

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

Module No # 05
Lecture No # 24
CUDA (CONTD...)

(Refer Slide Time: 00:33)


```
MATMUL: Matrix Data Transfer

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);

    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);
}


Courtesy Kirk & Hwu
```

So then let us begin I had very quickly gone through the matrix multiplication and I have tried to now put the slides the code as much as possible on one slide as a result of which it has become a little smaller but I think I can read it from here so you guys should be younger should be able to read from where you are okay. So we decided that in that this is not the only way in fact we will discuss what other alternatives.

We might have chosen you want in this case one thread to compute one element of the resulting product matrix okay and just to make sure we understand the parameter we are doing there is float star N float star M from the two matrices each is width cross width and we take M times M and produce the result in P. So this host code host function is being passed these parameters and to win enough space to store the result of these two input matrices product and the size of the matrices along with the matrices.

And we decided that for every element of the output one thread is going to create the output this will not work when the number of elements that you need to produce is more than the number of threads you can keep in a block then you are multiplying to big matrices bigger matrices then you got to do something else all right. So that is something else discuss after we take the simpler case so this simple we have one entire row of M one entire column of N being read by one thread okay.

The same we realize the same row will also being read by other threads and the same column will also being read by some other threads and so there may be additional reading the are potentially doing here okay. But again to keep things simple we have allocated space on the device those are the variable pointed to by MD, ND and DMPD. We do CUDA malloc on each of them to get enough space such that input matrices can be passed on from host and the output matrices can be produced on the device and then pass back on.

So CUDA malloc create that matrix CUDA malloc is something you have again to do on the host so that the now looking is happening on the device and you would do this when you need this pointer on the host right. If you want to do regular malloc on the device that pointer is not going to accessible on the host. Then CUDA MEM copy takes your float pointer from where you are copying.

For the example in the first MEM copy call that M the device handle is not a real pointer device handle that has been returned in the float pointer that you are refer it has an MD and the size how many elements wants to copy. The size should be calculated from the width I am doing where oh yeah right it is calculated from the width in the number of bytes. Then you call the kernel this is the host codes is there is the number 2 which is kernel in the location which will come later.

And the kernel itself which will come later and at the end of it you MEM copy the result right if some reason the two variables MD and ND have been modified on the device which is on this device it would not be then and you wanted those modification back then you would copy them back also and the CUDA okay.

(Refer Slide Time: 05:19)

MATMUL: Kernel Function

```
// Matrix multiplication kernel - per thread code
__global__ void MatrixMulKernel(float* Md, float* Nd,
                                float* Pd, int Width) {
    // Pvalue stores the matrix element computed by the thread
    float Pvalue = 0;
    for (int k = 0; k < Width; ++k) {
        float Melement = Md[threadIdx.y*Width+k];
        float Nelement = Nd[k*Width+threadIdx.x];
        Pvalue += Melement * Nelement;
    }
    Pd[threadIdx.y*Width+threadIdx.x] = Pvalue;
}
```

This is the current function which would take something similar to what a regular host matrix multiplication would take right. Kernel function is something that each thread is executed okay it needs a pointer to three matrices right and the width. So as a thread what would you be doing you know that you have to produce one element of the output you first figure out which element of the output you are producing right.

If you recall the regular host matrix multiplication loop then there is an eye that goes from 0 to N there is H that goes from 0 to N IJ is what you are producing and then the K also goes from 0 to N to produce IJ. So now you are saying which IJ and I then I am going to go loop over K right. So which IJ you are will be given by your thread ID you have generated so N width cross width block of threads then you get your Y index NX index it tells your I and J so to figure out where to read.

So this is where you DD will go right your index what are the MD and ND roll numbers and column numbers again your index. S your index test is you row and that column how do you get the element of that row now if it is a row then you just go 1 + 1 + 1 if it is a column because we have not declared the MD and ND as an array CUDA which will be 2D array or 3D we have declared to be a regular pointer.

So now we have to figure out the address based on your IJ you know the width so it is easy to do you take this in a loop. So K goes from 0 to width you keep varying your address within fetching

element from your address you are fetching the element from multiplying them and adding it from partial result right.

And once you have done you put your number in the proper place in PD if PD were in 2D array then you would say my index why my index J my index X because it is a 1D array again you have to compute the address one and how do you call it. So this is host code again am going to try to fit in number 2 in the same by making the other stuff smaller which we have looked at before.

So you are going to call that kernel that same name that we saw earlier with dimension of the grid dimension of the block and dimension of the grid as declared to be a one by one thing and dimension of the block is declared to be a width by width N and there is no shared memory being used so you do not say I want so many bytes of shared memory and then you send the 3 pointers and array. The parameter that you send here will get copy related by the proper pointer on the device side.

(Refer Slide Time: 09:16)

Single Thread Blocks

- One thread-Block computes Pd
 - Each thread computes one element of Pd
- Each thread
 - Loads a row of matrix Md
 - Loads a column of matrix Nd
 - Perform one multiply and addition for each pair of Md and Nd elements
 - Compute to off-chip memory access ratio close to 1:1 (not very high)
- **Size of matrix limited by the number of threads allowed in a thread block**

Courtesy Kirk & Hwu

So this is how you would do you would manage if there is only one not necessarily one SM. If you have small enough number of rows and columns that you do not exceed the maximum block size okay. But before we do that what can you do to reduce the number of reads how many total reads are happening here and should there be $2N$ should there be $2N$ square roots okay.

$2N^2$ is unique potentially unique elements so you have to remove but each element is being read by everybody whose is handling that row that people are handling it. So if copy that wrote that entire row once to shared memory then you will be reading shared memory still we reading N^3 times yes but your read will gets faster total number reads plus that is actually going to go up usually that will be a good thing shared memory is small.

But in this case since we are assuming that block fits and this block is also small matrix is also small okay now the block does not fit. The host does not see share them it is like local cache local cache for each of the processor in each has its own copy of shared memory. So would be complicated for host to say put it in that shape maybe that shared memory okay. So what if now the matrix size is bigger?

Does not fit into one block so you have to have more blocks so again there are probably many ways you which you can break the problem down one may be you sub divide your block your output matrix of two blocks of the size that you can handle and then you produce one block at a time each third block produces one block of the output matrix alright. So now that block is going to read all the blocks in a row in one matrix of the blocks in its column in the other matrix and then do the same thing.

But again there is a repetition within a block there is nothing you do about repetition across the block because there is no real synchrony that you can maintain across blocks but within a block you can still bring it within the shared memory.

(Refer Slide Time: 13:23)

MATMUL: Multiple Blocks

- 2D thread block computes a $(\text{TILE_WIDTH})^2$ sub-matrix of Pd
 - $(\text{TILE_WIDTH})^2$ threads/block
- Generate a 2D Grid of $(\text{WIDTH}/\text{TILE_WIDTH})^2$ blocks

Courtesy Kirk & Hwu

And read then from the shared memory whatever you are requiring something that we will talk a little bit later in more details is that when you read when that kernel of that threads within a block are all reading what kind of memory access are being generated for example let us go back to that loop of a kernel again.

For $K = 0$ to width U as thread are saying read this read that take the product accumulate right. The same time another thread at least the once with in your warp are doing same thing right and as long as other threads within the warp are near you right which it would be saying your block ID which goes $I = 0$ to width = 0 to width is basically the way you are device is matrix the possibility also the output matrix is possibility then the ID's within a warp will go from K to $K + 31$.

And unless this dimension of the matrix is there will be one chunk of row in it or chunk of column in some row if it matrix is 17 and 17 what will happen. So 16 so the warps are 32 right so 32 and if you have 17 is probably small amount let us pick bigger number 35 then you get 32 and then the next warp of 32 is going to be doing something over here and then something over here okay.

So coherence will be lost and so the good idea to organize your data in the way you would expect the warps to be organized and sometime it is more useful or more advantageous to let few of these elements or thread not do anything. For example if you had to have 33 is probably wrong

number you had to have 31 then instead of the 32 element of the warp going to the next row you would let 32 element do nothing okay.

So that the next row again remains aligned and so when you are looking at both structuring your computation and making struts alignment is something you will want to keep in mind things are generally okay good if they are multiples of 16, 32 typically on a CPU things are good they are multiples of 4. Here some granularity is somewhat big so we can get a block multiplied what happens if now there can this still be problem then you will have to break then you will have to say accumulate partial result for just this chunk where is my neighbor is also working accumulating partial results for this same chunk.

So there is some complication in code that we will have bring it to account for it but right now am not using shared memory at all just for simplification directly doing this on global memory even the size of grid is limited by the size of short 65535. So if your matrix size is 65535 times 512 or 1024 or whatever the limit is for the given architecture in each dimension then one grid one kernel call will not doing it.

So then you will say I am going to write multiple kernels and each kernel will produce one segment of my output where the segment is no bigger than 65535. We can do that was the next thing I was going to ask what is the difference between the 2 1 dot X compute capability 1 dot X that is the case 2 dot X that is not the case more over will I have mentioned this notion of streams right.

You can create streams and then multiple streams can have on going kernels so starting of kernels is not necessarily an issue as a CPU you can give all the kernel and say now let us wait for them to get that is one issue although now you can get 6 gigabytes of global memory and probably 8 coming soon it can be a problem something to be aware of but there is also issue of in fact let us back up.

When we broke a kernel which was one block for entire computation into some $KY \times K1$ by $K2$ blocks because one block could not do anything could not we have said let one block compute one thread compute a block of output rather than one output so on block is limited to you have

SM's then you are going to want to run enough blocks right. As a matter of fact even if you have one SM you have probably want to run enough blocks.

At least you definitely want to run enough threads which means enough warps and enough warps does not mean one or two enough warp means something in the order of 16 to 32 warps so if your warp size is big enough 32 warp of blocks which you can do in the 2 dot X it is probably okay but you would want many warps potentially many blocks on each SM and would want at least that number at least as many SM as you have.

But once you go beyond that you do not get benefit of having lots of available warps to run what is the benefit of having 32 warps the on warps is reading something from the global memory some other warps result has arrived and it is product can be computed. If you have enough so that all of these down times when you are waiting for others data to arrive somebody's data has already come in after there is not much further benefit to be gained by having extra warps.

Extra warps waiting to run in fact in principal at given clock cycle you need exactly one warp waiting to run module SM sometimes you can run 2 warps at a time and module those efficiency type issues as long as the number of warps you can run at any given clock instant or till then you do not need more just need have enough so that a small fraction will be ready to run hopefully every clock.

So what threads going what that is up to you the only the thing that the system does to you is it generates that many threads and it assigns to them hard coded thread hard coded meaning predetermined so if we have made 1D block then threads will go thread 0 will be running on processor 0 that is also known of particular value to you then thread 1, thread 2, thread 3 onwards.

What does threads 3 does you so what you do with that ID is variable? Is up to you 0 to 15 will always be in a warp so 0 to 31 and any multiple of 32 will be in the warp and even there 0 to 15 will be half warp and 16 to 30 one will be the other half warp.

(Refer Slide Time: 25:38)

Synchronization (Block)

- **__syncthreads()**
 - block barrier AND
 - ensure all global/shared memory accesses by all threads are visible in the block
- **__syncthreads(int predicate)**
 - returns the count of threads where predicate != 0
- **__syncthreads_and(int predicate)**
 - returns non-zero *iff* predicate != 0 for *all* threads
- **__syncthreads_or(int predicate)**
 - returns non-zero *iff* predicate != 0 for *any* threads



So when you do have multiple blocks or multiple threads within a blocks there has to be set some way to synchronize that so far the way CUDA was evolved you really first class citizens as far as synchronization is concerned only within a block threads within a block can synchronize themselves with each other and that is the main synchronization method is called synchronous.

It is like a barrier it is group is your current block and everybody must cause some threads the slide we have seen before. So if somebody is in a IF condition which skips the thread then your interrupt it is a little bit stronger than sync threads in that all your memory accesses is made before that particular all the rights you have done before the sync threads are guaranteed to have completed before you proceed from synchronize right.

So it is also a flush all your memory writes shared memory as well as global memory it is a flush right so as far as system is concerned and when each one goes out the global memory as piece of data there is a common global memory it is not replicated or anything you would in a database so there is one address were that everybody says I have written something and that should be there. Of course I wrote it at this time and I do not know when it is going to get to the level after I have come across the same thread it is the memory that is something you can reliable.

But do not think of that way in that you have no idea of the other block where it would be so as a program control is concerned you cannot rely on that fact it is just that system points of view this things has been done so really only make sure that or you can only assume this for the rest of the

threads within the block. If somebody else has written something to the data you have interested in you know that is there now.

There is a sync threads with certain variants for example you can give it up predicate give it an integer think of it as a predicate and then among all the threads of the block there some commonality with insight in that integer can be evaluated okay. For example you can if you use the standard sync threads then it does with predicate so the second call in the slide it will be sync thread everybody is going to come to the barrier.

But it will also return a value which will be how many calls everybody has to call sync thread right and then everybody is calling the same version of sync thread so sync thread some predicate is called by all the threads of the lock and you can now if everybody called with a 0 or not. So it is really a Boolean it is talking an hint you can imagine it has a Boolean everybody is so sorry that not everybody value is 0 but how many people value is 0.

You can get a counting version in this case and there is add and out which says that everybody or anybody is not equal to 0. Unfortunately the interesting version available on 2 dot X as you might have guessed there are also some special synchronization that you can do within a block although if a block is in lock step fashion there is certain things you may want to know about other threads that are executing with you at a given time.

And we will are going to take quick break for changing of the slide and then I will tell you think of what kind of things you might want to know about other threads that up in the SIMD in lock step with you X modulo conditionals length sometimes people may be diverging because of branches.

So let us continue it is only a Boolean right it only tell you let us look at the slide again so typically saying threads you just know that everybody came to this point right now suppose people come to this point within agenda I say I have want to do X or Y then you can figure out people agenda something about people agenda not everybody's agenda. You can ask how many people are coming with positive agenda right.

Or you can ask is everybody coming with positive agenda or is anybody coming with positive effect. So it is really that those variance. Yeah it is value that expression will be value and then passed with to the function call well it is the see Boolean so if it is an integer and Boolean true false means not 0 not 0 anything that is not equal to 0 is not true. You can use it in the logic yes yeah you can count number ones within a block yeah.

So what kind of things would you these kind of things we also useful with in a warp no need to have notion of sync right because your kind of in sync but you can may still want to know if everybody has a positive agenda right. You may want to agree on doing something sometimes you are co-operate right. You are not doing independent things you may want to decide should we go ahead we are all computing something and you want to do an early exit if everybody has decided that there is no point going forward you can quit everybody can quit.

Anybody says ok to quit may be it is okay to quit so those kinds of things is why is predicating is there and for warp within a warp you can still do all in any or it is all ballet which is a bit more rich it is not just counting how many people are coming with the positive agenda right it is giving you bit pattern of who is coming with a positive agenda. You have got a 32 bit agenda passing and there is 32 people in the warp is going to tell you which of them had non zero predicate that bit will be turned on.

So everybody can determine who is voting yes who is voting no and then based on that you can chose to do different things ahh the ballet is only supported on 2 dot X on any of supported I think on all of them these all required hardware support epically the ballet one rather complicated.

(Refer Slide Time: 35:16)

Intra Warp Synchronization

- `__all(int predicate)`
 - Return non-zero iff predicate != 0 for *all* threads
- `__any(int predicate)`
 - Return non-zero iff predicate != 0 for *any* threads
- `__ballot(int predicate)`
 - Return an int with nth bit set iff predicate != 0 for the nth thread of the warp
 - Only supported by devices of compute capability 2.x

What about the memory we need synchronization on the memory only within a block that is for a read you should have issued a read and your data has not come yet then you cannot come your next instruction schedule we would have issued right you can go ahead and do something else okay. There are atomic operations on memory either you can lock and full scale locks are not really provided at least on the hardware level even CPU's it typically only get OS implementing the variety of mutual exclusion method.

So at the hardware level basic locking on memory locations is provided so and to expose that there are instructions in the PTX as well as function cause that translate to those instructions which say modify something add something to this variable directly in the memory right add one to increment this variable. So which would otherwise mean read the current value add one write the result now it is increment this memory location meaning that if two people are incrementing or reading or make sure that you do not between them whichever occur first it up to you.

But once your increment has started right nobody is going to read the variable until you have updated okay and so there various you can look up the list you can find you can add things to it you can subtract things to it you can multiply atomically you can swap a value with the or register value with the memory location atomically that is what atomic will change function is.

(Refer Slide Time: 38:03)

Atomic Operations

- Read-modify-write on one 32-bit or 64-bit word in global or shared memory.
 - Atomic ops on 64-bit words in shared memory available for 2.x only
- If the target is mapped page-locked memory:
 - it's not atomic from host's perspective
- Only supported on signed and unsigned integers
 - except `atomicExch`
 - and `atomicAdd()` for 2.x
 - that also work for float.
- More generic Compare and Swap
 - `atomicCAS()`

You can take the min of the register value and a memory value atomically and thing that sort most of the atomic operations are actually designed for 32 bit integer operands okay eventually they got also upgraded to 64 bit operand but still integer long the 2 dot X has further generalized it so you can actually add two floating point things also in an atomic were but in 1 dot X you can only add integer things increment integer things and integer things and hence so forth.

The exchange of register and the variable because that kind of type free is allowed on all 1 dot X or 2 dot X that is exchange is some change in some sense of exception but they all provide this atomic gas with for compare and set okay if you have done an OS course you have seen either compare set or come version of compare and said what does it do this is a prerequisite we should know come there and sell.

It is definitely one of the ways of complimented that is a very kind of restricted of compare and said but it really as you compare you provide an address and you provide a value right and if that address is equal to the value then your new value can be put in there okay that is the standard comparisons. If it is equal to the value the that memory location does not get updated sometimes as a side effort you get that old value back it is not written then you know your value got written it meaning that the old value was what you comparing against.

Implement it is used for many lock free algorithm will talk about those also later on you do not want to physically lock a resource as you do for the critical section normally and unlock it is a

locking of sorts but very fine grant level. And so we will use compare and set to do lock free synchronization later on remember we talked about memory being mapped you are addressing the same memory on the host as well as on the GPU and if you are performing atomic operation on the GPU host in the meantime can also be right into it.

And because host really is physically writing to the same locations it is writing to the proxy how they interact with each other is not well defined. It could be done but then atomic organization will become extremely slow it has to then synchronize with the host also and so it is not provided it says that make sure that host is not operating by whatever mean you otherwise figure out this is an atomic operation only from the GPU's point of view.

In the context of the GPU so if there is other GPU operation on the memory location that will get locked for that small bill okay.

(Refer Slide Time: 42:12)

```
CAS Example

__device__ double atomicAdd(double* address, double
val)
{
    double old = *address, assumed;
    do {
        assumed = old;
        old = __longlong_as_double(
            atomicCAS( (unsigned long long int*)address,
                __double_as_longlong(assumed),
                __double_as_longlong(val+ assumed)
            ));
    } while (assumed != old);
    return old;
}
```

Here is an example of cache it is called atomic cache function in this case atomic cache is being used to do double operation so I said that you cannot take floating point thing and add together add with another there is no atomic add on floating point is accepting 2 dot X. And even in 2 dot X there is no atomic 2 dot double values okay but you can use compare and set to implement all other kinds of atomic acts.

In fact CUDA did not even provide all these different varieties of atomic function because they can be simulated they can compare and set. So how will they work and here there is a little bit of lingo here do not worry about too much about double long it is really a cast taking a double and thinking of it is as 64 bit integer and similarly long long as double is the opposite of that take 64 bit integer and interpret is as double in an architect independent way.

So this is a device function it wants to add a value atomically to this address okay and nobody should be able to do in the middle. So why would this work what is happening here? It is a spin lock that is something unless you have some other thing you cannot do something other things also in the meantime but compare and said typically spinlock style it is you should more equate it rather than with spin lock style with a lock which is non-blocking.

You are trying to block and if we did not get the lock you have to do something thing you have told that this lock is not available but if you are nothing better to do then the new spin lock. This is spin lock so what does this call provide you? So basically what are you trying to avoid? You are trying to avoid typically what do you do you read the value add the thing want to add and then write the result it is not about context switch for say But it is the logic remains the same.

Somebody else doing it at the same time go ahead so I read a value I add something and now I am going to write it but I want to write in a way that I ensure that nobody else has come along and modify the value in the interim that is what atomic add would have done and it would have log everybody out right here since I could not do here i have read it I have added the thing I want to add.

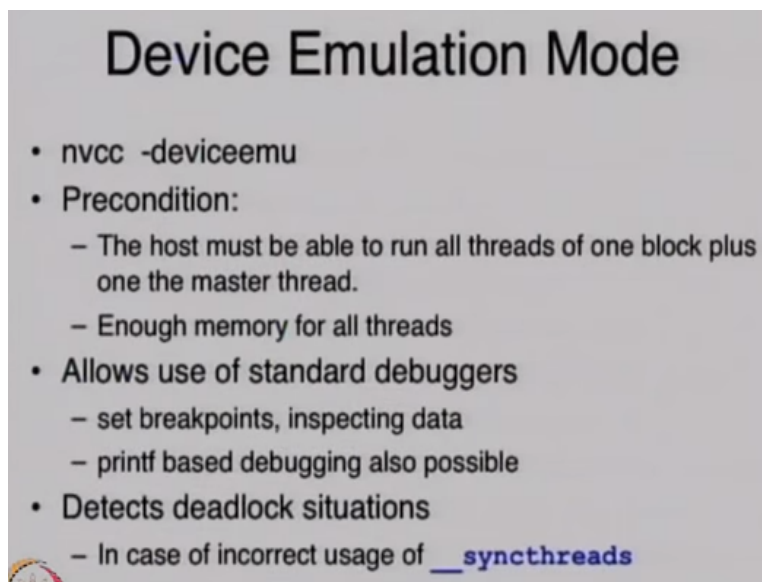
Before writing a check is it still the same if it is still the same that means nobody else come anything with it atomically with it atomically changed it to the new value and go on. Otherwise somebody has come and may be done something with it and my old value is become stale I have to really I read it and do that same thing.

If this time nobody else came in the interim then I succeed otherwise I continue to do this can potentially go on forever it is in some sense similar to atomic act. Atomic add is also trying to acquire a lock to that memory location and they modifying if I can acquire a lock because every

time I go to catalog somebody else get the lock. But here also the reason it is changed is somebody else read it and did it something so it is not that it is busy wait for no reason.

It is typically not being looking here right it is a loop just sold if you fail you are going to retry it is not a lock type loop where I am locked out and I keep checking the lock until somebody releases it. It is not that kind of it is loop that says I am going to go and do it but just at the time of doing it I will check whether I am doing the right thing about. If I am not then somebody else has done something else and I should go and redo it.

(Refer Slide Time: 48:20)



Device Emulation Mode

- `nvcc -deviceemu`
- Precondition:
 - The host must be able to run all threads of one block plus one the master thread.
 - Enough memory for all threads
- Allows use of standard debuggers
 - set breakpoints, inspecting data
 - printf based debugging also possible
- Detects deadlock situations
 - In case of incorrect usage of `__syncthreads`

Let us go quickly through some of the things you might want to try if you have not already there is a device simulation note somebody was asking me what if I do not have put a CUDA on my machine. CUDA will work with device simulation mode and it will give you some fair idea of whether your code is go speed right or not there is lots of problems that is not going to cache.

Primarily because the timing or very different also there are differences in the hardware specially in the one dot X hardware floating point denom operations are different from Intels denom operations. As a result of which if you are doing anything that requires demon then is going to be different. You can use regular GDP not CUDA GPD to compile this to debug this but again to be careful that some other.

(Refer Slide Time: 49:46)

Caveats

- Device data accessible to host code and vice-versa
 - Similarly, all functions callable from host/device functions
 - Could be thought of as a feature, if used carefully
- Race conditions are less likely to manifest themselves
- Warp vote is not properly emulated
 - Warp size is equal to 1 in device emulation mode
- Floating-point computation may not re-produce device results
 - Different compiler/instruction sets/architecture
 - Some store intermediate results in extended precision
 - Some support denorms, others don't

Erased conditions are unless you designed it very carefully are normally there in a parallel program. And so they will probably not get caught if they are running in a position mode and there are some things that happen within a block. So the synchronization will behave somewhat differently and some of them are not even implemented for example in fact this is I have not checked if they have updated it until CUDA 2.0 I think the voting within a warp was not properly emulated in the CPU or correctly emulated on the secure bug in it.

(Refer Slide Time: 50:40)

Thrust Template Library

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>

int main(void) {
    thrust::host_vector<int> h_vec(1 << 24);
    // Initialize host array now ..
    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;
    // Do more ..
    // Later transfer data to the device
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());
    return 0;
}
```

Alright there are other things there is something called thrust template library that you may use this is not for your assignment if you are going to do CUDA project. Because it has a reduce and

you can use it you have to implement your own reviews there is a reduce in the example also which is actually not going to work because it does not work for unlimited data.

The way this library simplifies things again I am not going to go through the library I am just going to review feel for what kinds of things you can do with it or how it simplifies writing CUDA programs at a slight cost of efficiency it is template you can declare host vectors and device vectors which is typically what you do right you create an array on the device you copy it over and then you do things on the device.

Here you can simply say a thrust host vector of type hint initialize with it certain size so in this case it saying 2 to the 24 element integer array is being generated on the host called H underscore vect and another vector D underscore vect is going to be initialize constructed with H underscore vect meaning you have just allocated the memory and copied it effectively. So internally somewhere there is a CUDA and alloc CUDA MEM CPU happening as a result of this.

You can also copy parts of it so you can say copy into the D vector being over here and over here from this host position and there are several others that you welcome to go look into and there is also somewhat older version of not template library but commonly used thing like produce can stuff like that called could PP some people you both of them and what is your verdict on which one.

Some people think that is faster and they are not right actually they are not wrong either because for something is being faster but for many other things could be P is faster. There are also with the SDK that you would typically installed and are installed in the systems which you are using there is something called CUDA utility library Util dot H it has some basic ease of operation type functions again am not going to go into that part and example is CUDA.

So you it is most of the stuff there is macro not all but most so CUDA safe call you wrap that inside wrap any CUDA call into that CUDA safe call macro and it turns it into if this cause succeeds otherwise brain something so it is going to if there is error in making that call then you are going to get a message. It is not well supported you would not it is okay do this when you are learning to program but this is not what you would not want to use long term okay.

(Refer Slide Time: 55:03)

Checking Return Values

- Functions return `cudaError_t`
 - `cudaSuccess`
 - `cudaErrorInvalidValue`
 - `cudaErrorInvalidSymbol`
 - `cudaErrorInvalidDevicePointer`
 - `cudaErrorInvalidMemcpyDirection`
- See `CUDA_SAFE_CALL` in `cutil`

```
CUDA_SAFE_CALL(  
    cudaMalloc((void **)&d_odata, MEM_SIZE)  
);
```

So let me take you through few code examples before I am going to talk about the detailed performance related issues. 3.0 does not have simulation mode interesting so now they are completely ruling out people having not CUDA cards.

(Refer Slide Time: 55:32)

```
CUA Sort Example  
  
#define NUM 256  
int main(int argc, char** argv)  
{  
    CUT_DEVICE_INIT(argc, argv);  
    int values[NUM];  
    for(int i = 0; i < NUM; i++)  
        values[i] = getNextValue();  
    int *dvalues;  
    cudaMalloc((void**)&dvalues, sizeof(int) * NUM);  
    cudaMemcpy(dvalues, values, sizeof(int)*NUM,  
               cudaMemcpyHostToDevice);  
    ParallelSort<<<1, NUM, sizeof(int)*NUM>>> (dvalues);  
    cudaMemcpy(values, dvalues, sizeof(int)* NUM,  
               cudaMemcpyDeviceToHost);  
    cudaFree(dvalues);  
}
```

Here is a simple example this is more detailed in some sense but more complete piece of code and am not going to again go through all the details of it just to give you a structure say you get use to is style one more time. So what is happening in the beginning then there is this is a CUD macro `cut underscore device underscore` in it which if you have multiple CUDA card then it initializes the proper CUDA card or one of them unless you control it using the bug be you can set your CUDA program to run on any of the available devices.

So what is happening here is that you are creating an array of value on the host using some method called get next value and you are going to copy that array value into D value which is the device version of the same array and then you going to call parallel sort to sort that array on the device. And at the end of it you are going to result take the result back out and free the memory in which you had sorting down and the values is an array is in-place sorting is done.

Is array will at stop function which are going to use and then here is the kernel is the parallel sort in the beginning it is using shared memory and using the external shared style which mean in fact something I should have pointed if you go back and look at the configuration the call for invoking the kernel it says how much memory in it. It says I have got an num elements in the array and increase the size int size in each element that many bytes shared memory I am going to require but a lock.

And the block is simple one block which is going to be the size of the array which means it is the kind of soring algorithm where you say I have got N processors working on N elements so I have got a set block size is N and being of array and this block is going to get some amount of shared memory so that all of the array can be stored in the shared memory and so how do you get the data into shared memory by copying it from B values inside here it is being called values because the formal parameter listed on the kernel is calling it values.

And one thread is reading one of the values ultimately they are going to sort it together they are all going to access different parts of the array but they together need to fetch this entire list into this location right. So they will decide there is N amount of work to be done N of us one will do each okay so everybody has read something and then they sync to make sure that the entire array has brought in right.

So you cannot simply assume that they have read and go on and next start to read things from the shared memory okay. At this point the entire data is in shared memory again we probably need to stop here