High Performance Computing for Scientists and Engineers
Prof. Somnath Roy
Department of Mechanical Engineering
Indian Institute of Technology, Kharagpur

Module – 04
GPU Computing
Lecture - 34
Introduction to CUDA programming

Welcome to the class of High-Performance Computing for Scientists and Engineers. We are in the 4th module of this course which is GPU computing. In the last few classes I have given you an introduction to GP-GPUs and discussed their utility in high performance computing, and also introduced a programming paradigm for GP-GPUs which is provided by one of the leading GPU manufacturing vendors NVIDIA and that is CUDA.

In today's class and in the following class also we will start looking into CUDA programs. We will see how a CUDA program actually looks and what are the different components in it and what are the different hardware and software instances and execution instructions we provide in our CUDA program.

(Refer Slide Time: 01:26)

So, this class is an introduction to the CUDA program. We will discuss a few of these issues CUDA being an API which we usually need not use for our computers which if they are not supported by CUDA enabled GPUs for normal computers, we do not use CUDA. So, we will

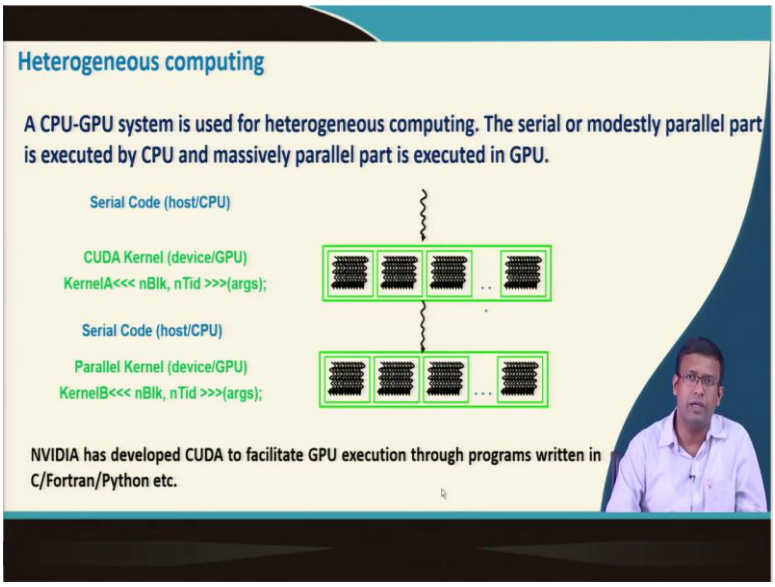
see I think it is judicious to have a couple of slides on CUDA installation and documentation on CUDA.

So, how can we get this software? How to install this software, some ideas on that and then we will see how CUDA programs can be compiled? Then we look into a sample program which is again provided in the CUDA programming guide as the first CUDA program in that guide which is an addition of two vectors.

We will see what CUDA parallel versions of vector addition programs look like and then we will see some of the components of this program which are typically of importance in order to understand CUDA programming. These are kernels and threads and how data transfer happens across host and device and how this device data is managed by the CPU.

So, by device we mean GPUs and by host we mean CPUs in a typical CPU GPU CUDA program.

(Refer Slide Time: 02:44)



Well, so, I am discussing a CPU GPU program and this is called a heterogeneous program. Or the CPU GPU system when you use it is called a heterogeneous computing where the serial part of the program or modestly parallel part; that means, parallelized in 4th threads, parallelized in 10th threads, that type of part. That part runs in the CPU and the massively parallel part which is parallelized in terms of thousands of concurrent instructions is executed in GPU.

So, when usually we talk about a GPU program it is mostly a heterogeneous paradigm following program where part of the program is running in CPU; which is sequential or modestly parallel and the massively parallel part is running in GPU. Therefore, if we look into the program, it looks like that there will be a serial code which is running as an instruction in the host or CPU. Then groups of threads will be launched in blocks, and these multiple threads will take care of the parallel component of that program again in the serial part there will be CPU code, again there will be multiple threads which are running in GPU. This we have discussed in detail in last class.

Therefore, what we understand, from here is that when we try to develop a CUDA program or when we execute a CUDA program, we have to consider the fact that a part of the program will run in CPU and at some instance CPU will offload it jobs to the GPU and that part will run and as massively parallelized component of that program in GPUs.

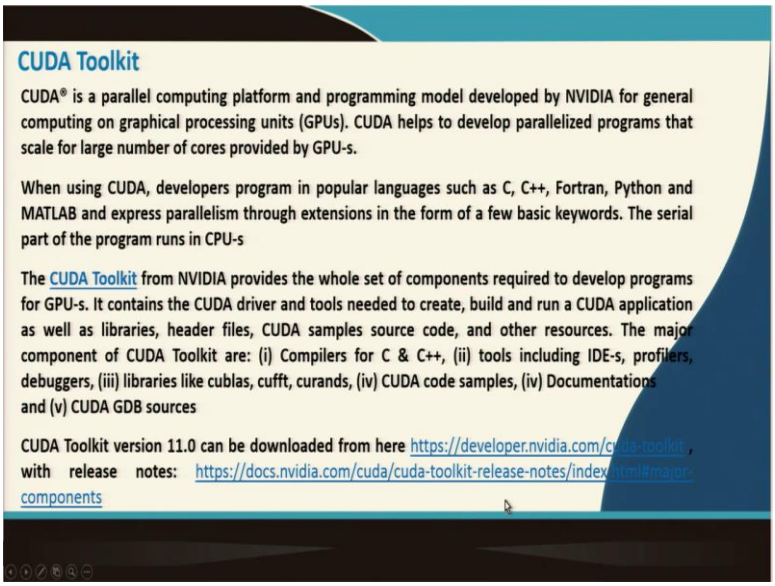
Therefore, a programming paradigm is required which supports heterogeneous computing, which helps part of the program to run in CPU and which offshoots the program to GPU, allows communication between CPUs and GPUs and then executes the program in GPU.

So, for that NVIDIA has developed CUDA, which is initially named as compute unified device architecture, now they do not use this term, they use the name CUDA only for that API, which facilitates GPU executions of programs which are written in common programmer friendly languages like C, Fortran, Python.

So, you have a C program or you have a Fortran code, you identify certain part of the program which can be massively parallelized, and now develop a CUDA C or CUDA Fortran program where over the C or Fortran program; you write CUDA instances and ask that parts to be executed in GPUs.

This has certain similarities with the previous programming of parallelization APIs like, especially with OPENMP also MPI, which we have understood that we can start with a basic C or Fortran or Python or MATLAB code. We can identify the regions where it can be parallelized and as the GPU is to take care of that region, there CUDA instances will be required.

(Refer Slide Time: 06:08)



So, CUDA is a parallel computing platform as well as a programming model which is developed by NVIDIA. CUDA provides the parallel programming platform, but CUDA itself is a programming model also. For general computing; using graphics processing units, which we call GPGPUs computing.

CUDA helps to develop parallelized programs that scale for a large number of cores provided by GPUs and this is done at the compiler and software support-based level, that when you execute a CUDA program it goes over 1000's of threads, maybe 10,000's of threads. Typically, if we think of an open MP program running over a large number of threads, there will be substantial overhead due to contention, memory issues, scheduling, load balancing, etcetera. As we increase the number of processors there will be more overhead. Therefore, if we think of a CUDA program it is running over a large number of processors and they can be huge overhead.

However, the CUDA itself is designed in such a way that through its own system level support and hardware software-based scheduling, it can take care of the overhead, and the overhead is really not that high and you can get a very high speed up.

So, CUDA provides the scalable parallelization over a large number of cores present in a GPU. While using CUDA developers' programs in popular languages like C, C++, Python, Fortran, etcetera.

The parallelism is expressed in form of few basic keywords or few basic CUDA constructs. The serial part of the program runs in CPUs and the massively parallelized part is expressed using the CUDA constructs or CUDA instances that go to the GPUs.

Now, the entire CUDA software, it can be available from NVIDIA website as CUDA Toolkit and this CUDA Toolkit will run in GPU. So, if you have NVIDIA GPUs you can download CUDA Toolkit from NVIDIA website and if you have NVIDIA GPUs you can run it on there.

This CUDA Toolkit provides a whole set of components required to develop programs for GPUs, it contains CUDA drivers and tools needed to create, build and run CUDA applications, as well as libraries, header files, sample source codes, profilers, debuggers, etcetera.

So, this CUDA Toolkit contains everything related to developing the CUDA program, building the executable, compiling the CUDA program, and running the CUDA program. As well as if you have to debug the program if you have to profile the program you will get the support is supporting files in the inside the CUDA Toolkit.

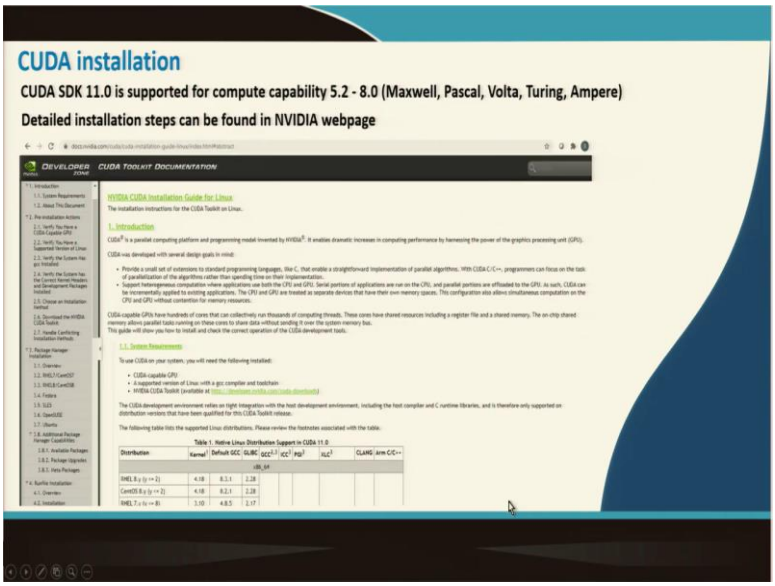
Furthermore, you get documentations inside CUDA Toolkit. So, everything related to CUDA is found inside the CUDA Toolkit. The only thing you have to do while, if you want to write a CUDA program and execute it you have to download CUDA Toolkit from NVIDIA website.

The important components are compilers for C++, different Toolkits, profilers, different tools, profilers, debuggers, libraries like cublas, cufft, curands, blas is the basic linear algebra library, many linear algebra solvers, matrix solvers are included in blas.

The blas libraries equipped with CUDA capability are present as cublas libraries which are also a part of the NVIDIA CUDA Toolkit. There are CUDA samples, there are documentations and debuggers like CUDA GDBs and their source codes also. So, everything is available inside CUDA Toolkit.

CUDA Toolkit version 11 can be downloaded from NVIDIA website with its relevant release notes.

(Refer Slide Time: 10:31)



The CUDA Toolkit version eleven gives us CUDA software development kit 11.0, it is the same, but the version of CUDA is 11.0; however, compute capability we have discussed earlier, it is a hardware support provided for GPU functionalities, and for CUDA eleven it is supported for compute capability 5.2 to 8.0.

So, for only the GPUs Maxwell, Pascal, Volta, Turing and Ampere, especially Maxwell to Volta widely available in market, Turing and Ampere are the newer are the most recent versions of GPU. However, all of them are supported by CUDA 11 if it gives support for CUDA 11. If you have a GPU which is of different version, you might not be able to run CUDA 11.

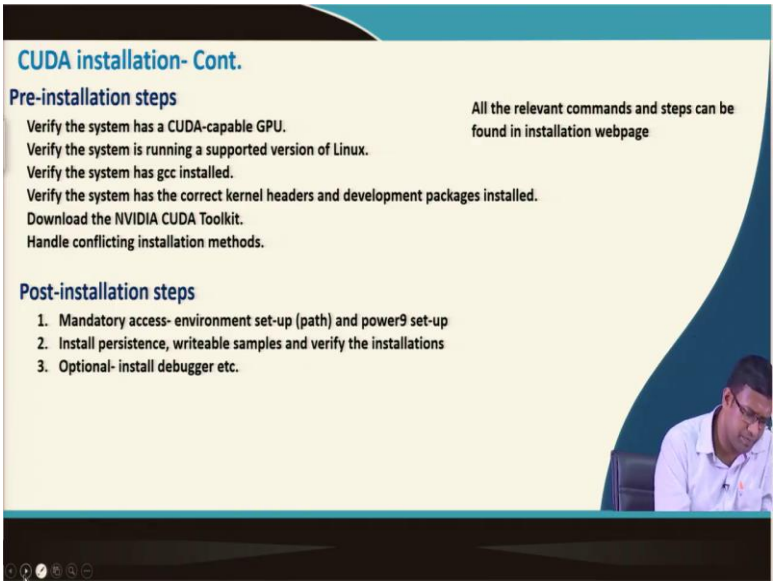
So, you have seen what is the compute capability and there are standard commands for that, what is the compute capability of the GPU you are using and which version of CUDA is applicable for that.

So, once we go to the CUDA rule Toolkit installation guide, that is; obtained in the NVIDIA site, we can find there are introductions that CUDA does and then it gives the system requirement.

It says which version of you need a CUDA capable GPU for that, you need a supported version of Linux with GCC compiler and tool chain. So, which version of Linux and which version of GCC is required, and then you need the NVIDIA key that CUDA Toolkit through which you can install CUDA over that?

So, three of the important aspects of installing CUDA will be, what is, whether you have the right GPU card, whether you have the right operating system like, if you are working in Linux, I have shown you here the Linux screen installation screenshot, but it can be installed in different platform involving windows mac etcetera. So, if you are using a particular operating system, which version of the operating system supports that CUDA Toolkit and what is the version of GCC that is required for that you. You have to check all these things and then you can install CUDA.

(Refer Slide Time: 13:00)



So, before installation you have to see that the system has a CUDA capable GPU. The system is running a supported version of Linux. We have to also see that the compute capability version for that GPU supports the particular version of CUDA you are trying to install. If you are using an older version of GPU, we have to use an older version of CUDA also, well.

So, verify the system has GCC installed, verify the system has correct kernel headers and development packages and then download NVIDIA CUDA Toolkit and handle complete installation methods.

Now, for all these verifications, whether you have the right version etcetera, you can find commands inside the CUDA installation guide, that you execute that command and check whether you have that capabilities.

After installation, what is mandatory is that you set the environment's path. If you do not set the right path, the compiler will not be executed as well as the executable will not be built. So, to set the path through bash_profile or module etcetera. You have to set the right path, so that you can find the compiler.

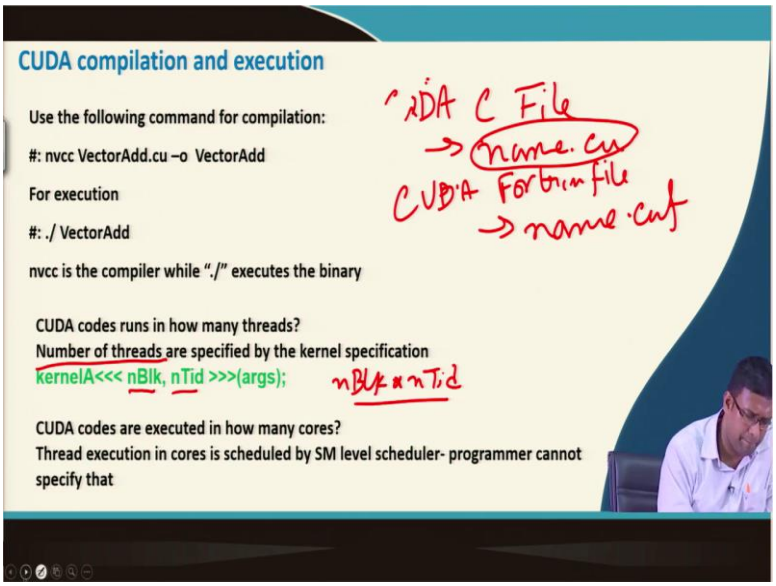
Then you have to install persistence writeable samples and then verify whether the installation is perfect, and optionally you can install debugger. It is a good idea to install debugger, while installing CUDA because when you are writing a program, you know that this program has a massively parallel component. So, part of it is a heterogeneous program with a massively parallel component, part of it will run in CPU, there is data transfer across CPUs and GPUs and part of it will run in GPU.

Therefore, debugging using primitive methods may not be very helpful, rather the debugger can give you a more efficient way of identifying the bugs in the code, much advisable that you have the debugger installed and use debugger for finding out the errors in the code.

Another thing to mention is that, when you execute part of the jobs in GPU, you can execute a write statement, you can write something to the screen. But that is supported only for some higher versions of compute capability or some more recent versions of GPU. Older versions of GPU do not support that and again because there are a large number of threads writing to the screen requiring certain extra questions and some practices are to be followed while writing.

So, instead of giving printf or star star, write structure comments and identifying the issues in the code it's of; obviously, advisable idea and that is not only for CUDA programming, for any general programming, even using C, Fortran basic languages. It is always advisable that you use a debugger and find out the bugs using debugger.

(Refer Slide Time: 16:23)



All the relevant commands and steps can be found in the installation webpage. CUDA compilation, when you develop a CUDA file, CUDA C file the extension is dot cu. CUDA CUDA C file's name is dot cu and CUDA Fortran file will have an extension name dot cuf.

So, when you write a CUDA C file; that means, a C program in which you write CUDA instances, the file name is to be given as filename dot cu, cu is the extension.

So, the compilation command is nvcc vectoradd.cu -o vectoradd. So, it will build an executable vectoradd and then you execute that file and this will be a heterogeneous program, which will run in CPU as well as the part which in CPU asks the GPU to run that will be executed in the GPUs.

Nvcc is the compiler, and this executes the binary dot.

Now, the question is CUDA runs in how many threads? Well, writing this we cannot specify the number of threads or by environment variable we cannot fix the number of threads. If we remember in openMP, we can set and specify the default number of threads, but in CUDA we cannot do that outside the program only inside the program when you use the kernel specification, kernel is the function which will be called by CPU and executed in GPU.

We can specify the block number and thread number, and the kernel will be executed as a grid of blocks, each block will have a number of threads. So, the total number of threads will be

nBlk (that is the block number) x nTid (that is the number of threads) in which this kernel will be executed.

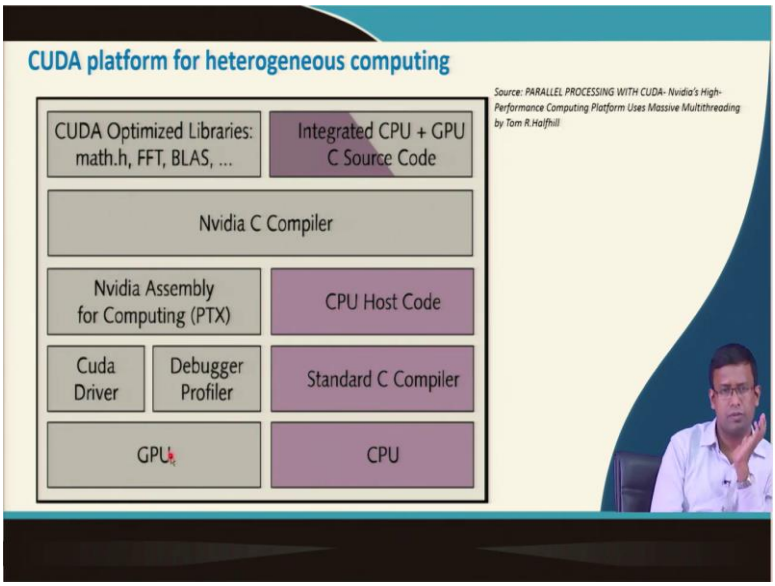
It will be executed in the total number of threads that is specified by number of blocks and number of threads, inside each block as specified during kernel calling. We cannot control it outside that or like for MPI you are specifying the number of processors in the run command for open MP we are specifying number of threads to the environment variable setup, we cannot do that. In CUDA in CUDA it has to be hard coded inside the program. This is the number of threads we are going to launch. So, if you want to change the number of threads you have to change these arguments. In how many cores CUDA codes will be executed? Well, in all cores you run threads which are expected to be much more in number compared to the cores.

Basically, we will discuss execution and scheduling of CUDA threads in a later class. So, basically when you launch these threads, a group of the threads goes to different streaming multiprocessors and each streaming multiprocessor runs a large number of threads. Of course, the number of threads if not specified otherwise are usually more than the number of cores. So, they are scheduled within the number of cores.

So, they run over all the cores, but there is a scheduling of the threads as well as scheduling of the active cores.

Thread execution in cores is scheduled at the SM level scheduler. Programmers cannot specify that, in these cores will be active and this thread will go to this core, also programmers cannot identify which core is running which of the threads. This is controlled by the SM level scheduler.

(Refer Slide Time: 20:46)



Now, when you develop a CUDA program for heterogeneous computing, we understand that it runs in both CPU and GPU. There are CUDA optimized libraries like cublas, cufft, etcetera which completely run in GPUs. If you yourself develop a program, that will be an integrated CPU plus GPU code, which we are discussing through heterogeneous parallel programming.

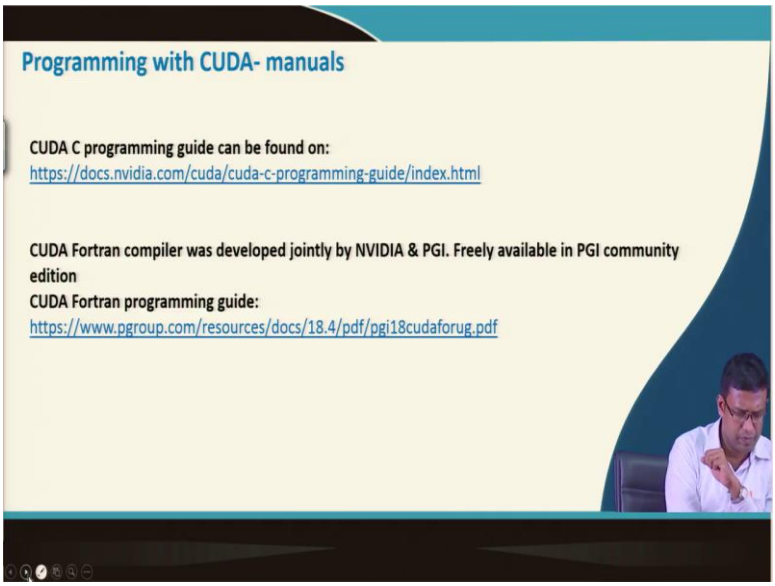
Part of the program, the purple part will run in the CPU and the grey part what is shown here, part of that program will go into the GPUs. Then you call the NVIDIA C compiler, it compiles two components of the code; one is the GPU part of the code which is compiled using an instruction set arguments called PTX, NVIDIAs own assembly language for GPU programs, and that goes to the GPUs. Their CPU part is compiled using the standard C compiler. So, when you are compiling nvcc GCC, nvcc means; NVIDIA C compiler which has CUDA for the GPU part and GCC itself is compiling the C part of the code. So, that is going to run on the CPUs.

So, when you run a CUDA program, it is compiled in two components; - one component is compiled for GPUs and a binary is built for the GPUs, one component is compiled for the CPUs and they have an interaction which is done using a wrapper inside the CUDA.

So, part of it runs in CPU and part of it goes to the GPU. And if you are using CUDA optimized libraries, they are written in PTX compatible language and therefore, they directly run in the GPUs.

But when you write your own program which is basically a C source code it has integrated CPU plus GPU code. CPU code is compiled using the standard C compiler, follows thus compilation commands syntaxes of standard C compiler and goes to the CPU. The GPU part uses PTX which is NVIDIA assembly compiler for GPUs and then it goes to the GPUs.

(Refer Slide Time: 23:22)

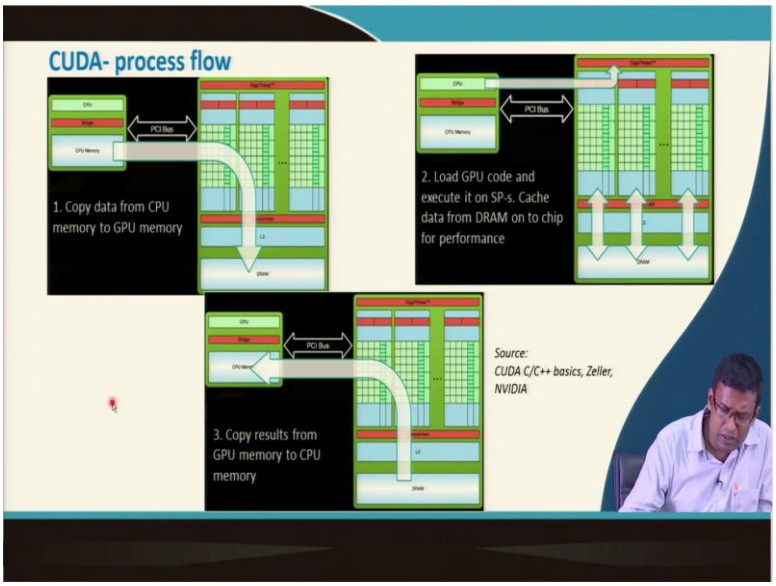


CUDA C programming guide can be found on this particular website and whatever I am showing in this class and also in some of the subsequent classes we will be following CUDA C programming guide, only because it is this guide which specifies how CUDA has to be used.

So, CUDA is also used over other languages like Fortran, Python, etcetera, and CUDA Fortran compiler is developed jointly by NVIDIA and PGI. PGI is a Portland group they have their own compiler sets and in the PGI compiler edition CUDA Fortran is freely available, the guide can be found here and it can be downloaded from PGI website.

So, all of the recent versions of PGI Fortran have CUDA Fortran inside it.

(Refer Slide Time: 24:27).



We will look into a sample CUDA program. We have seen that CUDA program which a developer builds or an application scientist works on is an integrated GPU CPU program. It is a heterogeneous program, part of that runs in CPU, part of that runs in GPU. We cannot directly ask the GPU to run something; we have to always communicate with the CPU.

So, we have looked into the process flow in the last class, that is that first you define the variables in the CPU memory and then CUDA copies the data from CPU to GPU. This is the first step of any CUDA CPU GPU program, that everything is defined in CPU and then CUDA copies the relevant memory from CPU to GPU.

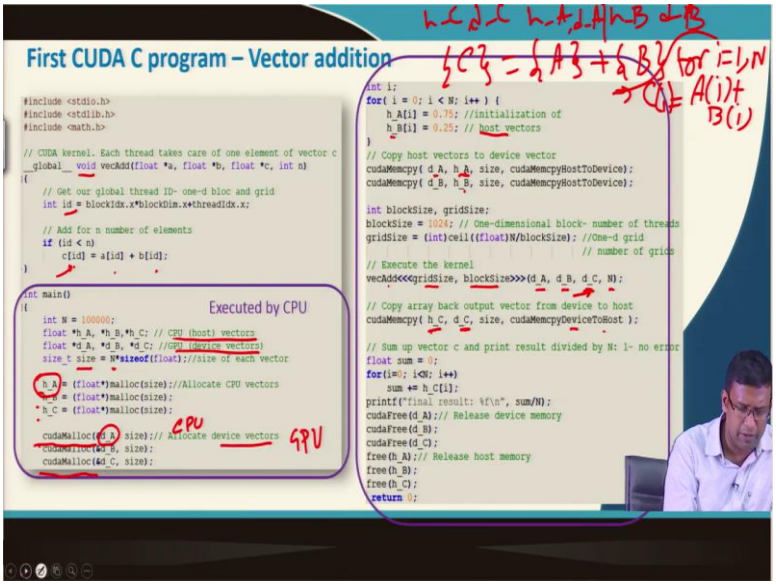
Then the CPU asks the GPU to launch multiple threads, and while launching these threads this concurrent instruction stream takes data from the GPU ram or device RAM and processes the data. So, it writes back the data in the device RAM. These files are not again readable by the user. So, it has to be copied back to the CPU, when CPU and GPU is connected via PCI express bus.

Therefore, CUDA instructions are to be given which will copy data from the device RAM or GPU RAM to the CPU RAM.

So, these three are the essential steps in a CUDA program. First copy data from CPU to GPU, ask to give a job to GPU as GPU is to execute certain jobs by launching a number of threads. While doing that they will modify some of the items in the GPU memory and then copy the

GPU memory data to the CPU memory data and write it to the files or read it or do whatever the programmer wants to do.

(Refer Slide Time: 26:46)



So now we will see a program which is our first CUDA program vector addition. There are two vectors of size 100000, one is vector A, another is vector B and they are to be added. So, we will run a loop of i = 1 to 100000 and add each element of the vector. Now these 100000 iterations are not done using a CPU C code, rather done as a function which will be executed in 100000 threads in a GPU.

So, how is this done? There are two parts of the program; - first we are inside the C program, first we write the CUDA kernel. This is the part of the program that will be executed in the GPU. Then we write the main function, this is the part which will be executed by the CPU, and this CUDA kernel which is add, this kernel is called as a function from the GPU in a particular format. We will discuss this format in the next class.

So, what is done here, we first define the number N = 100000 it's the size of the vectors. Then define the vectors, now these vectors, what we can do when we specify the values of the vectors? We cannot do anything in the GPU memory. Whatever we will do, we will do in the CPU memory. So, these vectors are to be defined in the CPU first.

So, we define a copy of the vector in the CPU. So, what we are trying to find out again is that vector C = vector A plus vector B. That is written as $C_i = A_i + B_i$., this is for i = 1, N, for N = 10000, we are doing $C_i = A_i + B_i$.

So, this A and B are two vectors or two arrays whose addition will run in GPUs, but we cannot define the vectors in GPU we cannot set their values in the GPU so you have to do that in CPU.

So, we make a copy of the vector in CPU which is h_A, h _ B, h stands for the host; CPU is the host. We make another component which is the device component, or GPU component d _ A, d _ B and d _ C.

So, we define h _ A, h _ B, h _ C, which is CPU or host vectors d _ A d _ B d _ C is GPU or device vectors. Even if we are as we are defining them, we cannot define them inside GPUs. GPU memory we cannot see we are defining them inside CPUs only. So, one question is that how GPU will know that these vectors exist, we will see that.

Now, we define the size of the vectors which is variable size which is the floats of N, N is 10000, that is a number of the elements in the vector now h _ A B and C, they have the size of size and we allocate the CPU vectors using simple malloc.

For the vectors which will rest in the device or GPU, we cannot go inside the GPU at allocating them there. We have to use a CUDA malloc command. This is the command by which the CPU allocated spaces in the GPU memory.

The vectors d _ A d _ B and d _ C, these are the spaces in the GPU memory with this address, and the size they are allocated in the GPU memory, by whom? By CPU and therefore, it is not the malloc CPU that is doing this using CUDA. So, this is CUDA malloc. Using the command CUDA malloc these vectors are allocated in the GPU memory.

So, CPU allocate device vectors using CUDA malloc commands. Well, once this is done, in GPU we cannot do anything, we have to give the values of the vector. So, we write all the elements of A are 0.75 all the elements of B are 0.25.

We can only initialize them in the host. We cannot do them in GPU, but now this addition will be done in GPU. GPU will do C = A + B. So, GPU needs to know the values of A and B.

Now what do we do? We copy the host vectors. Vectors are allocated in the CPU earlier, we copy the host vectors to the device vectors. We take the vectors from the CPU, copy them, and h _ A is chosen, and it is copied to d _ A in the GPU using the command CUDAmemcpy. We will discuss the CUDA memcpy in detail, but it takes the device array, the host array, their size and writes that this direction of copy will be host to the device.

So, it will be copied from host to device. All the values of h _ A for size of size will be copied to a vector d _ A which is now residing in the memory of the device or memory of the GPU.

Similarly, h _ B is copied to d _ B. So, this copying part is important. Then we define the block size that there will be 1024 threads in each block. That size of each block is 1024. The grid size has a total N number of grid points. So, this is the integer value of N by block size total number of that.

So, in case N is not divisible by 1024 there will be more threads than N. here, N is 100000. So, there are actually more threads than N, but that is okay. But we can see that the number of threads is large enough; of course, the number of threads is 100000 of course, no GPU has 100000 cores. So, threads are more than the number of GPU cores.

Well, then we call the executable vecadd which will take the grid size which is number of blocks inside the grid the block size number of threads inside each block and the parameters d _ A, d _ B, d _ C, d _ A plus d _ B will be d _ C and their size N.

Now we call this CUDA kernel. This CUDA kernel is a function which is specified using __ global __ specifier and it returns void; it takes A as d _ A B as d _ B and C as _ C and size N.

Then finds out the global id of each thread which is obtainable based on the grid size and block size. There is a huge component in CUDA programming called thread algebra by which we can find out the global id of the thread based on the local id of the thread.

Local id means the id of the thread inside the particular block and global id means there is a number of the thread combining all the blocks in the grid. So, you get the global id of the thread.

How we will discuss later. Then if id is less than N, because we are adding for 100000 numbers, we do this if id is greater than that we do not do that. So, there will be some threads which are

not executed. We do not do anything, but up to 100000 threads each thread will take one element of C because these threads have been unique id starting from 0 to 999999.

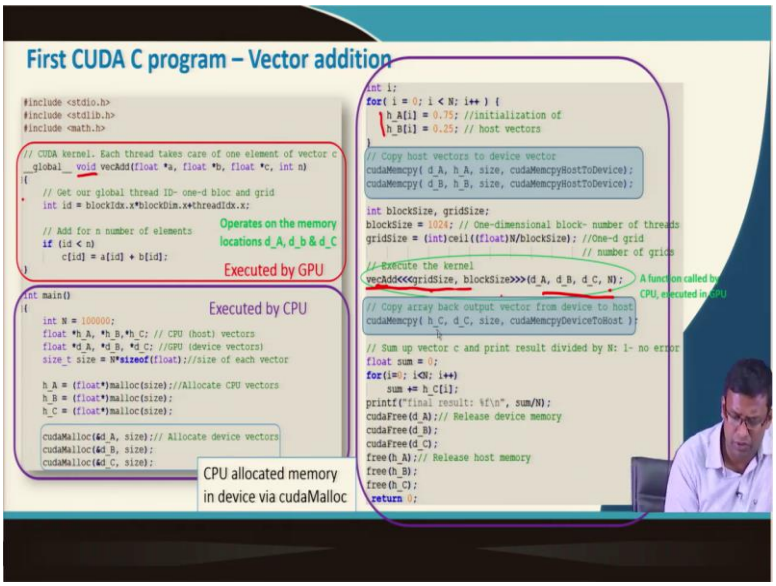
So, each thread will take one particular element of A and one particular element of B and add to be, C and therefore, they will write to the memory location d _ C in the GPU.

So, this kernel is executed in the GPU. Now the GPU variable is not readable to us, so we have to copy it back from the device from the GPU to the host, and the copying direction is the device to host. So, the which is copied the source comes later the destination comes first in CUDA mem copy.

Then just for correctness we see that if the sum is correct, if the sum is 1 for each element. So, if there are 100000 elements, the sum is 100000, we divide it by N and get 1 as that.

Then we have to free the device memory, is the same way we release the host memory, same way we can use CUDA free to free the device memory you cannot go into the GPU and free the memory; so, this is again executed by CPU and freed.

(Refer Slide Time: 36:22)



So, if we look into the program, these blocks are executed by the CPU. The kernel function is executed by GPU. What is this kernel function? This particular function is the kernel function which comes with this structure and syntax and this is executed by the GPU.

Allocation, CPU allocated allocates memory in the device via CUDA malloc. Allocation of GPU memories is also done by CPU. CPU copies vectors from host to device. So, vectors are initialized at host and then copy it from host to device, CPU to GPU using CUDA mem copy.

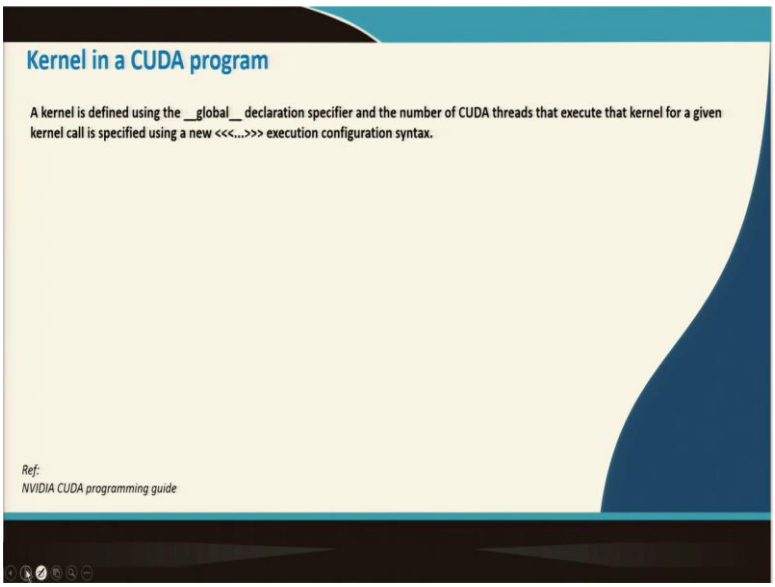
Then we call the kernel which is a function called by the CPU. This function vecadd is a function called by CPU with these arguments and there is a grid size, block size specified along with that it goes to the GPU.

What does it do? It operates on the memory locations of the variables of the arguments passed through the function. It operates at that memory location. Now, these memories are the GPU memories. A kernel cannot operate on the CPU memory, it only operates in the GPU memory.

A CPU cannot operate in the GPU memory, it can allocate GPU memories, it can free GPU memories, it can set the values by copying, but it cannot do any floating-point operation using GPU memory or logical operation using GPU memory. The CPU cannot do that, the main program cannot do that, a kernel only can change GPU memories, and that is done by the CUDA kernel.

Then the GPU memories are not readable by the CPUs otherwise, have to be copied back from CPU to GPU.

(Refer Slide Time: 38:00)



In the next class, we will look into the details of the kernels; what is its specification, what are the syntaxes, etcetera.