shared variable but there is only one instance of that shared variable for the entire block.

So every thread is not going to get an instance for this variable ok. They can read it same time they can read it at different time if you want some shared access to be controlled by syncing. And all that is possible remember that not the entire block runs in lock step right. So different walks of block will be at different parts of their code so you have if you want to measure.

They all reach certain point before writing to that shared memory or before accessing that shared memory. We have to ensure that using sync threads and this is concurrent right but random that is everything is trying to write to that. One of them will succeed in 1 dot X in 2 dot X through constant memory not that matters it just means you have less of the shared memory available for the rest half.

Shared memory is specific to a block right so for every block whenever the system says here is one more block you going to get one instance of the shared memory allocated created for that block. That will have be arguments so it is not exactly the same way right. The arguments are it is like the system taking away the part of the shared memory for its usage. So it is not like you have the access to ensure that the arguments are in shared memory.

It will not run right it will if it can figure it out that compiled time it would not it will give you a compiler array. There are situations especially in 2 dot X which has generalizing little bit more that you would not be able to know at compiled time. But we will get a run time so the shared variable will have the lifetime of a block. They are again as I said static right so u call one device function again and again that the device function can assume that what I wrote in the shared memory is still there.

When it comes to the device function next time it is a local shared variable but as you declared it to be static. They yes it is no it is also for global functions right. So it as long as so when you when you call this global function certain n umber of blocks need to executive and that shared memory is reserved for that block. Until whenever the block is created that shared memory is going to get tied to it. Nobody else can use that to each block right. Certain portion of the shared memory get tied to a given block and until that kernel is done.

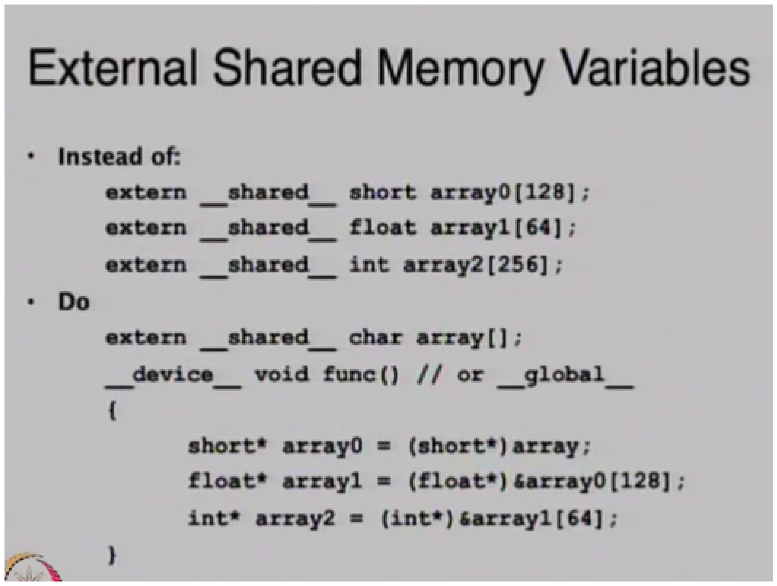
That blocks owns that shared memory and nobody else will write to it on the local device functions will the local global variables meaning inside the device variables are not static. So they will go away but if its global scope device variable then they will stay across kernel calls ok. Because global function global variables or global memory does not get rehearsed right it is that variable is visible is all the functions within a file.

So whichever calls it cease the same address extends are not allowed for any of these device variables except for that block for shared memory that you had reserved right. When you said in this kernel in location I need 64 bites of shared memory or whatever that is declared explicitly as extend ok. I will I will show you an example shortly this volatile keyword just like in open MP cash can play have a when you have a shared memory. It can go to the cash but cannot reach the shared memory.

Same thing can happen this is not an issue on one dot x because it does not have a cash but on 2 dot X because it has a real cash. If you want to make sure that has got to shared memory every time you have written it either you have to sync or you have to declare it to be volatile. Then it will every time it will write to the memory ok. So the some restrictions listed here with five scopes only there is no external out struck a union not allowed in formal parameters or on local variables.

Again this is probably going to just a software restriction going to happen someday. Shared cannot be used on local variables on the host right host function that you have taken declared to be host function. Obviously as I mentioned constant and shared anything that has a constant and shared identifier or qualifier is going to behave as a it is static ok so ok.

(Refer Slide Time: 1:03:58)



External Shared Memory Variables

- **Instead of:**

```
extern __shared__ short array0[128];
extern __shared__ float array1[64];
extern __shared__ int array2[256];
```

- **Do**

```
extern __shared__ char array[];
__device__ void func() // or __global__
{
      short* array0 = (short*)array;
      float* array1 = (float*)&array0[128];
      int* array2 = (int*)&array1[64];
}
```

This is where the shared memory the block that you have said I want to use how you use it right. So you declare one variable of type extend which is outside the function ok. So if you think in terms of C so what you think in the bottom they do part let us focus on that first. You are basically saying extend shared cared array some size you are not telling the size at this point. And you are not actually declaring it anywhere in your program right typically that extend will be declared somewhere else some other file have declared it.

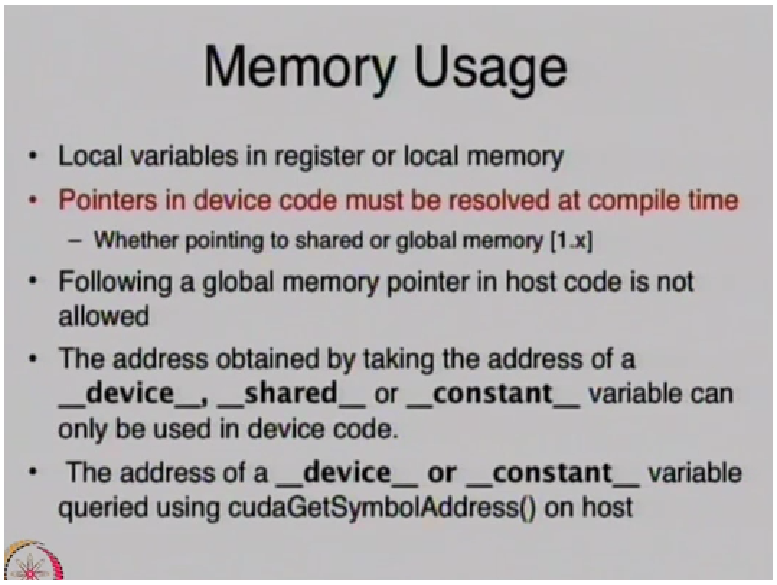
So that somewhere else is automatic right this going to happen by magic through MCC. You simply declare one extend shred care or whatever type you want to make it an undetermined size. The size is going too filled in at the configuration time right. So when you make that kernel launch it is as if this shared care array has been called with that size ok. You make another launch the size going to change and are determined at one time. And that is why you are not allowed to have multiple of these right.

You want you are instead of the compiler taking ion the responsibility of adding up call together making fit certain size at known at the configuration launch time. You simply declare one variable of type shared which is extended and inside any device function. You can if you want to allocate separate variable separate blocks inside that you can make point as referring to different offsets in this global block of shared memory global using loosely ok.

Block that is single block for the block of threads there will be one such declaration then it is a regular C. It is on the CPU you can declare a device function but I mean device variable. But it will not go in the shared memory it will go in the global memory global memory yeah. You can declare many device things and they all be allocated in global memory it is only for shared memory that this thing has been done.

Because its declaration is happening automatically right. You are not actually declaring it is of certain size or somewhere else.

(Refer Slide Time: 1:07:24)



Memory Usage

- Local variables in register or local memory
- Pointers in device code must be resolved at compile time
  - Whether pointing to shared or global memory [1.x]
- Following a global memory pointer in host code is not allowed
- The address obtained by taking the address of a **__device__**, **__shared__** or **__constant__** variable can only be used in device code.
- The address of a **__device__ or __constant__** variable queried using cudaGetSymbolAddress() on host

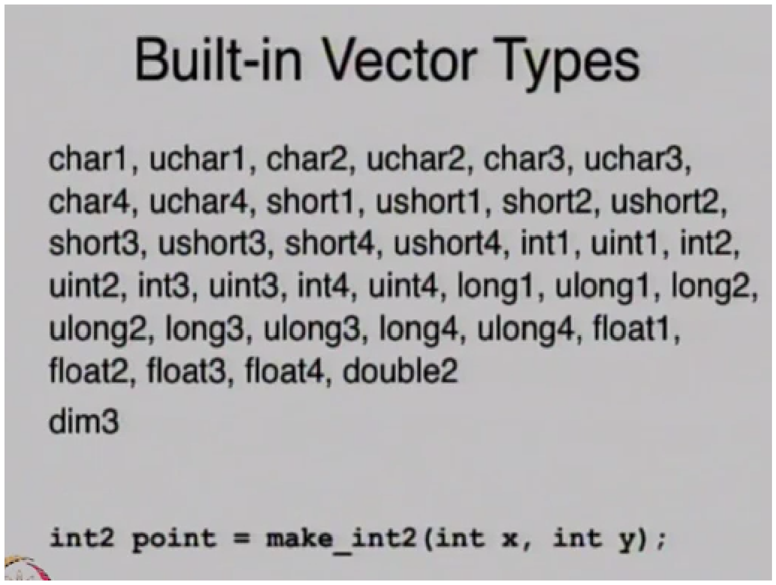
The local variables are tempted to be in register all the time if they don't fit in the register allocation. Then they are going to go in global memory ok which is called thread private or thread local memory pointers must be resolved at compiled time. But there is only one point x ok what that says is that it must know at compiled time.

Whether even reference is going to point to a memory in the device section the global memory section or in the shred memory section ok there is no size restriction on 2 dot X. Of course you would not take a pointer that is on the device and follow it be refers on the host right. So that again that would not many make sense does not exist on the host. But on 1 dot X you have to know whether this is a device reference or a global memory reference sorry the device meaning global memory reference or a shared memory reference.

So that is somewhat high level description of the pieces right. How you declare variables, how you declare functions, how do you call functions, what do you think about this design device variable for host you can set a value to it. Yeah meaning that you are asking that should be initializing to certain thing. You no it is not like you are setting that some variable to some function.

You are going to get symbol address and you are going to set a value using the function called CUDA something. Yeah it is not going to turn into a regular c assignment and it has all this types. Unless few more it has vectors types of vectors defined predefined it also has half 16 floats. That to share some memory it defines however does this does not have the half.

(Refer Slide Time: 1:10:38)



Built-in Vector Types

char1, uchar1, char2, uchar2, char3, uchar3, char4, uchar4, short1, ushort1, short2, ushort2, short3, ushort3, short4, ushort4, int1, uint1, int2, uint2, int3, uint3, int4, uint4, long1, ulong1, long2, ulong2, long3, ulong3, long4, ulong4, float1, float2, float3, float4, double2

dim3

```
int2 point = make_int2(int x, int y);
```

What would you change you now know the architecture right. You seen that basically there is some number of things that have to run in lock step their multiple of these and then lock step things have to share some memory. And then there is a global memory that is they across right it is a kind of hierarchy it will design.

So what programming model would you had for it other than saying that I am going to write single threaded single processor CPU code and I want you to figure out how to run it on CUDA right. That would be a good thing so it is extremely challenging and does not exist here so how you would make a programming model that HH is aware of the architecture. And is something that allows the programmer to explicitly express the parallelism so no parallelism is being inferred which has been the theme right.

The open MP or MPR before is nothing all been inferred you could have done it using combination of open MP and MPR right. You would say here is the bunch of shared memory that refers to the shared memory. And then MPI like all that refers to the other that is also shared memory of global memory is also shared. But it is not it has a global scope of sharing where there is a local scope of sharing right or may be at two level open MP. Ultimately it's a shared memory so is it basically open MP right. Here CPU is another core is not it.

So and infact you have talking about CPU having one core it really has multiple cores typically so few cores on CPU and few cores on GPU. So now you have three levels right there are these two bulks of things that doing MPI between them. Because they would not physically share the memory then within this bulk there are two levels threads. That share that memory global memory and their sub threads that share the shared memory there is there is nothing new there right.

It is somebody is doing that translation for you at the day all these memories have a separate address right. In CUDA one dot x there address was directly put in to the instructions. So and instruction for loading from shared memory was different from an instruction the up code was different from an instruction loading from global memory right. And that got regularized or generalized to saying I do not want to put in the up code I just want to say read from memory you do right. You just use certain section of the bids to say that this if your addresses.

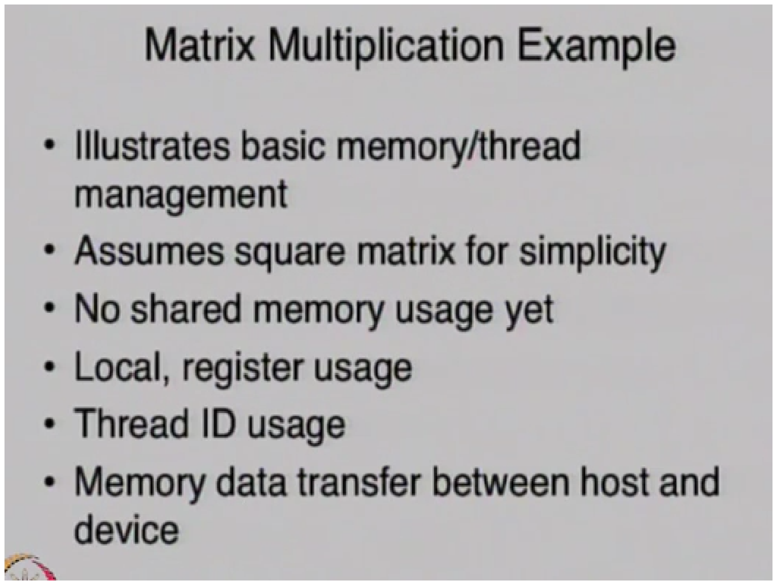
First rebids or 0000 in that CPU and 001 in that CPU if 010 and it is the third CPU if it is 111 then it is the first GPU. You can do that and that is what was done at the GPU level in 2 dot X right where it said instead of putting things in your court we all say if the first bid of the address or first some bids of the address say something.

Then you are in global memory otherwise you are in shred memory the instruction is same its read from memory and then the hardware simply looks at the address to figure out whether this is referring to the global memory address or to a local memory address. And then strip of the correct bids to get to the offset within the within the local memory or within the global memory or within the shared memory. The same thing that has generalized on multiple GPU side ok so at some level this is this is open mp only in real difference is that in open mp loop was the mechanism by which you said do this in parallel right.

You said for I = 0 to parallel for I = something you do this and there is for generalized sections in all those things but PV the core of the matter was that you express parallelism using a loop here. You express parallelism by writing a function and putting the loop in the configuration ok and now that there are multiple levels of memory of sharing. That somehow has to play a role somewhere you can you can do that there also in the open MP also that is what the core constructs that is on the default construct.

But you can create one which focuses of another right. So this guy continue and this guy is on now several parallel things that they done together so there are many ways in which you can do that. It here it is just that the structure instead of having a loop has turned into a configuration and the nesting where you automatically wrote more and more nested loops. And here the nesting is kind of static it is a grid.

And the grid has certain number of blocks; block has certain number of threads you have to use this static configuration right. You cannot now continue to say that thread is certain number of sub threads at an interactive or run time basis ok. We will look at some code example. **(Refer Slide Time: 1:18:55)**

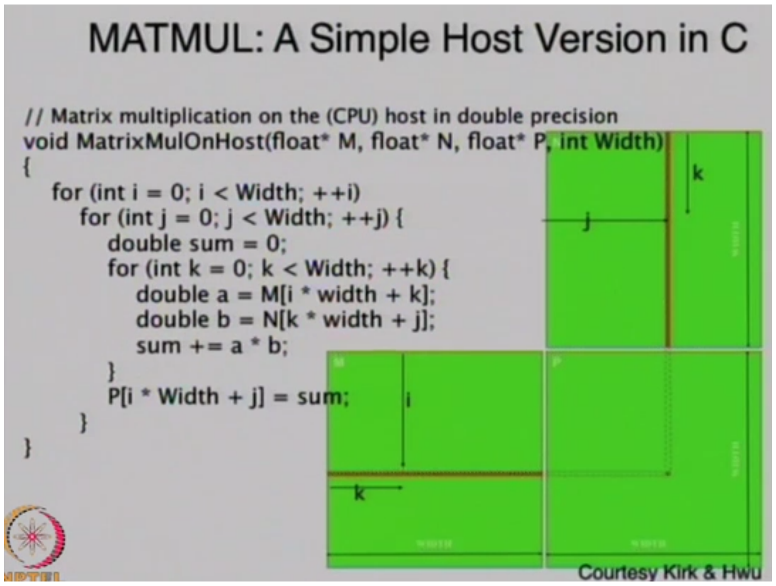


Matrix Multiplication Example

- Illustrates basic memory/thread management
- Assumes square matrix for simplicity
- No shared memory usage yet
- Local, register usage
- Thread ID usage
- Memory data transfer between host and device

And one that is commonly provided in all CUDA manuals and all loops and which you should probably took look at. I want to do matrix multiplication so I got some matrix A and some matrix b and I am going to take one row here one column here. Apply the products and then sum it together you want to do this today it will not actually help you much with the assignment.

(Refer Slide Time: 1:19:47)



MATMUL: A Simple Host Version in C

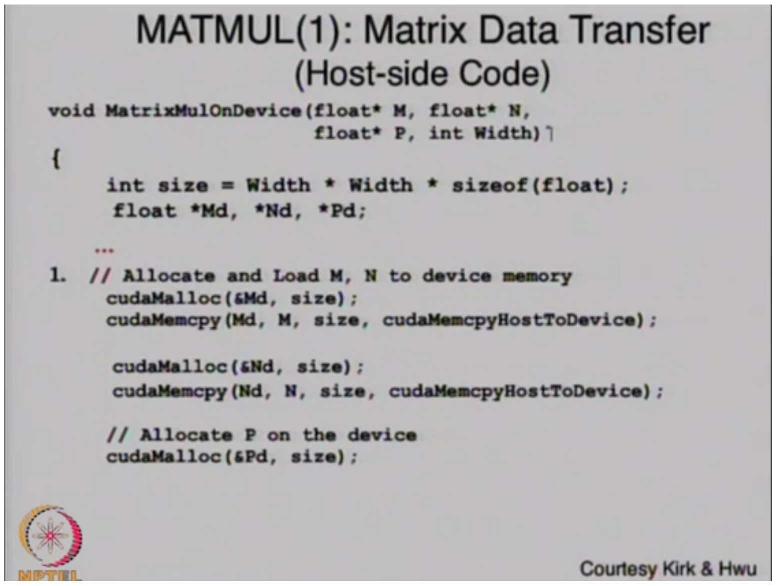
```
// Matrix multiplication on the (CPU) host in double precision
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
   for (int i = 0; i < Width; ++i)
      for (int j = 0; j < Width; ++j) {
         double sum = 0;
         for (int k = 0; k < Width; ++k) {
            double a = M[i * width + k];
            double b = N[k * width + j];
            sum += a * b;
         }
         P[i * Width + j] = sum;
      }
}
```

Courtesy Kirk & Hwu

Because I am not going to do any reduction stuff here but let me jump to something that may be helpful atleast in the organization. So this is what you see here is typical matrix multiplication c code right for I = 0 to and this square matrix this is just to make things center 0 to it J = 0. It then you move K here and here and so you have to figure out whether you at what level you want to parallelize right. You can say I am going to a simple way is every result element is independent of other results.

So let us compute the result in parallel so I am got in biometrics and square thread each start producing one result right. You can go for that you can say let us make a cube threads each taking one element here one element from here multiplying it and then doing reduction later on right. This is the simpler one and to set this up.

(Refer Slide Time: 1:20:52)



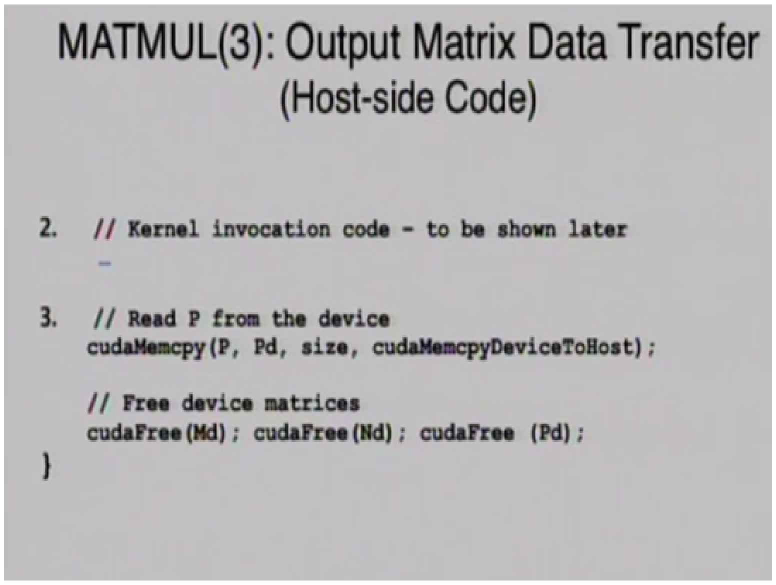
MATMUL(1): Matrix Data Transfer (Host-side Code)

```
void MatrixMulOnDevice(float* M, float* N,
                       float* P, int Width)
{
    int size = Width * Width * sizeof(float);
    float *Md, *Nd, *Pd;
    ...
1.  // Allocate and Load M, N to device memory
    cudaMalloc(&Md, size);
    cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);

    cudaMalloc(&Nd, size);
    cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

    // Allocate P on the device
    cudaMalloc(&Pd, size);
```

Courtesy Kirk & Hwu

You would this is still a host function although it is called matrix on device it will allocate some amount of memory on CUDA copy the inputs to the device.

(Refer Slide Time: 1:21:16)

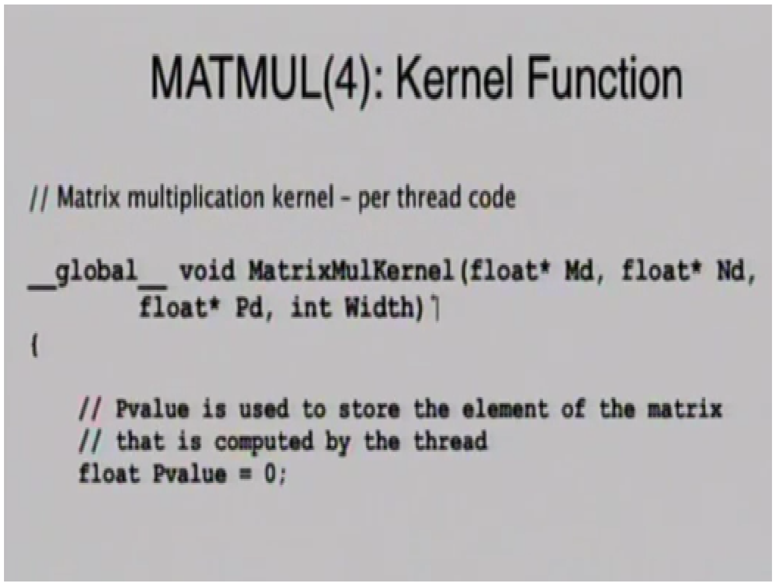


MATMUL(3): Output Matrix Data Transfer (Host-side Code)

```
2.  // Kernel invocation code - to be shown later

3.  // Read P from the device
    cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);

    // Free device matrices
    cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
}
```

And then do what is written in number two in web the kernel show you that also and then what the kernel is dine copy of the result back and free the memories that you have allocated.

(Refer Slide Time: 1:21:35)



MATMUL(4): Kernel Function

```
// Matrix multiplication kernel – per thread code

__global__ void MatrixMulKernel(float* Md, float* Nd,
        float* Pd, int Width)
{

    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;
```

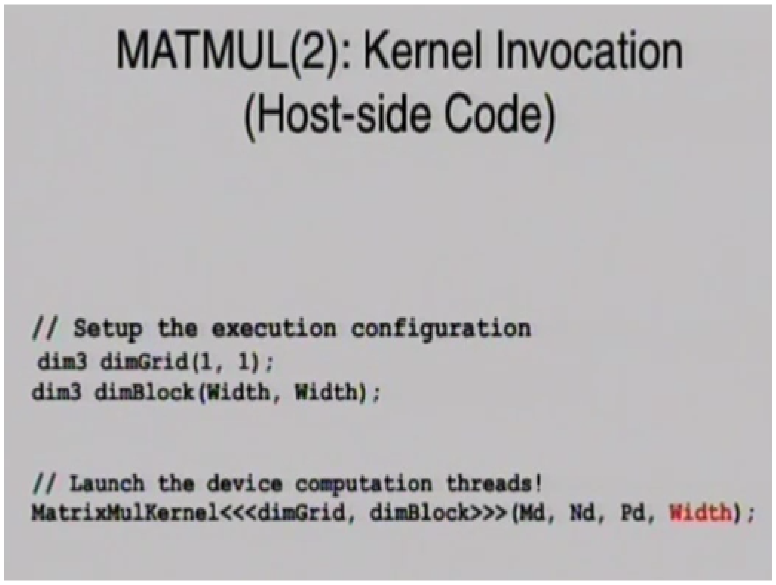
The kernel itself would I think I should restructure this it is become little hard to read I think breaking it into different slides. So here is the bulk of it so we have a kernel and we need to take a set if elements from here set of elements from here and figure out where to get things from depend on your thread id right.

I am not going to go through the derivation of your position of which element you are taking the product of and a then you keep taking products and keep adding to the partial system because here you are producing one result per thread one output might trick results per thread. And so this is all that there is in the kernel and the invocation will simply say run the kernel with I do not want any shared memory in this case everything is happened directly on the global memory.

And you would want to use shared memory because you are taking these products and adding it to the sum in this case that some probably is on the register. So it does not matter but if you wanted to reuse something over and over you may have wanted to you shared memory and so in the invocation. You say here is the number of grids I want to run which is one by one right.

Everything is in a single block the block I want to run is n cross n right I J will give me so that my block my index within the block are I and J index will tell me which result row column I have to produce. And then send these matrices ok the matrices that you are sending or the parameters that you are sending are.

(Refer Slide Time: 1:23:57)



MATMUL(2): Kernel Invocation (Host-side Code)

```
// Setup the execution configuration
dim3 dimGrid(1, 1);
dim3 dimBlock(Width, Width);


// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

What CUDA pointers right the handles that when you did CUDA MALLOC you got up a pointer to that or s handle to that ok? So you send the three pointers and an additional argument which is what is the size it will ok. So let me stop there.