High Performance Computing for Scientists and Engineers

Prof. Somnath Roy
Department of Mechanical Engineering
Indian Institute of Technology, Kharagpur

Module – 04
GPU Computing
Lecture - 39
OpenACC programming for GPU-s

Welcome to the class of High-Performance Computing for Scientists and Engineers and we are in the last module of this course which is GPU computing. In the last few lectures, we looked in detail on CUDA programming and we have seen CUDA programming can give good performance which scales very well in GPU's for matrix vector products, matrix-matrix products, matrix solvers, type of scientific computing applications.

Well, now I want to show you another paradigm of GPU programming which is OpenACC based programming. Why is this important? Due to the fact that we are talking about scientific computing when looking into HPC. Scientific computing is mostly done by people who are working long; who have worked from long in domain sciences like physics, mechanical engineering, chemical engineering, biology, chemistry, materials etcetera and they have as expected usually limited exposure to computer science. When we look into GPU computing, we understand that there are several issues for which one needs to take care of the flow of instruction and flow of data from CPU to GPU and within the GPU itself.

The optimizations in GPU using CUDA require substantial expertise in that particular field. So, somebody when directly coming from a non-computer science background is a programmer on its own for the particular applications on which he is working, can find it difficult to work in CUDA and write CUDA parallel programs. This is one very serious issue and therefore, we will see in a while that multiple industrial organizations came together and thought of doing something for the people in scientific computing.

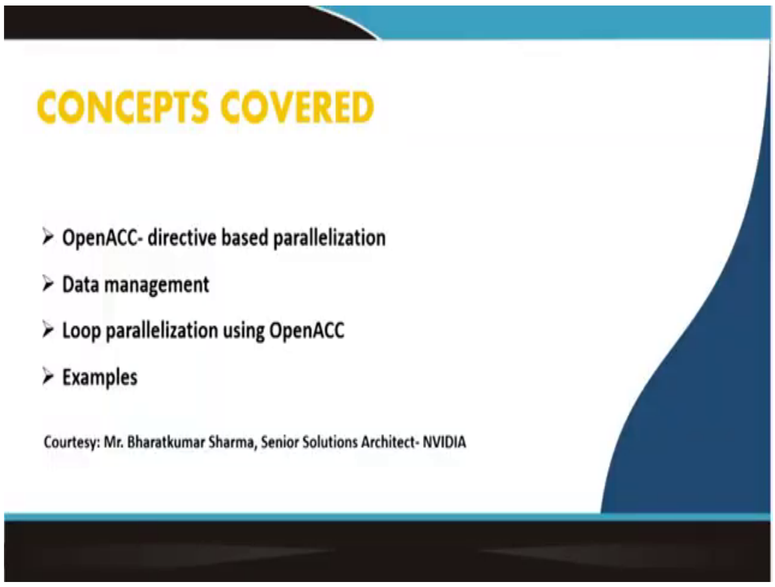
There are few important aspects for which I thought that introducing OpenACC or an easier way of developing accelerated programs can be important here. One is that as I understand many of us have limited expertise in terms of computer science or software-based applications. We mainly look into our domain specific application, so it will be easy for us to use something like OpenACC which requires less involvement in terms of data and instruction handling.

The next might be that many of our CS friends will actually look into the applications developed and maintained and utilized by people of non-CS background and see that they are using some languages or some a paradigm like this OpenACC through which they are paralyzing their program.

The third issue might be that many times you are not the developer of the program, but you get a program, and you have to use it. If you can get hold of a GPU code, you might like to parallelize it with your limited expertise. So, you can find OpenACC useful or you are working with a group who are using OpenACC or this type of parallelization because as I am mentioning every time that many times, people are challenged with their programming knowledge specially, GPU you we understand, require certain expertise that is that expertise is missing in that group. So, there can be many instances for which this is important.

As a matter of fact, I collaborated with a group from NVIDIA in Bangalore, India and their system analysts and engineers suggested to me that a module of OpenACC will be very vital in this course because they are seeing many scientific application communities migrating towards the OpenACC program. Well, with this background, we will see what OpenACC is.

(Refer Slide Time: 05:19)



CONCEPTS COVERED

- OpenACC- directive based parallelization
- Data management
- Loop parallelization using OpenACC
- Examples

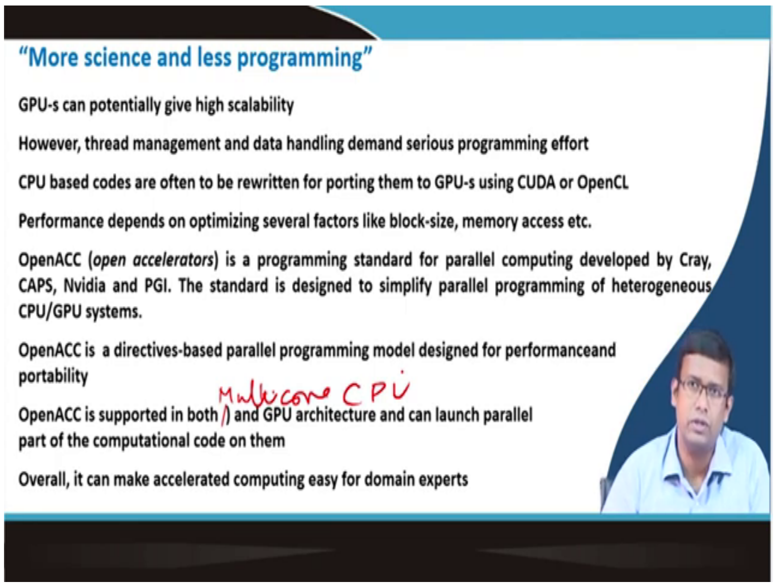
Courtesy: Mr. Bharatkumar Sharma, Senior Solutions Architect- NVIDIA

It uses a directive-based parallelization. So, I will see what is directive base parallelization, but you can think that it is very similar to OpenMP type of programming where only we are using the directives in that program, we are not calling any functions or not using any libraries only using the directives.

How can we do data management in OpenACC? Because we understand that in GPU, calling data from CPU RAM to GPU RAM, copying data and copying it back is important, so what do you do for that? How can we parallelize a loop using OpenACC? Because most of the scientific computing applications depend on iterative loops, how can we parallelize it and we will see some of the examples.

Mister Bharatkumar Sharma as I mentioned him, he is a Senior Solutions Architect from NVIDIA and many of the slides I will show have direct contribution from his end and he is one of the key people who suggested that OpenACC must be a part of this discussion.

(Refer Slide Time: 06:22)



"More science and less programming"

GPU-s can potentially give high scalability

However, thread management and data handling demand serious programming effort

CPU based codes are often to be rewritten for porting them to GPU-s using CUDA or OpenCL

Performance depends on optimizing several factors like block-size, memory access etc.

OpenACC (*open accelerators*) is a programming standard for parallel computing developed by Cray, CAPS, Nvidia and PGI. The standard is designed to simplify parallel programming of heterogeneous CPU/GPU systems.

OpenACC is a directives-based parallel programming model designed for performanceand portability

OpenACC is supported in both Multicore CPU and GPU architecture and can launch parallel part of the computational code on them

Overall, it can make accelerated computing easy for domain experts

So, if you look into the OpenACC forum, what they try to assert is more science less programming. A person working in a science engineering domain will be more focused towards science less focused toward programming and that is what the community expects from him.

So, you should spend less effort in writing the program or developing the program and thinking how can I bring certain data from device memory, what should be my compute global memory access ratio, where should how should it be mapped to the to the shared memory, how it can be accessed by different threads, what should be my optimum block size these things we expect a scientist to less worry about these things so, that he can do more better science. I mean this is inevitable for getting better science, for getting accurate solutions he has to learn large problems and looking into GPU's might be inevitable.

But while doing that, he must be less bothered about parallelization efforts rather you should be able to spend more time about science and use more of the developed tools in his domain.

So, we know GPU's can give very high scalability. We have seen 90 times scalability in some of the matrix computing examples in last class. However, thread management and data handling demand serious programming effort. With a very small background like in OpenMP with a very initial background, one can jump and start converting his program, but in a CUDA program, if you have your legacy code, it is very difficult to directly convert it into a GPU code. You have to identify the portions which can be parallelized, you have to do the right thread algebra, many things are there to copy data from GPU to CPU etcetera, allocate variables, many operations have to be done.

So, CPU based codes are not very easily ported into GPU's like MPI and OpenMP. You can start with the basic C code and then start modifying it, very soon you can develop a parallel code. But in a CUDA GPU code, the CPU based codes cannot be very easily ported into GPUs using CUDA or OpenCL. You may have to rewrite the code and some of the legacy codes we can think about are millions in line size. So, rewriting the entire code might be very difficult. Another thing is that performance depends on optimizing several factors like block size, memory access, shared memory and many issues. So, if people from different industrial organizations working on GPU and accelerator based HPC, thought of proposing something like OpenACC or open accelerator where the compiler will take care of the parallelization efforts.

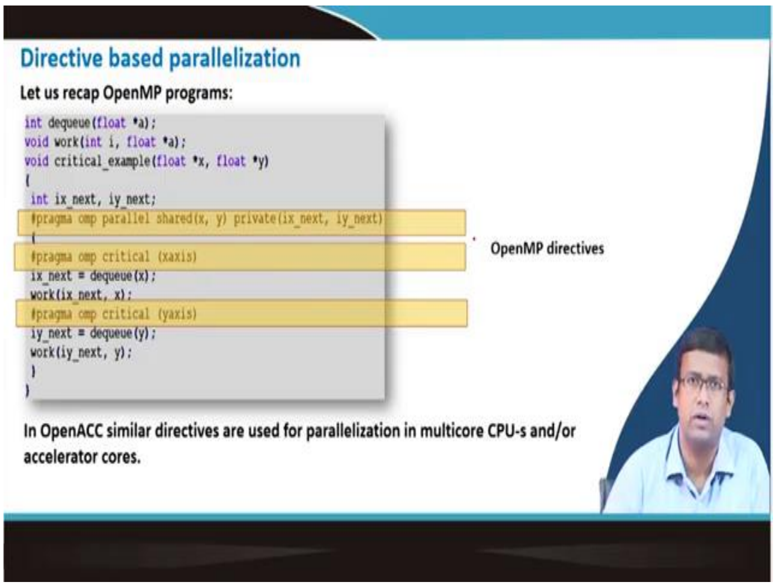
Programmer has to only identify which part of his program he wants to parallelize and the compiler itself will take care of that. So, this programming standard developed by Cray, CAPS, Nvidia, PGI and the standard is designed to simplify the parallel programming in heterogeneous CPU-GPU systems.

This is a directive based parallel programming model designed for performance and portability. So, with less effort, you can get optimized performance that is one aspect and portable means you can take it from across different platforms. You can run on a multi-core system, you will get good performance, you run on a Nvidia GPU, you will get good performance, you go to an Intel multi-core system you will get good performance, you go to an AMD GPU, you get good performance all these things have been taken care of. The compiler will see which hardware it is working on and look for that performance.

Nvidia supports both multi-core CPU and GPU architecture and can launch parallel parts of the computational code in either of them and overall, it can make accelerated computing easy for domain experts.

The caveat is there, this is for shared memory system, this is for single instruction multiple data type of model parallelization, for distributed memory system you have to use MPI, OpenACC cannot be used that, but it can replace some of the functionalities of OpenMP because it works on shared memory system.

(Refer Slide Time: 11:21)



Directive based parallelization

Let us recap OpenMP programs:

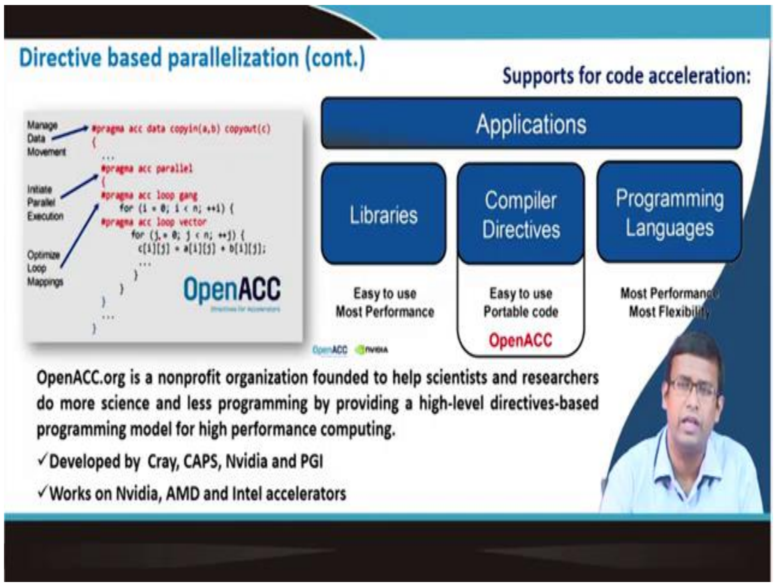
```
int dequeue(float *a);
void work(int i, float *a);
void critical_example(float *x, float *y)
{
 int ix_next, iy_next;
 #pragma omp parallel shared(x, y) private(ix_next, iy_next)
 {
 #pragma omp critical (xaxis)
 ix_next = dequeue(x);
 work(ix_next, x);
 #pragma omp critical (yaxis)
 iy_next = dequeue(y);
 work(iy_next, y);
 }
}
```

OpenMP directives

In OpenACC similar directives are used for parallelization in multicore CPU-s and/or accelerator cores.

So, we let us recap the OpenMP program because we talk about directive-based parallelization and we can recall the directives we have used in the OpenMP program. So, if we see that in particular location, we want to parallelize that use this shared and private memory and then, some of the work will be done in a critical manner that each thread will do on its own, there will be a sequentiality among them. So, this region will be parallelized, this region will be critical; this is done by OpenMP detectives right. These statements #pragma or exclamation OMP for Fortran or #pragma for C at the OpenMP directives. In OpenACC similar directives are used for parallelization in multicore CPU's and or accelerator cores or GPU cores.

(Refer Slide Time: 12:16)



Directive based parallelization (cont.)

```
#pragma acc data copyin(a,b) copyout(c)
{
  ...
  #pragma acc parallel
  {
  #pragma acc loop gang
      for (i = 0; i < n; ++i) {
  #pragma acc loop vector
        for (j = 0; j < n; ++j) {
          c[i][j] = a[i][j] + b[i][j];
          ...
        }
      }
  }
  ...
}
```

Manage Data Movement
Initiate Parallel Execution
Optimize Loop Mappings
OpenACC
Supports for code acceleration:
Applications
Libraries
Compiler Directives
Programming Languages
Easy to use Most Performance
Easy to use Portable code
OpenACC
Most Performance Most Flexibility

OpenACC.org is a nonprofit organization founded to help scientists and researchers do more science and less programming by providing a high-level directives-based programming model for high performance computing.

✓Developed by Cray, CAPS, Nvidia and PGI

✓Works on Nvidia, AMD and Intel accelerators

So, let us look into directive-based parallelization in a little more detail. If you have an application which has to be parallelized, application means the program which has to be parallelized, then there are three general methods for that one is calling parallel libraries. Say you have a particular scientific computing example where there is a matrix solver, you call a hyper library which has inbuilt parallel matrix solver function. So, you call that function.

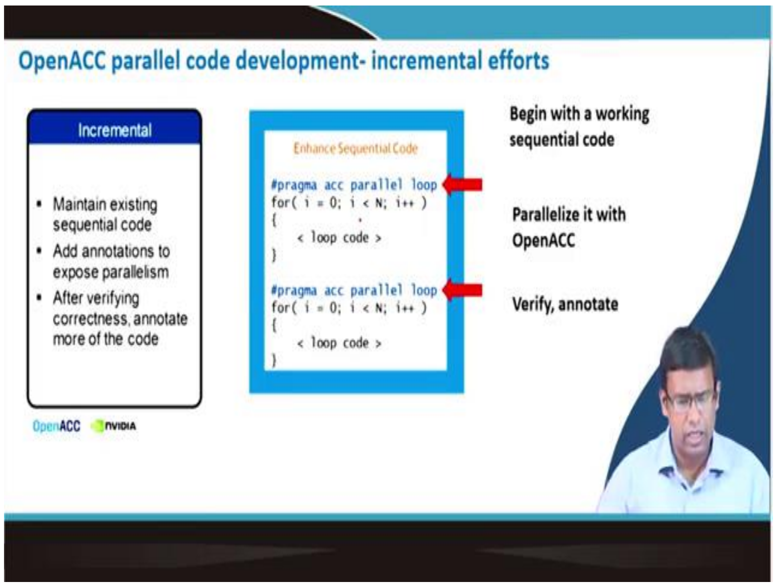
You can use compiler directives like OpenMP we have seen, open OpenACC is also there or you can use programming languages say you are using OpenCL that is a programming language for parallelization. It gives you very good performance, it is very flexible; however, you need to learn the language, you need to learn the syntax and semantics.

In the library it is easy to use and usually get good performance also, but it is not much flexible there, there are certain restrictions on that. But in between that you can use compiler directives which are easy to use and you can have your own coding library, you have to use their library function which has been built by somebody else, but you can use your own code and parallelize it using the compiler directives. Compiler will read the directive and do the part of parallelization. So, you have to only have a direct compiler that takes this part and parallelize it accordingly.

So, if we look into an OpenACC program, we can see different directives like #pragma, acc, data, copy in, copy out. This is a directive for data movement. How data will go from CPU to GPU and GPU to CPU etcetera. #pragma acc parallel compiler knows that this zone will be parallelized. Then, #pragma acc loop gang, loop vector these are different ways of parallelization of the iteration loops. So, these are the directives which you put inside your older CPU code and get a parallelized code.

OpenACC. org presents this set of directives, this is a non-profit organization which in which many companies are there and they help scientists to do more science and less programming for HPC and this platform is developed by Cray, CAPS, Nvidia, PGI and this OpenACC which we are talking about this particular directive base parallelization works on number of platforms including Nvidia and AMD GPU and Intel multicores, Intel accelerators etcetera ah.

(Refer Slide Time: 15:08)



OpenACC parallel code development- incremental efforts

Incremental

- Maintain existing sequential code
- Add annotations to expose parallelism
- After verifying correctness, annotate more of the code

Enhance Sequential Code

```
#pragma acc parallel loop
for( i = 0; i < N; i++ )
{
    < loop code >
}

#pragma acc parallel loop
for( i = 0; i < N; i++ )
{
    < loop code >
}
```

Begin with a working sequential code

Parallelize it with OpenACC

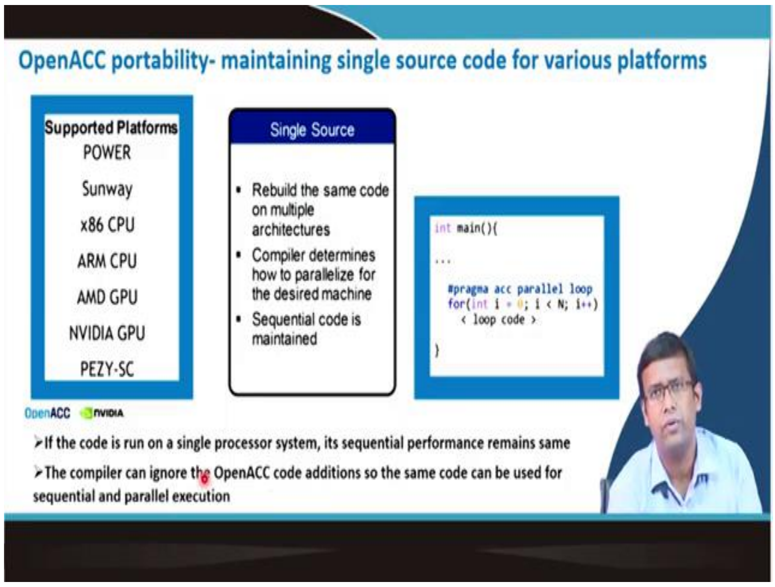
Verify, annotate

So, the efforts if we see in parallelizing an OpenACC job, we call them incremental efforts. So, is its incremental effort? You have your older serial code, then you can, you have to put only the OpenACC directives and then, you can add annotations to expose that they are parallelized only introducing these directives will parallelize the code.

So, you take your older CPU code, look into the parts which can be when parallelized and start putting the directive. After putting a directive in one particular region, you see that you are getting the right results. You can run into a single processor, it will run like this older code, if you run it as a multiprocessor or GPU, it will run as an accelerated code, but the results must be the same. Check the results. So, once you check that this part is correct, you understand that this part has been parallelized and go to the next part.

So, because of this type of programs, there are fork-joint models of the threads each fork you look into and parallelize that. Begin with a working sequential code, parallelize it with OpenACC and then verify and annotate that part and go to the next block parallelize it.

(Refer Slide Time: 16:22)



OpenACC portability- maintaining single source code for various platforms

Supported Platforms
- POWER
- Sunway
- x86 CPU
- ARM CPU
- AMD GPU
- NVIDIA GPU
- PEZY-SC

Single Source
- Rebuild the same code on multiple architectures
- Compiler determines how to parallelize for the desired machine
- Sequential code is maintained

```
int main(){
...
  #pragma acc parallel loop
  for(int i = 0; i < N; i++)
    < loop code >
}
```

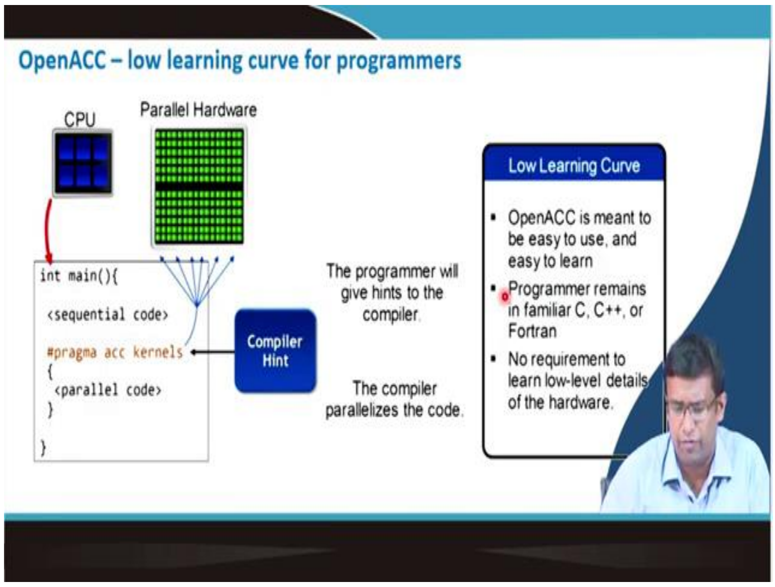
- If the code is run on a single processor system, its sequential performance remains same
- The compiler can ignore the OpenACC code additions so the same code can be used for sequential and parallel execution

That is why we call that an incremental way of parallelizing; look into each part and parallelize it well. Then, it is portable, you develop a code using OpenACC and that code will run on all of these platforms. So, once you develop OpenMP code, it will run in all of these platforms and in each platform, like a multicore CPU platform like x86 CPU or we go to an AMD GPU. They are different platforms.

But the compiler will identify what is the hardware, what is the architecture and how data should move, how the kernel should be launched, what should be done for the right parallelization. So, this will be left on the compiler and it itself will do that optimization. If you run on a single processor, the sequential code will be maintained, it will give you the sequential results. So, that is why it is a portable program also.

If the code is run on a single processor system, the sequential performance remains the same. The compiler can ignore OpenACC code additions so that the same code can be used for sequential and parallel execution. So, when you run on a sequential platform or when you run a platform where some of the parallel drivers in terms of hardware or software and missing compiler simply ignores the pragma line and runs it as your old sequential code. So, you do not need to maintain multiple codes for different platforms.

(Refer Slide Time: 17:58)


OpenACC – low learning curve for programmers

```
int main(){
 <sequential code>
 #pragma acc kernels
 {
  <parallel code>
 }
}
```

The programmer will give hints to the compiler.

The compiler parallelizes the code.

Low Learning Curve

- OpenACC is meant to be easy to use, and easy to learn
- Programmer remains in familiar C, C++, or Fortran
- No requirement to learn low-level details of the hardware.

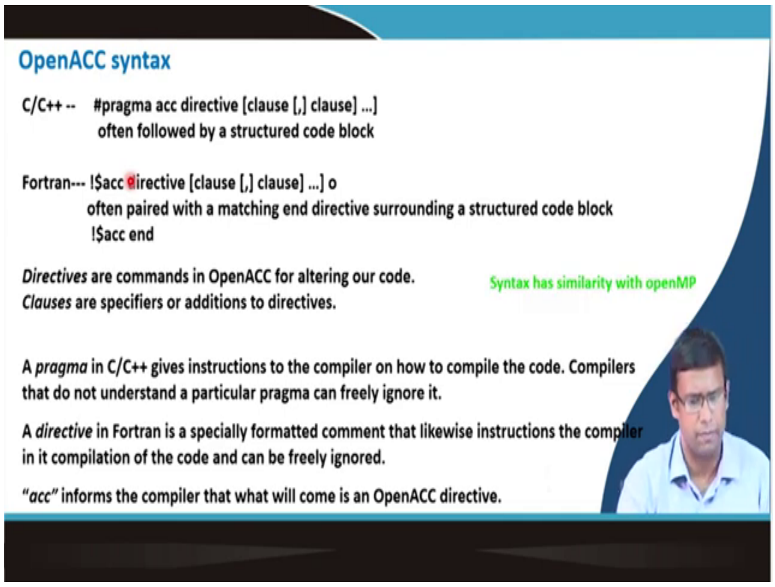
It has a low learning curve. Low learning curves means you say you have written you have identified a parallel zone; you have written #pragma acc kernels. This directive tells the compiler that this has to be parallelized. How will it be parallelized? You do not need to know about that. It is a starting point. You can do certain things to improve the performance, but to start with, you do not need to know about that compiler will take care of that. So, it is easy to use and learn.

You only need to know a few directives; you have to identify that this location can be parallelized, you have to have some basic idea of parallel computing and call these directives and the compiler gets the hint and works with that. Your knowledge in C, C ++ Fortran is sufficient. You do not need to know low level details of the hardware as well as how to handle managing those things. You do not need to know the details which you know for CUDA programming, you no need to go for that.

So, one of the issues might be that you may not get the best performance because you are leaving everything on the compiler and the compiler has been optimized for certain general applications; for your specific applications you may not get the best performance. So, for that, you will may have to do some other optimization or in some point of time, you may not get the best performance using OpenACC we, if we compare in many cases, we can see CUDA can give you better performance or OpenCL can give you better performance because you have more flexibility from programmers point of perspective there.

But you can get reasonably good performance with less effort in some time, if you are not very efficient in CUDA programming, you can do things in a wrong manner, but in OpenACC as the compiler is taking care of everything, it will give you reasonably good performance.

(Refer Slide Time: 19:55)



OpenACC syntax

C/C++ -- #pragma acc directive [clause [,] clause] ...]
often followed by a structured code block

Fortran--- !$acc directive [clause [,] clause] ...] o
often paired with a matching end directive surrounding a structured code block
!$acc end

Directives are commands in OpenACC for altering our code.
Clauses are specifiers or additions to directives.

Syntax has similarity with openMP

A *pragma* in C/C++ gives instructions to the compiler on how to compile the code. Compilers that do not understand a particular pragma can freely ignore it.

A *directive* in Fortran is a specially formatted comment that likewise instructions the compiler in it compilation of the code and can be freely ignored.

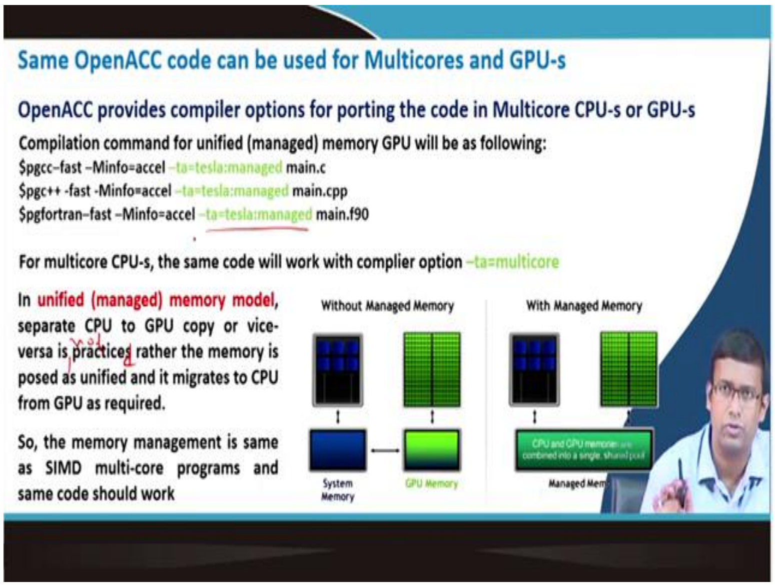
"acc" informs the compiler that what will come is an OpenACC directive.

So, OpenACC syntax is in C, it is #pragma acc then directive with the clauses and then it is followed by the structured code block in many cases. In Fortran! $ acc directive and clause and then, this is paired with a matching directive surrounding a structured block and, in many cases, you show that the directive is being ended by a dollar acc and inside this whatever you put will follow the directive.

Directives are commands in OpenACC for altering our code. Clauses are specified inside the directive or additions to the directive. We will see some of the clauses later. A pragma in C, C ++ gives instruction to the compiler how to compile the code. Compilers that do not understand the particular pragma can freely ignore it. It will not give you an error or a bug. If the compiler is unsure about a pragma, it will ignore it and the remaining part if you take away this pragma or if you take away these two lines, the remaining part is your sequential program that will operate in that manner. So, they are very optimized to give you error free outputs, but that can hinder the performance. If the compiler is ignoring, the application engineer may not understand that the compiler is ignoring that part, but for CUDA or even for OpenMP, if it is ignoring you will get an error message there.

A directive in Fortran is specially formatted. So, the specially formatted comment starts with an exclamation. So, if it is if the compiler does not understand it, it is an explanation, it will be ignored by the simple Fortran compiler. Acc informs the compiler; both case acc is present that it is an OpenACC directive. Similarly, you know open OpenMP you have seen OMP is present there ah.

(Refer Slide Time: 21:54)



Same OpenACC code can be used for Multicores and GPU-s

OpenACC provides compiler options for porting the code in Multicore CPU-s or GPU-s

Compilation command for unified (managed) memory GPU will be as following:

```
$pgcc–fast –Minfo=accel –ta=tesla:managed main.c
$pgc++ -fast -Minfo=accel –ta=tesla:managed main.cpp
$pgfortran–fast –Minfo=accel –ta=tesla:managed main.f90
```

For multicore CPU-s, the same code will work with complier option –ta=multicore

In unified (managed) memory model, separate CPU to GPU copy or vice-versa is not practiced rather the memory is posed as unified and it migrates to CPU from GPU as required.

So, the memory management is same as SIMD multi-core programs and same code should work

OpenACC provides compiler options for porting the code from multicore CPU-s and GPU-s and this is very important; this is a PGI compiler instance is shown and PGI is one of the developer groups of OpenACC. So, when you run this, you put the compiler option -ta = tesla managed. It shows that when this will be compiled, this will be compiled for tesla is a GPU Nvidia tesla GPU card compiled for a GPU card with managed or uniform memory access data.

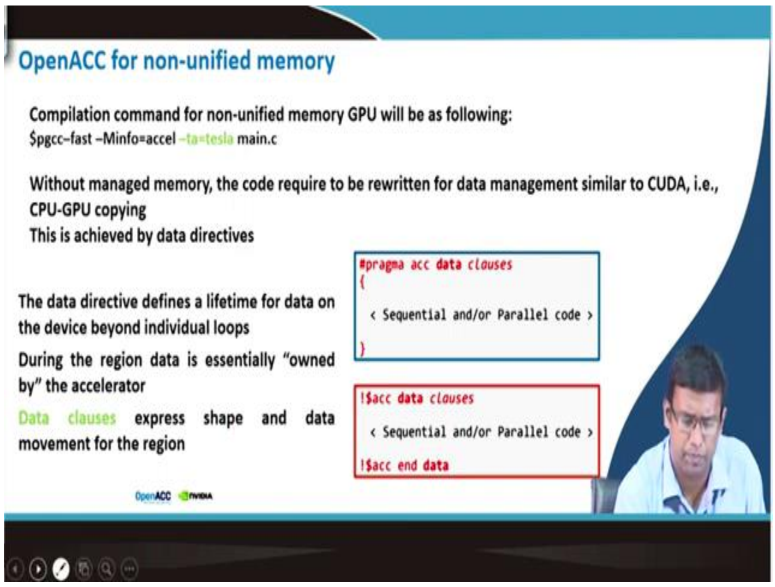
If you write a compiler option - ta multicore, the compiler will know that it is compiling it for multicore CPU-s. So, just by giving the compiler option, the same code that can be compiled for different jobs. It is a unified or managed memory access model because we are using tesla managed, separate CPU to GPU copy or vice versa is not practiced.

So, data is not copied separately from CPU to GPU or GPU to CPU rather we consider a unified memory access that the entire data is present in somewhere combining CPUs and GPUs and whichever host or device will require the data will migrate towards that.

You remember at the beginning of discussions on GPU, we talked about unified memory access. So, by using tesla managed, we can use unified memory access. So, this is a general CPU-GPU access, there is a CPU RAM, there is a GPU RAM and data is copied in between them. In a managed memory or unified memory access, there is a single combined memory and data is migrated towards CPU and GPU based on the application. So, that also you can do by providing the compiler access that it will consider a GPU and considered as a unified memory access GPU.

So, the memory management is the same as SIMD multicore programs and you develop a program for multicore OpenMP type programs using OpenACC for multicore systems using SIMD model. The same program just with changing OpenACC program not an OpenMP program, for OpenMP you can also use GPU but some other features are included, but if you develop a OpenACC program multicore program, the same program can run. If you write - a multicore with the compiler option, it will be built for running in a multicore system. If you write - ta tesla managed, it will run in GPU's.

(Refer Slide Time: 24:38)



OpenACC for non-unified memory

Compilation command for non-unified memory GPU will be as following:
$pgcc–fast –Minfo=accel –ta=tesla main.c

Without managed memory, the code require to be rewritten for data management similar to CUDA, i.e., CPU-GPU copying
This is achieved by data directives

The data directive defines a lifetime for data on the device beyond individual loops

During the region data is essentially "owned by" the accelerator

Data clauses express shape and data movement for the region

```
#pragma acc data clauses
{
  < Sequential and/or Parallel code >
}
```

```
!$acc data clauses
  < Sequential and/or Parallel code >
!$acc end data
```

If you want to run this in a non-unified memory system like the memory systems, we considered in our previous CUDA examples CPU has a different memory, GPU has a different memory and data will be copied from CPU to GPU or GPU to CPU as required just use the compiler command - ta = tesla. Tesla managed will give you unified memory access; simple tesla will give you non-unified memory access. In CUDA also, you can use unified memory access in the recent versions of CUDA. We have not discussed that, but you can use that you can look into CUDA C programming guide and find that.

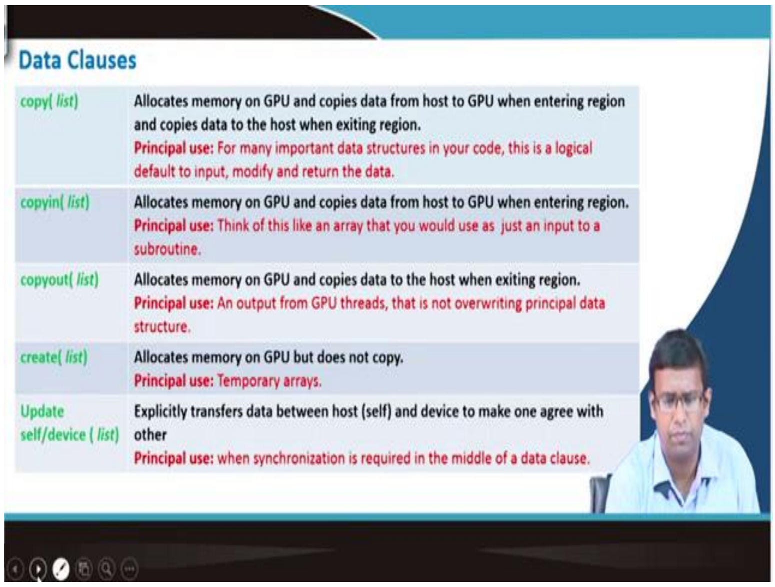
Well in OpenACC in unified memory accesses are important because then the same code you do not need to copy from device to host or host to device, it is like a unified memory or managed memory case and the same code can go to both multicores and GPUs.

In a multicore system, especially in unified memory access or UMA multicore systems there is a single RAM which all the cores are accessing. So, if you are using managed or unified memory, it is working in a similar manner.

Without manage memory, code required to be rewritten for data management similar to CUDA that is CPU to GPU copying and this CPU to GPU copying is obtained by data directive that you write pragma acc data with the clause that CPU to GPU or GPU to CPU how the copying will be done using this directive, you can copy the data. We will see some of the examples soon.

The data directive defines the lifetime for data on devices beyond individual loops. During the region, the data is essentially owned when we give in the data directory within this region, any data we talk about is owned by the accelerator it is on the GPU from that. If using an unmanaged system, then you have to copy from CPU to GPU and within the directive, it will be in the GPU and then we can copy back to from GPU to CPU. Data clauses, the clauses express the direction of movement and shape and size of the data.

(Refer Slide Time: 26:54)



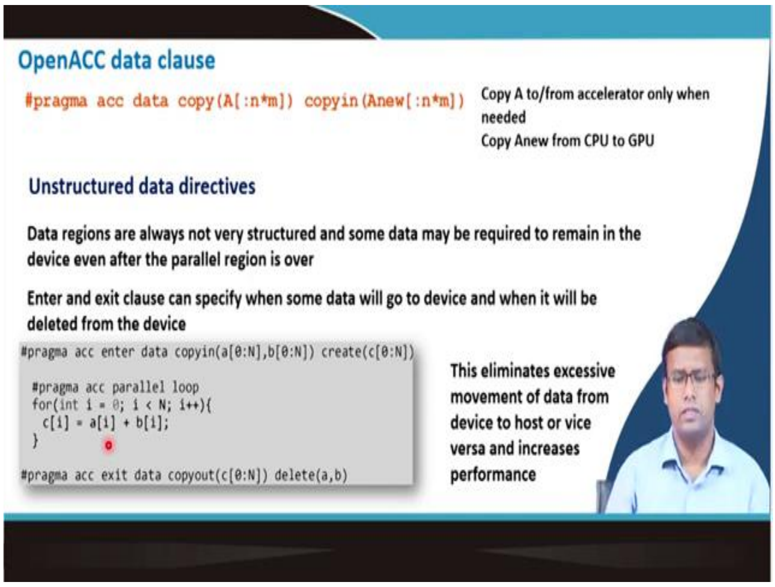
Data Clauses

copy(*list*)	Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region. **Principal use:** For many important data structures in your code, this is a logical default to input, modify and return the data.
copyin(*list*)	Allocates memory on GPU and copies data from host to GPU when entering region. **Principal use:** Think of this like an array that you would use as just an input to a subroutine.
copyout(*list*)	Allocates memory on GPU and copies data to the host when exiting region. **Principal use:** An output from GPU threads, that is not overwriting principal data structure.
create(*list*)	Allocates memory on GPU but does not copy. **Principal use:** Temporary arrays.
Update self/device (*list*)	Explicitly transfers data between host (self) and device to make one agree with other **Principal use:** when synchronization is required in the middle of a data clause.

These are the clauses. Copy tells that memory is allocated on GPU and data will be copied from CPU to GPU and GPU to CPU as required. When copying from CPU to its GPU will be done, then will be CPU to GPU during entering the parallel part, after the parallel part it will be copied from GPU to CPU. So, both CudaMemcpy host to device, device to host is taken care of. Copyin says it will be copied from CPU to the GPU. It will do CUD Malloc allocate the memory as well as it will do the copy in. Copyout will do GPU to CPU copy along with allocation.

Create we will just do the allocation part. Then, sometimes we use an update command which is like a synchronization command. If there are multiple iteration loops, the data which is lying on the CPU and on the GPU, it is required that they are synchronized, so use update that it considers one of the data and checks that it agrees with the CPU-GPU data. So, whatever part is not copied, it copies it there and they have their utility for some of the examples about synchronization you need to update, if you need to use an input from CPU unit copyin and so on. We can observe their similarity with CudaMemcpy and CUDAMalloc.

(Refer Slide Time: 28:24)



OpenACC data clause

```
#pragma acc data copy(A[:n*m]) copyin(Anew[:n*m])
```

Copy A to/from accelerator only when needed

Copy Anew from CPU to GPU

Unstructured data directives

Data regions are always not very structured and some data may be required to remain in the device even after the parallel region is over

Enter and exit clause can specify when some data will go to device and when it will be deleted from the device

```
#pragma acc enter data copyin(a[0:N],b[0:N]) create(c[0:N])

 #pragma acc parallel loop
 for(int i = 0; i < N; i++){
   c[i] = a[i] + b[i];
 }

#pragma acc exit data copyout(c[0:N]) delete(a,b)
```

This eliminates excessive movement of data from device to host or vice versa and increases performance

The data clause in OpenACC is pragma acc data copy, copyin and copy A of size a has size n*m and Anew has size n*m; that means, A will be copied to accelerator when needed and copied back from the accelerator when the part is done and Anew will be only copied from the CPU to the GPU. GPU is the accelerator.

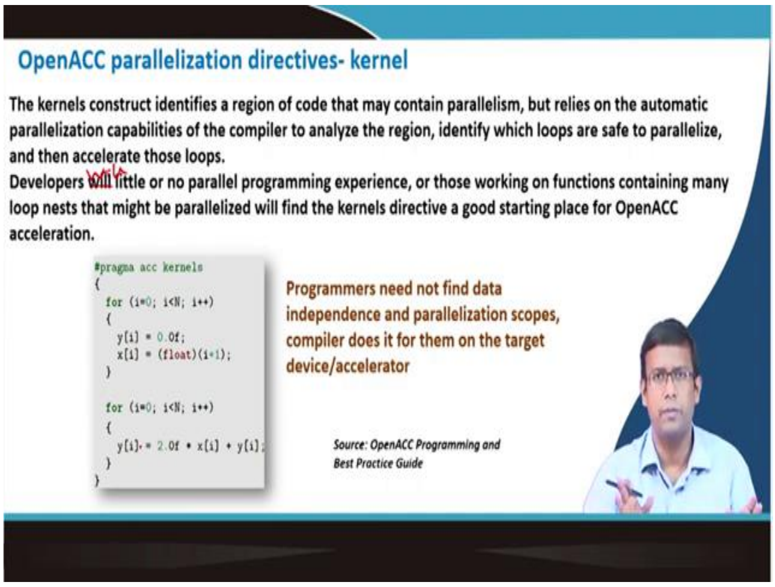
Now, this copy is defined only for the parallel region, but we might require the data even outside the parallel region and may be again the parallel another parallel region will be done and the similar variable will be operated, but if we say that was available only for the previous parallel region, again we have to copy the data, copying from CPU to GPU is the costly affair.

So, we can set that when this will be copied to the GPU and then we can say that it is flushed off, it is deleted from the GPU. These terms we also can set using directives and data regions are always not very structured and some data may be required to remain in the device even after the parallel region is over.

Enter and exit clause with the data directive can help us enter the clause that will tell that data will go in the GPU and stay there. Exit will say that now this data is not required, GPU will erase the data. So, this is like pragma acc enter data copyin so, a, b is copied into GPU and they will stay in the GPU and c is created in the GPU, c will also stay in the GPU as long as we require it and then we write c = a+b. So, a, b is de-allocated from the GPU, GPU does not store any part of a and b, c still receives copy from CPU to GPU, but yeah so, a, b, c all erased from the GPU memory.

But if we run another loop here, with another parallel loop here and few more parallel loops run a sequential loop etcetera here, the a, b and c will still remain in the GPU that GPU copy will be unaltered. This eliminates excessive movement of data from device to host and vice versa and therefore, increases performance.

(Refer Slide Time: 30:48)



OpenACC parallelization directives- kernel

The kernels construct identifies a region of code that may contain parallelism, but relies on the automatic parallelization capabilities of the compiler to analyze the region, identify which loops are safe to parallelize, and then accelerate those loops.

Developers ~~will~~ with little or no parallel programming experience, or those working on functions containing many loop nests that might be parallelized will find the kernels directive a good starting place for OpenACC acceleration.

```
#pragma acc kernels
{
  for (i=0; i<N; i++)
  {
    y[i] = 0.0f;
    x[i] = (float)(i+1);
  }

  for (i=0; i<N; i++)
  {
    y[i] = 2.0f * x[i] + y[i];
  }
}
```

Programmers need not find data independence and parallelization scopes, compiler does it for them on the target device/accelerator

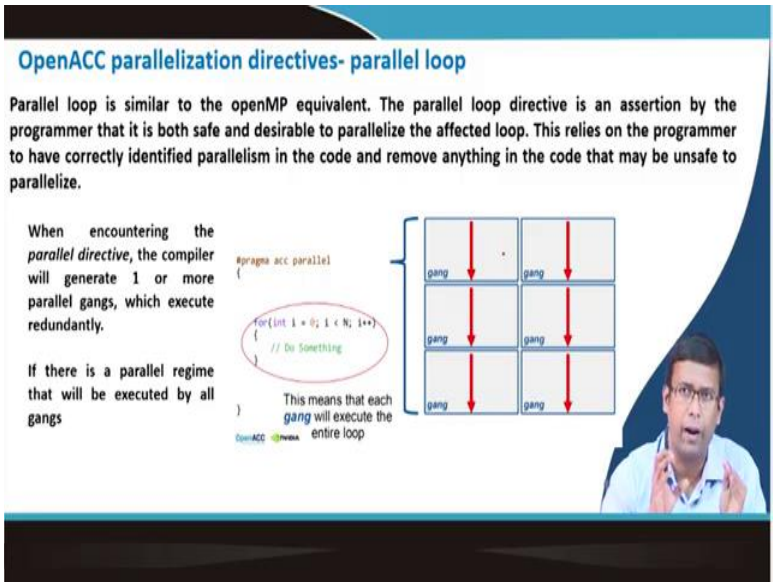
Source: OpenACC Programming and Best Practice Guide

OpenACC parallelization directive, when we have seen earlier the simplest parallelization directive is kernel. If we can identify a particular region which has to be parallelized and write the pragma acc kernel, this construct will identify this region that takes that region of the code and check if it thinks that it may contain parallelism. This parallelization is done by the programmer, the directive is given by the programmer, but it is left on the compiler to parallelize it. Compiler will check that whether there is sufficient data independence and iterations are there, it considers it for the parallelization and then, if it checks that there is sufficient data independence, then it will automatically parallelize it. So, the compiler will analyse the region, identify which loops are safe to parallelize and then accelerate those loops.

Developers with little or no parallel programming examples or who are working on functions that contain many loops which you cannot go in individual loops and parallelize; you can find this kernel directives a good starting place for OpenACC acceleration. Pragma OpenACC kernels and there are many parallel loops it will try to see which of the loops can be parallelized here.

Programmers need not find data independence and parallelization scopes; compilers will do it for the target device or accelerator. So, you just write pragma acc kernel, it will take care of the rest.

(Refer Slide Time: 32:23)



OpenACC parallelization directives- parallel loop

Parallel loop is similar to the openMP equivalent. The parallel loop directive is an assertion by the programmer that it is both safe and desirable to parallelize the affected loop. This relies on the programmer to have correctly identified parallelism in the code and remove anything in the code that may be unsafe to parallelize.

When encountering the *parallel directive*, the compiler will generate 1 or more parallel gangs, which execute redundantly.

If there is a parallel regime that will be executed by all gangs

```
#pragma acc parallel
{

  for(int i = 0; i < N; i++)
  {
     // Do Something
  }

}
```

This means that each *gang* will execute the entire loop

Parallel loop is similar to the OpenMP equivalent. Instead of a kernel, you can use parallel. Parallel gives you more flexibility and more liability as more responsibility as a program that you have to identify that this is a parallelizable loop and you have to tell how the program will parallelize it.

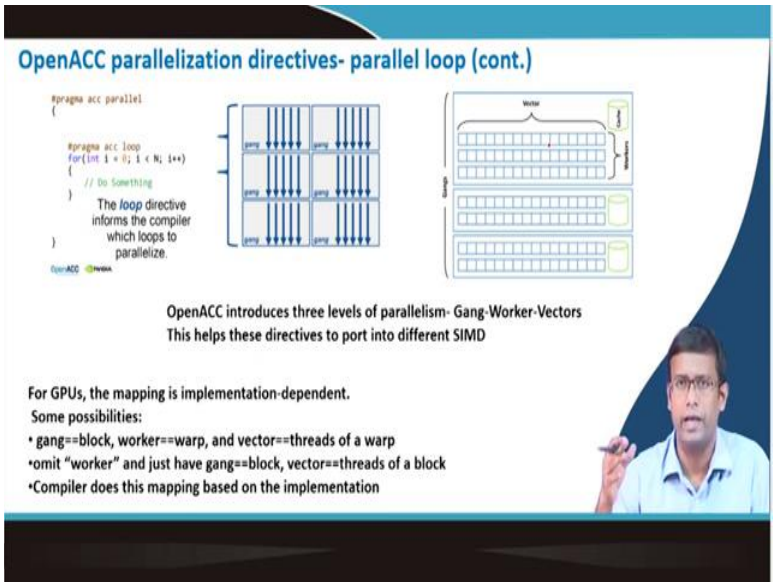
Parallel loop is an assertion by the programmer that this is safe and we desire it to be parallelized and then the compiler will parallelize it. In a kernel compiler programmer may not know that if it can be parallelized. Programmer thinks that it is a compute heavy zone; ask the compiler to parallelize it. In case of; parallel, programmer has to be sure that this has to be parallelized and also programmer has certain flexibility in terms of how that can be parallelized.

When encountering the parallel directive, the compiler will do something. Because we think of parallelizing in accelerators and GPU's, the compiler has something in mind. In a parallel region, it is known to the compiler that this is for sure to be parallelized. It will try to do some equivalent of the blocks, threads etcetera. So, what will it do? In the parallel regime, it will launch many gangs.

Each of the gangs will do exactly the same work, but now if there is an iteration and we ask this to be a parallel for loop, then each gang will take part of the iterations and operate over the number of iterations. So, it is something like what each block is doing, and each block is operating a number of threads. Similarly, the gangs will execute that.

There will be a parallel regime, if there is a parallel regime that will be executed by all the gangs. If there is a parallel for regime, then each gang will take part of the iterations and do it.

(Refer Slide Time: 34:18)



OpenACC parallelization directives- parallel loop (cont.)

```
#pragma acc parallel
{

    #pragma acc loop
    for(int i = 0; i < N; i++)
    {
        // Do Something
    }
}
```

The *loop* directive informs the compiler which loops to parallelize.

OpenACC introduces three levels of parallelism- Gang-Worker-Vectors
This helps these directives to port into different SIMD

For GPUs, the mapping is implementation-dependent.
Some possibilities:
- gang==block, worker==warp, and vector==threads of a warp
- omit "worker" and just have gang==block, vector==threads of a block
- Compiler does this mapping based on the implementation

Say we have a OpenACC parallel, pragma acc parallel and then pragma acc loop; that means, this for loop has to be parallelized, then each gang will take some iterations of the loop and run them in parallel. So, if we look into OpenACC, this has its own paradigm of parallelization, its own hierarchy.

The gang is the largest unit of parallelization. There will be multiple gangs, each gang working in parallel. Inside each gang, there are workers. Workers are the next group of parallelization which has all the workers share the same cache. Inside the worker, there are many vectors and vectors are the units of parallelization.

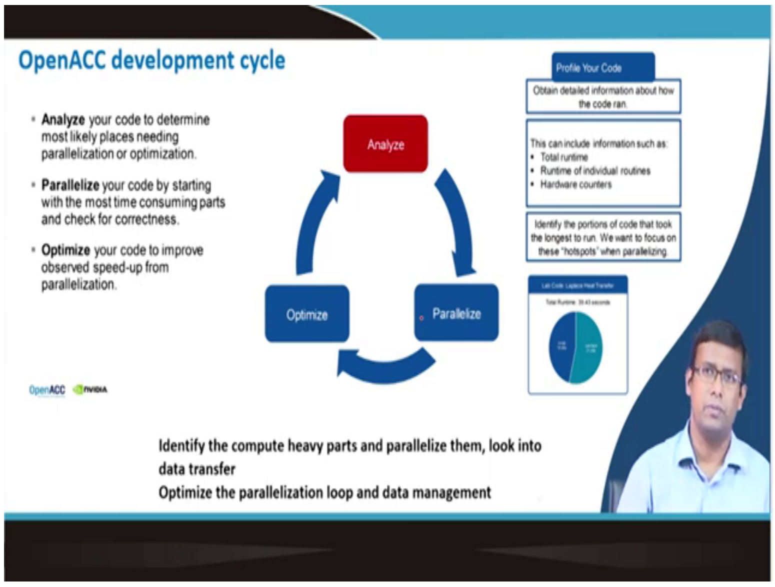
OpenACC introduces these three levels of parallelization: gang, worker, vector. This helps the directive to port into different SIMD architecture starting from multicores to GPU's. For GPU's, the mapping is implementation dependent. Some cases, a gang is the block, if there are going to multiple streaming multiprocessors, each streaming multiprocessor gets a gang and worker is the warp inside the gang which is operated at once and there can be multiple warps inside the gang like multiple warps inside the block, there can be multiple workers inside the gang and vectors are the threads of the warp.

In some cases, we do not need to think about the warp, we just look into the gang and then the vector. So, based on the implementation OpenACC does this parallelization of how it will assign workers or whether it will assign workers or not.

What will be the block size? What will be the thread size? If you do not specify anything, you can do something on specifying the gang and loop, where the gang will be activated, where the vectors will be activated, what is the unit of the workers, you can do something on the specification in OpenACC.

You can look into a OpenACC best practice guide and find that, but if you do not want to do that, OpenACC will itself do it. If you do not specify any block size, by default grid size is 1 and the total number of threads are block size. If you do not specify any block size, it will go to only 1 block, 1 thread. But if you do not specify anything on gangs loops, vectors, OpenACC will do the optimization and run it as the number of gangs and vectors it thinks efficient and optimized for that application.

(Refer Slide Time: 37:04)



OpenACC development cycle

- **Analyze** your code to determine most likely places needing parallelization or optimization.
- **Parallelize** your code by starting with the most time consuming parts and check for correctness.
- **Optimize** your code to improve observed speed-up from parallelization.

Analyze
Parallelize
Optimize
Profile Your Code
Obtain detailed information about how the code ran.
This can include information such as:
• Total runtime
• Runtime of individual routines
• Hardware counters
Identify the portions of code that took the longest to run. We want to focus on these "hotspots" when parallelizing.
Lab Code: Laplace Heat Transfer
OpenACC
NVIDIA

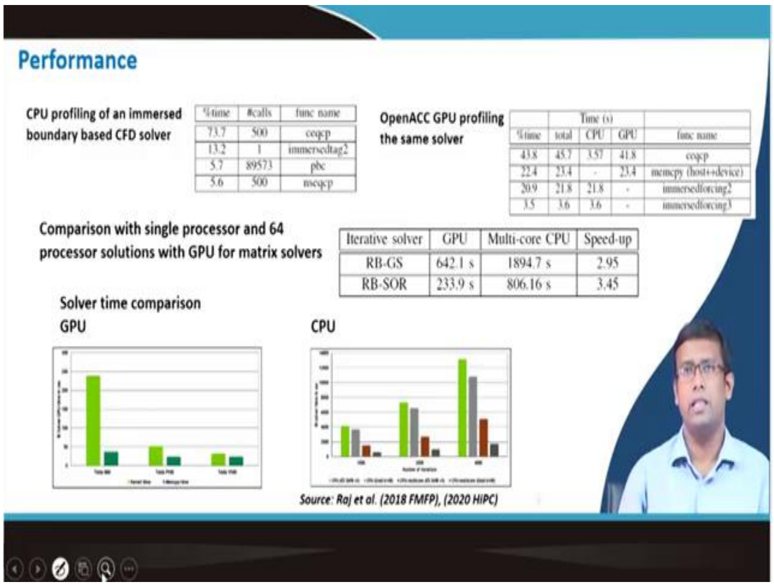
Identify the compute heavy parts and parallelize them, look into data transfer
Optimize the parallelization loop and data management

So, OpenACC development cycle is typically specified by OpenACC best practice guide and the engineers working on OpenACC. So, this is a formal way of approaching a large program and see how we can parallelize it.

First you have to analyse the code and determine the places where parallelization is required. How to do that? If you take the code and profile it, find out the total runtime, runtime of universal routines with Nvidia nsight; you can profile it and there are many profiling software available and hardware counters. Identify the portions of the code which took longer time and find out which is the hotspot which has to be parallelized. So, analyse the code, profile it and identify which has to be parallelized.

Then, parallelize that part, in a code there is a particular total run time of 39 seconds something, takes 30 seconds something, takes 90 second like that and consider that 30 second part, parallelize it using the OpenACC part. While parallelizing it, look into the data transfer part, look into the non-data dependent locations and use the right parallelization directives there. Once parallelized, optimize that using the loop and data management and then you get the parallel program and this is the most formal way of approaching a scientific computing job that you have your own legacy code, profile it, find out the hotspot, parallelize it and then try to optimize the performance.

(Refer Slide Time: 38:43)



Performance

CPU profiling of an immersed boundary based CFD solver

%time	#calls	func name
73.7	500	ceqcp
13.2	1	immersedtag2
5.7	89573	pbc
5.6	500	mseqcp

OpenACC GPU profiling the same solver

	Time (s)			
%time	total	CPU	GPU	func name
43.8	45.7	3.57	41.8	ceqcp
22.4	23.4	-	23.4	memcpy (host+device)
20.9	21.8	21.8	-	immersedforcing2
3.5	3.6	3.6	-	immersedforcing3

Comparison with single processor and 64 processor solutions with GPU for matrix solvers

Iterative solver	GPU	Multi-core CPU	Speed-up
RB-GS	642.1 s	1894.7 s	2.95
RB-SOR	233.9 s	806.16 s	3.45

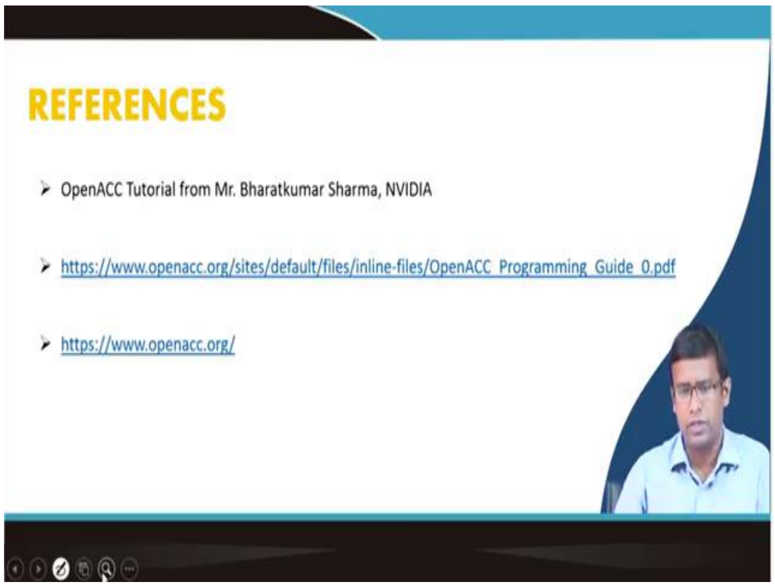
Solver time comparison

Source: Raj et al. (2018 FMFP), (2020 HiPC)

We can see an example that this is the immersed boundary based CFD solver and this was published in one of my students' papers. So, he found that 73 percent time is taken by one particular sub routine and that has been parallelized and then, it took 43 percent time and most of the time is spent on GPU, it went to GPU and as GPUs are faster the overall time requirement is reduced. This basically required iterative solvers. So, from multi core CPU, 64-Processor Xeon CPUs, 2 GPU's, there are 3 to 3.5 times speed up. So, OpenACC tune GPUs can give you much faster performance than 64-Core Processor CPU's.

These comparisons between CPU and GPU time for this immersed boundary solver. In CPU, in a single CPU, it was taking something like 1300 second; it came down to little less than 100 seconds in a Multicore 64-Processor CPU whereas, in GPU, we can see that this time required is around 10 or 20 seconds. So, the performance can be very well improved if you identify the hot spots and then parallelize and optimize a legacy science engineering code using OpenACC.

(Refer Slide Time: 40:19)

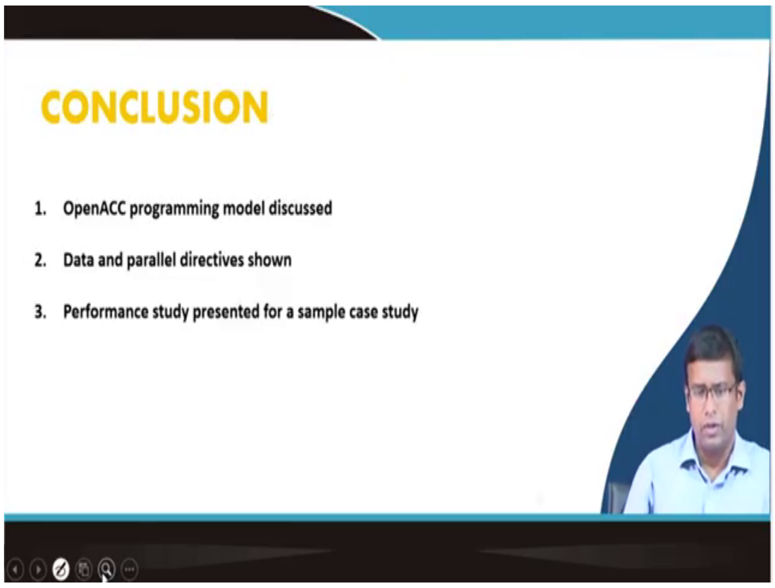


REFERENCES

- OpenACC Tutorial from Mr. Bharatkumar Sharma, NVIDIA
- https://www.openacc.org/sites/default/files/inline-files/OpenACC_Programming_Guide_0.pdf
- https://www.openacc.org/

Well, so, we looked into OpenACC tutorial from Mr Sharma of NVIDIA, OpenACC programming guide best practice and programming guide and openacc.org has sufficient material available on OpenACC which you can start reading and utilize for getting your application ported in GPUs as well as on multicore CPUs.

(Refer Slide Time: 40:39)



CONCLUSION

1. OpenACC programming model discussed
2. Data and parallel directives shown
3. Performance study presented for a sample case study

We looked into OpenACC programming model, data and parallel directives and performance study for a sample case.