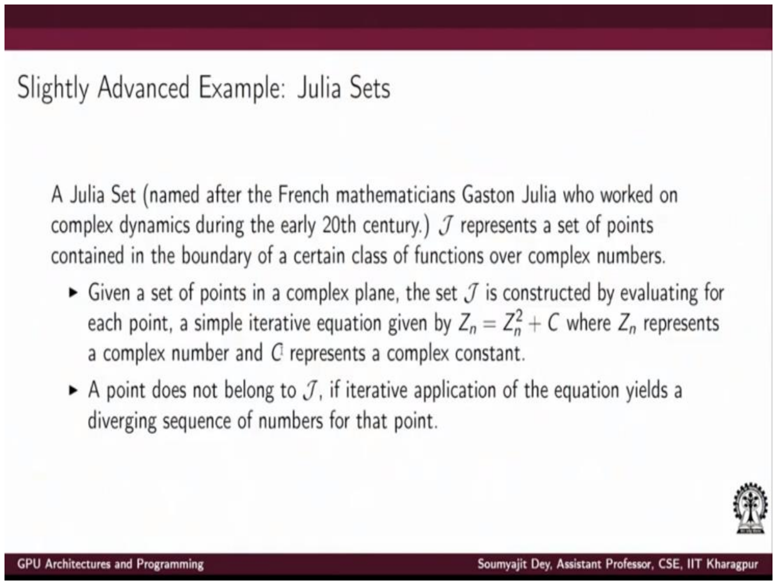
GPU Architectures and Programming

Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Module No # 03
Lecture No # 12
Intro to CUDA programming (Contd.)

(Refer Slide Time 00:31)



Slightly Advanced Example: Julia Sets

A Julia Set (named after the French mathematicians Gaston Julia who worked on complex dynamics during the early 20th century.) $\mathcal{J}$ represents a set of points contained in the boundary of a certain class of functions over complex numbers.

- Given a set of points in a complex plane, the set $\mathcal{J}$ is constructed by evaluating for each point, a simple iterative equation given by $Z_n = Z_n^2 + C$ where Z_n represents a complex number and C represents a complex constant.
- A point does not belong to $\mathcal{J}$, if iterative application of the equation yields a diverging sequence of numbers for that point.

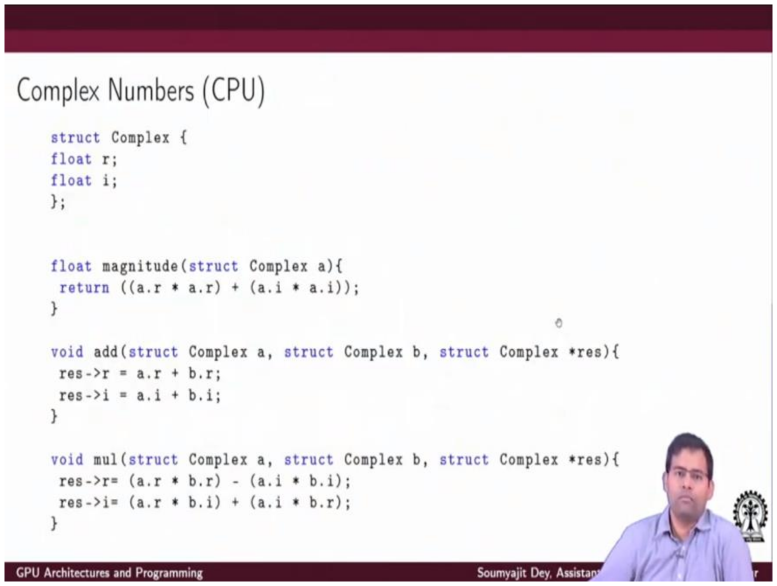
GPU Architectures and Programming — Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Hi So let us go for some more advanced examples on parallel program writing in CUDA and here we will pick on some mathematical thing entity called Julia sets. So what is a Julia set it is named after French mathematician Gaston Julia who worked on complex dynamics during the early twentieth century. Now J represents a set of points contained in the boundary of a certain class of functions defined over complex numbers.

And we define this kind of shapes as follows that you are given a set of points in complex plane a set J can be constructed by evaluation for each point a simple iterative equation which is given by this. So Jn is iteratively computed using the previous value of Jn being squared plus a constant complex constant C being added. And if a point does not belong Julia set then it would a behavior that is in iterative application of the equation will yield a diverging sequence of numbers for that point.

So essentially what would happen is for certain specific points on the complex plain if I continuously iterate over that equation it will I will get a sequence that is diverging then I will say that the point does not belong to J and otherwise if I get a sequence which is not diverging I will say that yes this point is part of the Julia set. Now of course for you can understand that this computation is very much a function of what is my choice of this complex constant for different choices of complex constants I will be getting different Julia sets.

(Refer Slide Time 02:17)



Complex Numbers (CPU)

```
struct Complex {
float r;
float i;
};


float magnitude(struct Complex a){
 return ((a.r * a.r) + (a.i * a.i));
}

void add(struct Complex a, struct Complex b, struct Complex *res){
 res->r = a.r + b.r;
 res->i = a.i + b.i;
}

void mul(struct Complex a, struct Complex b, struct Complex *res){
 res->r= (a.r * b.r) - (a.i * b.i);
 res->i= (a.r * b.i) + (a.i * b.r);
}
```

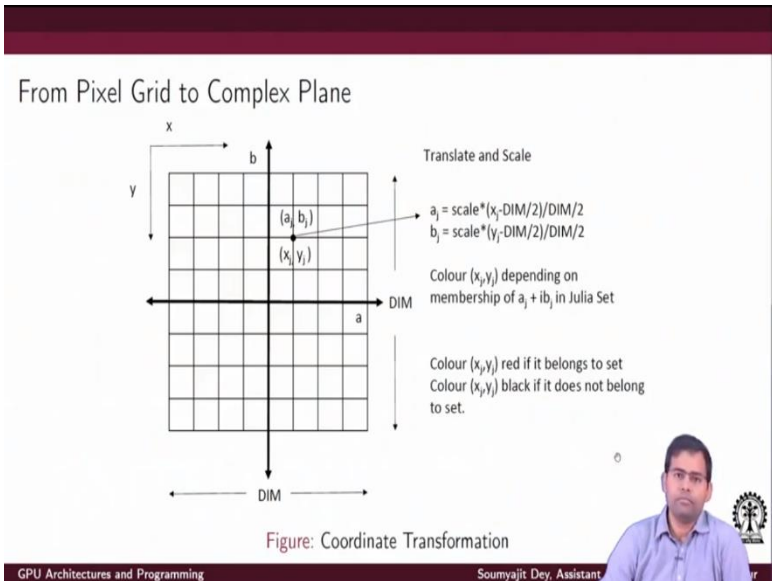
GPU Architectures and Programming   Soumyajit Dey, Assistant

So I mean just again I mean how do we represent complex numbers in CPU’s. So I mean you can have a structure complex with variables r and i I mean they are there I mean our float variables and they represent a real and i mean real imaginary part and I can have several different operations for example computing the magnitude and doing an addition and also doing some basic multiplication.

So if I look at doing an addition I am expecting 2 complex numbers are provided and I am going to write back the result of the addition by using a pointer to a complex types structure arrays. So it is as simple as this and again I can do a multiplication in the complex plane using standard mathematical formulas like this. So as you can see that I have been provided with complex type input a complex type input b and I am do I am computing these results of the multiplication in terms of the real part and the imaginary part separately.

So this first equation will evaluate the real part and the second equation will evaluate the imaginary part and once it is done I have the result return back in this point using this pointed arrays right.

(Refer Slide Time 03:32)



From Pixel Grid to Complex Plane

Translate and Scale

a_i = scale*(x_i-DIM/2)/DIM/2
b_i = scale*(y_i-DIM/2)/DIM/2

Colour (x_i,y_i) depending on membership of $a_i + ib_i$ in Julia Set

Colour (x_i,y_i) red if it belongs to set
Colour (x_i,y_i) black if it does not belong to set.

Figure: Coordinate Transformation

GPU Architectures and Programming   Soumyajit Dey, Assistant

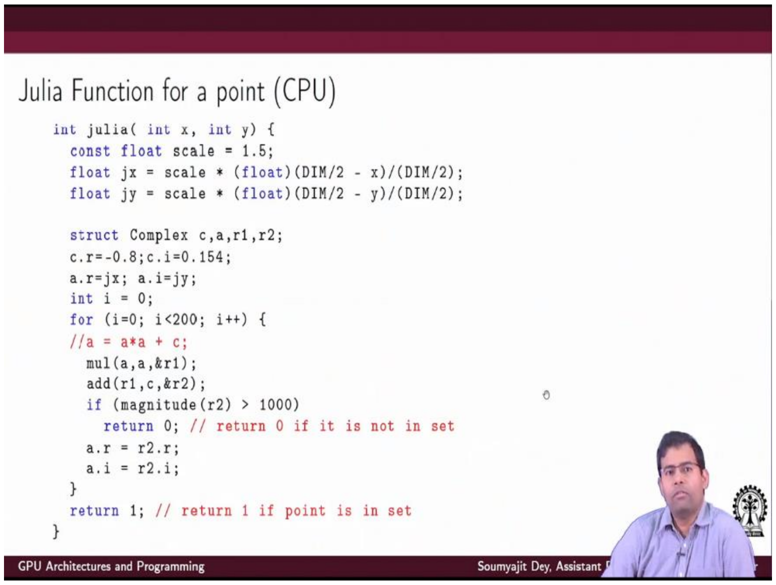
So the first thing our program will do is it will try to make a mapping from the pixel width to the complex plane. Of course we are trying what we are trying to do our objective here is we will also like to write the GPU code to compute a Julia set and represent those values as a bitmap on the screen that will kind of create a very nice looking pattern. So for doing that the screen is represented by a pixel grid and I have to map each point from the screen is coordinate from the x y location to a corresponding a b location right.

So this is how I have the x y's coordinate starting from this corner and here I am going to compute the corresponding a b. So first thing we will do is we will do a for computing this ab points for the given xy points I do a scaling. Before that scaling I subtract from this xa dimension by 2 but and that gives be a value here with the respect to origin. And also if I do a subtraction over the y that also gives me a value in terms of b.

And I do a further division by dimension by 2 to do a scaling. So when I say that I have a corresponding value in terms of ab and I will be doing some computation on this and then I will determine whether this value is whether this coordinate is a member of Julia set or not. The fact that it is member will be represented by a different color. So the color coding we are going to follow here is that color xy color xj yj.

If it is red that would mean that it belongs to the set and if it is black that means it does not belong to a set. So what all I would expect at the end that there is a blackened screen and on which I have a red colored pattern which is representing the Julia set.

(Refer Slide Time 05:36)



Julia Function for a point (CPU)

```
int julia( int x, int y) {
  const float scale = 1.5;
  float jx = scale * (float)(DIM/2 - x)/(DIM/2);
  float jy = scale * (float)(DIM/2 - y)/(DIM/2);

  struct Complex c,a,r1,r2;
  c.r=-0.8;c.i=0.154;
  a.r=jx; a.i=jy;
  int i = 0;
  for (i=0; i<200; i++) {
  //a = a*a + c;
    mul(a,a,&r1);
    add(r1,c,&r2);
    if (magnitude(r2) > 1000)
      return 0; // return 0 if it is not in set
    a.r = r2.r;
    a.i = r2.i;
  }
  return 1; // return 1 if point is in set
}
```

Now how do the Julia function look like for a specific point I mean when I am doing a CPU side computation and I am trying to evaluate whether a specific point on the screen is a member of the Julia set or not. So you assume that you are supplied with the xy coordinates that is the coordinates with respect to this corner here. The first thing you are going to do is your going to do this transformation in terms of scaling and you are trying to find out what is the corresponding ab values of the coordinate.

Of course DIM represents that dimension that means the total size here of the screen size that I am considering and I am also considering that it is a square sized area or DIM cross DIM So using this transformation that we have discussed here in terms of aj and bj we now create this variable in this as jx and jy right. So when I have these jx and jy created I used them to now denote the new coordinates here in terms instead of x and y.

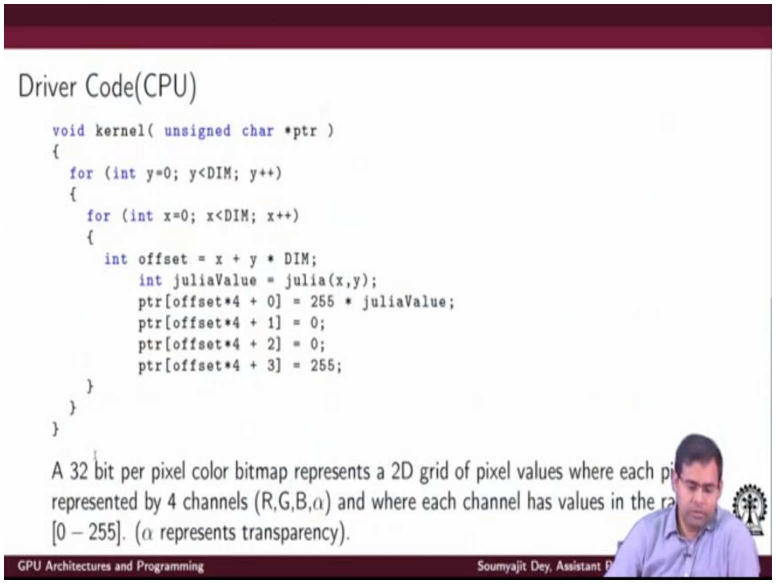
So this essentially will give me a result of applying this formula here right. So essentially I will be doing all my Julia set computation on these jx and jy and it is very standard. So just to look at the original mathematical formula all we are supposed to do is we are supposed to iterate over this equation multiple times see what are the result? Whether the result is converging or not and based on that we take a decision whether it is a member of Julia sets or not.

And of course, there is this complex param constant parameter which we assume is provided to us. So here for our purpose this complex parameter is assumed to have a real part this and imaginary part as given here. And so we consider that the input value has a real part that is jx and the input value has an imaginary part that is jy. And then we are into inside this loop which is iterating 200 times and it is basically executing this equation a times a + c getting recomputed and recomputed.

So for doing that since a is of the type complex and we carry out this function multiply. Let us you multiply a and a write back the result in r1 and then you take the content of r1 and you add it to c and then you return it r2 and you check whether the magnitude is greater than something some space given value or not. If it is really so then you return back, you say that the item is not in the set otherwise you say that yes indeed the item is member of the set.

The way you say this it is a member of the set is you write back this value of r2 here and you also go back to the computation here. So at the end what will happen is if inside this 200 number of iterations it is never the case that the magnitude is crossing this thousand then I have the final value here in a and it is and if the loop executes up to this 200 iteration never returning back here then you finally make a return from this point and you have the final value that is computed stored in the complex data type a.

(Refer Slide Time 09:37)



Driver Code(CPU)

```
void kernel( unsigned char *ptr )
{
  for (int y=0; y<DIM; y++)
  {
    for (int x=0; x<DIM; x++)
    {
      int offset = x + y * DIM;
          int juliaValue = julia(x,y);
          ptr[offset*4 + 0] = 255 * juliaValue;
          ptr[offset*4 + 1] = 0;
          ptr[offset*4 + 2] = 0;
          ptr[offset*4 + 3] = 255;
    }
  }
}
```

A 32 bit per pixel color bitmap represents a 2D grid of pixel values where each pi represented by 4 channels (R,G,B,α) and where each channel has values in the ra [0 – 255]. (α represents transparency).

GPU Architectures and Programming Soumyajit Dey, Assistant P

Now so this is a code that is trying to design a membership decision for a provided x, y value right that would be summary here. You are given the x, y value from that you are trying to create a corresponding complex number a you evaluate this equation. You see whether it is diverging or not. If you and your definition of divergence here is that ok I will execute this loop 200 times.

If even after that this magnitude does not cross thousand then I will simply return back assuming that yes this is a member of the Julia set corresponding to of course this choice of complex parameters see here. Now of course there is a membership decision function and it has to be called so for that I have a driver code which is here in terms of the kernel. So essentially I am thinking that ok for the entire input bitmap there is representing the entire screen area.

I am going to do a pixel by pixel membership testing for whether that because any specific pixel is a member of the Julia set or not that is my target. So for that I set up this nice cascade of loops each loop is a derating from 0 to dimension because of course I am going to cover the entire screen here. So coming back here I have this cascade of loops from 0 to dimension and inside this cascade what i do is first I make a computation of this offset that means what is the location I am interested in offset is of course x + y times the dimension.

Why that is required we will come into that but before that we compute the Julia value that means I passed these parameters x and y to this membership testing function Julia that we have discussed earlier. And in return backs and tell me that yes this x y value is indeed a member of this I mean this x y parameter value is indeed a member of Julia set or not. So that is available here so if it is really a member then Julia value would get a value 1 right.

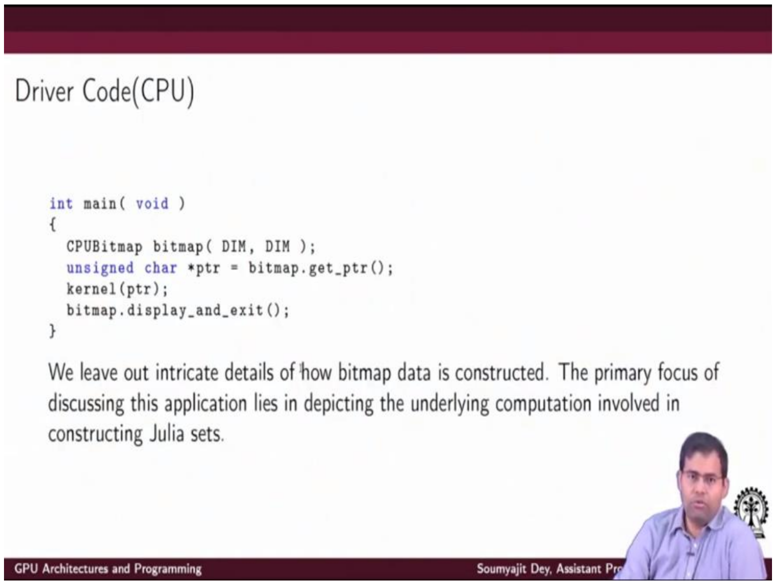
Now if it gets a value 1 then we have this corresponding location. So that means the offset times 4 + 0 is being provided with 255 is being assigned the value 255 right. Now why do I have this offset getting multiplied by 4. Essentially we are thinking that in standard bitmaps a 32 bit pixel color bitmap is being represented by a 2D grid of pixel values and each pixel has got its color being represented in terms of 4 channels. So each channel is for red, green, blue and alpha. Where alpha represents the transparency and each channels value is to be inside this range of 0 to 255.

So I mean with proper with suitable combination of this values I can generate the corresponding color. So since I have to give for any pixel value I have give the corresponding value of these channels this is how we are doing it. First we are thinking that this ptr array is representing the bitmap. We do the offset computation to traverse to the corresponding location in the ptr. So offset is nothing but x + y times the dimension.

So essentially I can think that these offset represent one cell here. Now the value in this cell I mean it is of course each cell has got 4 channels so this are the multiply by 4. And after that you have to keep the corresponding value for each cell right. So that is why you have 4 values + 0 + 1 + 2 and +3. As we have decided earlier that if it is a member then we are going to give a color it with red and otherwise it is all black.

If I have to color it with red but we do is Julia value is 1. In the first channel represents red. So I multiply 1 with 255 and rewrite in the red channel. For the other G and B channels we anyway keep them at 0 and we also have the transparency value to full. So this is the kernel code for CPU which is going to take each of the ptr’s that means each pixel position and transform and tell me back whether this is going to be a really inside a member of Julia set or not and if it is a member of the Julia set then it is going to paint that pixel position by itself.

(Refer Slide Time 14:37)



Driver Code(CPU)

```
int main( void )
{
  CPUBitmap bitmap( DIM, DIM );
  unsigned char *ptr = bitmap.get_ptr();
  kernel(ptr);
  bitmap.display_and_exit();
}
```

We leave out intricate details of how bitmap data is constructed. The primary focus of discussing this application lies in depicting the underlying computation involved in constructing Julia sets.

GPU Architectures and Programming
Soumyajit Dey, Assistant Pro

So this is the kernel code in terms of the CPU. Now of course that code needs to be called for that we will have a main function in c where we first do the bitmap setup by using these functions I mean. So these are the basic bitmap set up which will return the CPU bitmap and we get the pointer that I mean this p pointer which is going to point to the screen locations.

Now of course in this context we leave out the details of how bitmap data is constructed. Since the primary focus here is that we are trying to see that how to do so some computation on the bitmap. Of course I mean getting bitmaps and with using suitable headers and all that is very simple. There are standard programs and we of course provide you the references right here for that.

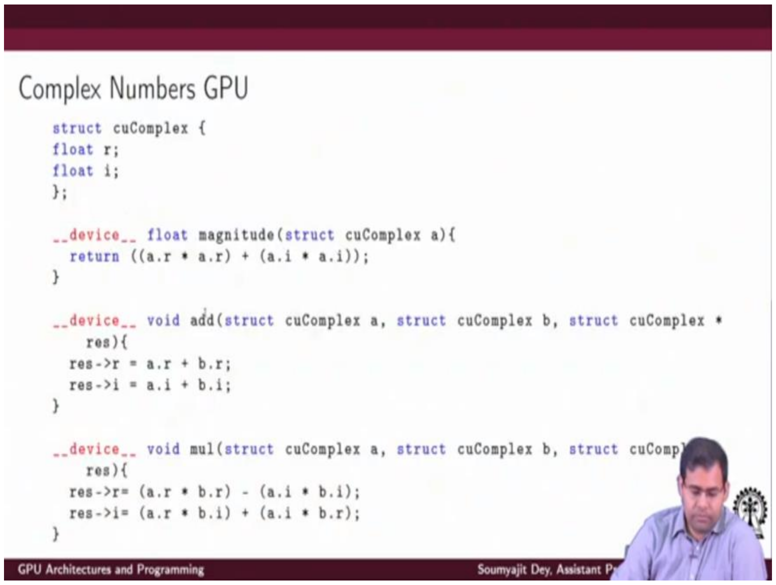
So here we get the pointer to this differ bitmap for the screen we call the kernel with these pointer locations for each of them and we finally do a display and exit here. Now how about doing this computation for the GPU side of the program. Now again I would like to say that this kernel code that has been demonstrated here is only for one location here right. But how does it work because the call has been made as you can see for the base position right.

This CPU side port makes the kernel called for the base position and then from the that mean I am passing kind of this location in ptr right and then I have these two outer loops which are going to traverse through the entire grid of locations and do the color compute a this coloring part and before that it is going to do the membership testing by using this Julia function.

So this as you can see that just like for matrix multiplication I had a cascade of loops first the cascade of loops the outer cascade of 2 loops were there to choose the row for the matrix 1 and the column for the matrix 2 and then the inner loop was doing the computation. So in a similar structure I have this outer 2 cascade of loops which are traversing over this bitmaps. And that inner code is actually doing the Julia set computation and then deciding whether this is the member of Julia set and then correspondingly coloring up with a suitable value whether it is red or black.

So the overall idea is very it mean very simple that since we are assuming that the CPU is single threaded. So there is a one thread of computation that is going to over this multiple outside cascade loops and in each iteration of the inner loops it is going to do the Julia set computation and coloring business.

(Refer slide Time 17:30)



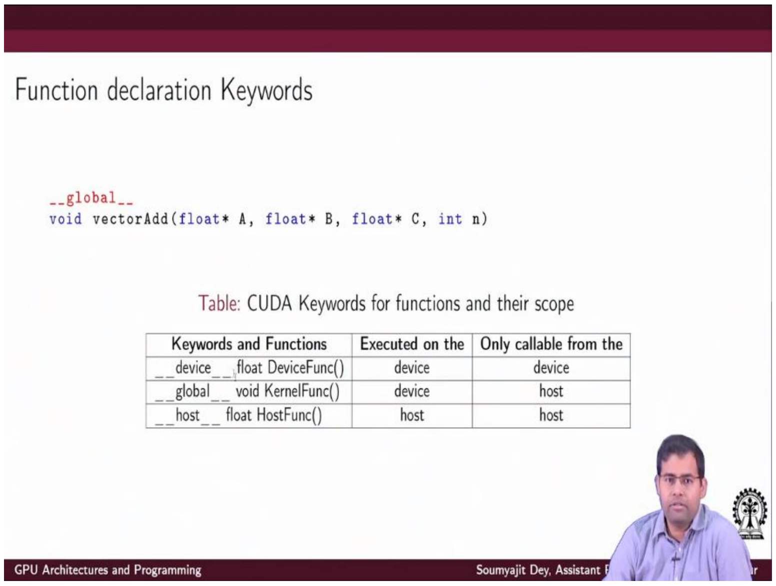
Complex Numbers GPU

```
struct cuComplex {
float r;
float i;
};

__device__ float magnitude(struct cuComplex a){
  return ((a.r * a.r) + (a.i * a.i));
}

__device__ void add(struct cuComplex a, struct cuComplex b, struct cuComplex *
    res){
  res->r = a.r + b.r;
  res->i = a.i + b.i;
}

__device__ void mul(struct cuComplex a, struct cuComplex b, struct cuComp
    res){
  res->r= (a.r * b.r) - (a.i * b.i);
  res->i= (a.r * b.i) + (a.i * b.r);
}
```

GPU Architectures and Programming   Soumyajit Dey, Assistant P

Now what about the corresponding code in the GPU. So again we will have assumption that we have some device functions here. Now if you remember for in terms of GPU kernels we made a classification there is a device function as well as global function. So the global function is a kernel that can be called from a host side code and a device function is a GPU kernel which can be called from some other function that is already executing in the GPU right. So just for your remembrance let us just traverse back to this functional declarations.

(Refer Slide Time 18:10)



Function declaration Keywords

```
__global__
void vectorAdd(float* A, float* B, float* C, int n)
```

Table: CUDA Keywords for functions and their scope

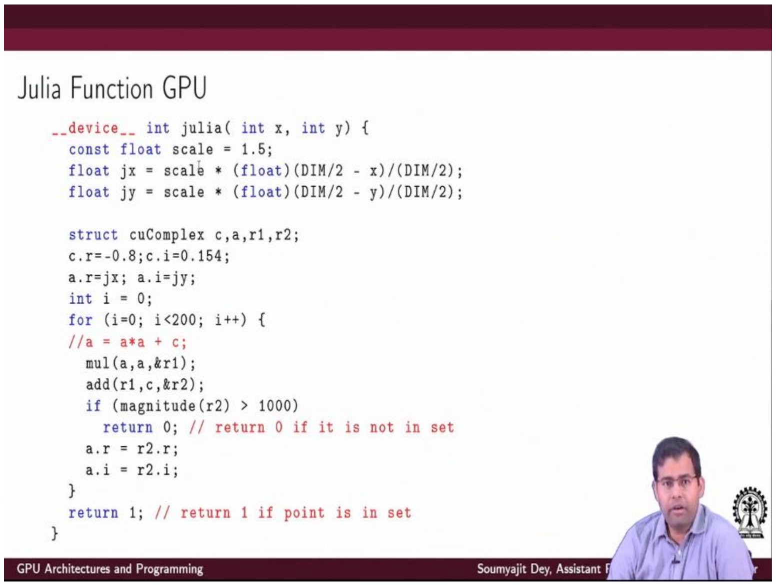
Keywords and Functions	Executed on the	Only callable from the
__device__ float DeviceFunc()	device	device
__global__ void KernelFunc()	device	host
__host__ float HostFunc()	host	host

GPU Architectures and Programming Soumyajit Dey, Assistant

So here we had device type function which execute on device and they are callable only from device functions. Whereas we have global functions which executes on device but they are callable from host side functions. So here as you can see we define some device site functions so one for magnitude one for addition one for structure. Now these are just like our earlier programs which were basically the c programs which represented doing this computation in the complex plane in terms of magnitude addition and multiplication.

So they are just very similar here and I mean because they are like standard c functions there is no hint of parallelism or anything here.

(Refer Slide Time 19:13)



Julia Function GPU

```
__device__ int julia( int x, int y) {
  const float scale = 1.5;
  float jx = scale * (float)(DIM/2 - x)/(DIM/2);
  float jy = scale * (float)(DIM/2 - y)/(DIM/2);

  struct cuComplex c,a,r1,r2;
  c.r=-0.8;c.i=0.154;
  a.r=jx; a.i=jy;
  int i = 0;
  for (i=0; i<200; i++) {
  //a = a*a + c;
    mul(a,a,&r1);
    add(r1,c,&r2);
    if (magnitude(r2) > 1000)
      return 0; // return 0 if it is not in set
    a.r = r2.r;
    a.i = r2.i;
  }
  return 1; // return 1 if point is in set
}
```

GPU Architectures and Programming Soumyajit Dey, Assistant

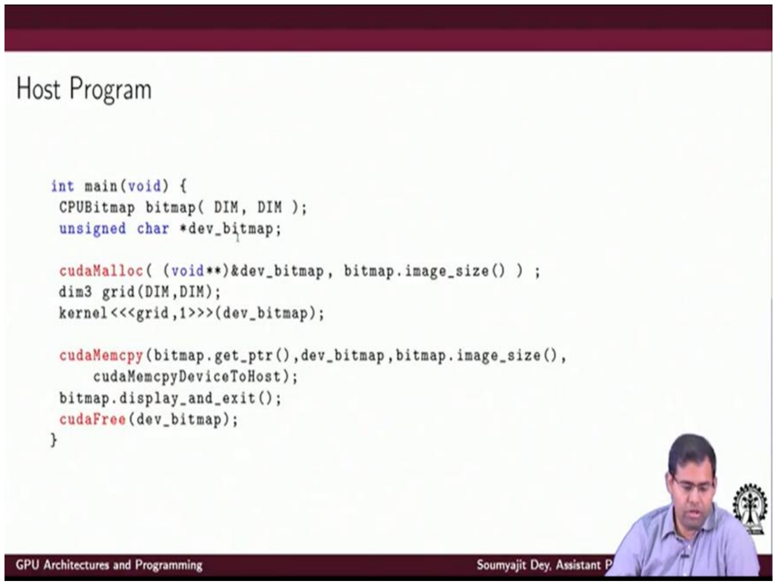
We also have another device function which is named Julia. Now this is the function that is going to do the computation that is the membership of the Julia set and that is also quite similar as you can see that you do the computation of the positions on the with respect to the shift of origin and scale first with respect to jx and jy and then you use those value to initialize a complex data type a you make your simple choice of c I mean in this complex constant values with respect to both the imaginary and real parts here.

And you define and you define these different types of I mean complex types I mean were just calling cu complex where the CUDA equivalent part right. And so you have this complex constant c which has been defined and you have the input a you have this original loop here I mean this just like the normal c code. Essentially you are doing the multiplication then the addition and then you are doing the magnitude testing whether it is converging or not.

And if you are able to execute this code 200 times without ever returning out then you say that ok this is going to be the member of this Julia set. So this is just like the normal CPU side code. But then then the question is why do I start saying that yes it is a GPU, GPU site code. The reason is this is a device side function which will execute on the GPU and it will be called from some other GPU kernel.

So now let us look at the other GPU kernel. So this is it so this is a global function that means this is the GPU kernel which is going to be launched from some host side code. What does the GPU kernel do? The GPU kernel first identifies the x and y values. So that is essentially it can find out based on the block id and the block Idx dot x and block Idx dot y right. Now to get an idea that how is this block defined.

(Refer Slide Time 21:20)



Host Program

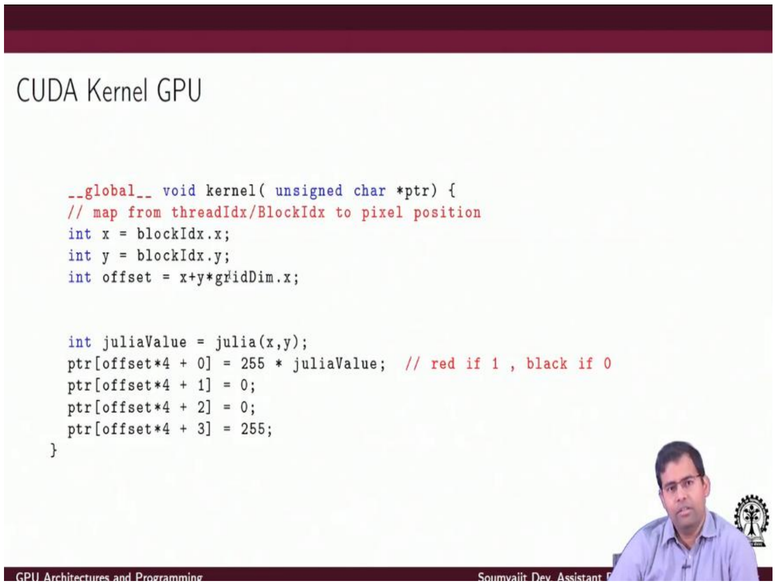
```
int main(void) {
 CPUBitmap bitmap( DIM, DIM );
 unsigned char *dev_bitmap;

 cudaMalloc( (void**)&dev_bitmap, bitmap.image_size() ) ;
 dim3 grid(DIM,DIM);
 kernel<<<grid,1>>>(dev_bitmap);

 cudaMemcpy(bitmap.get_ptr(),dev_bitmap,bitmap.image_size(),
     cudaMemcpyDeviceToHost);
 bitmap.display_and_exit();
 cudaFree(dev_bitmap);
}
```

Let us go further to the host program so this is the host program. As you can see in the main I have the CPU bitmap a I mean that function like earlier which is providing me the bitmap definition and then I do a CUDA Malloc on the GPU side memory. On the GPU side memory I am trying to transfer the bitmap and then I design a grid here of type dimension 3.
And then in this grid definition as you can see when the kernel is called.

The kernel is called with this block and number of threads per block as one. So what is a block here a block is just a containing this dimension, dimension number of blocks right. So essentially that would mean for each position I just have a block Id and block Idx and block Idy and every block has got just one thread. So that is why with comma I have a one here. These things will become more clear as we progress with a course and then we have a CUDA Mem copy comment.

Once the kernel has executed we can assume that all the set computational has been done and accordingly the bitmap has been painted and then the bitmap display function is called and after that we have a CUDA Free for the device bitmap.

(Refer Slide Time 22:50)



CUDA Kernel GPU

```
__global__ void kernel( unsigned char *ptr) {
// map from threadIdx/BlockIdx to pixel position
int x = blockIdx.x;
int y = blockIdx.y;
int offset = x+y*gridDim.x;


int juliaValue = julia(x,y);
ptr[offset*4 + 0] = 255 * juliaValue;  // red if 1 , black if 0
ptr[offset*4 + 1] = 0;
ptr[offset*4 + 2] = 0;
ptr[offset*4 + 3] = 255;
}
```

GPU Architectures and Programming   Soumyajit Dey, Assistant

So coming back here we have this CUDA kernel whom I have provided with these many number of threads. 1 thread per block and then number of blocks we have defined is dimensions square right. So using the block ID I am getting a direct map to the pixel I hope that is clear. So we go back to this nice picture here. So essentially now we are saying that we are defining we are launching these many threads as is the number of pixel and I am also saying that we are launching these many blocks that means essentially we are saying that we are launching these block with only one thread per block.

So once that launch is being done every x, y value of course will get the corresponding block Idx dot x and block Idx dot y and using them I first do the computation of offset. That is nothing but I mean I have the x axis + the number the y value that is the number of rows that would be getting multiplied by the dimension of the grid. Now the dimension of the grid has been set as you can see the dimension of the grid is set as the original DIM dimension value that we have discussed earlier in that graphic slide.

So now for each thread in this kernel I have got the idea that this thread has to decide for the Julia set membership position of 1 x, y coordinate right. So for that this Julia function will now be called and this Julia function is just like the original Julia function for the c code is just a GPU equivalent 1 but it is a device function which means this is already a kernel. This kernel has been launched with dimension times dimension that means a number of threads where each thread is suppose to look up to the membership of 1 element right 1 pixel.

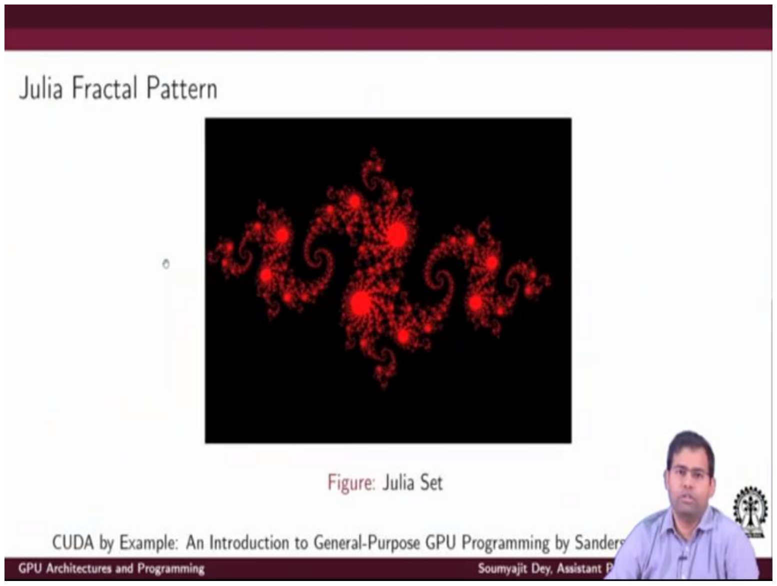
So since this is already a kernel and it is going to launch a c code which will do the membership testing that is why this Julia function is now being called a function which is a device side function. So it has got this device to (()) (25:23) with it. So once being passed with the parameter x and y it should return back the Julia value and just like earlier we will be using this Julia values to assign the corresponding suitable (()) (25:37) channels here to this ptr memory locations.

And with that when this kernel execution finishes we have got the bitmap colored properly with respect to the Julia set computation. And finally the bitmap display image is able to display properly. So again where does it get different in terms of the CUDA program and originals GPU side program. Like we have discussed earlier the original CPU side program had this cascade of loops.

But now when we start talking about the Julia set computation we just do not have this cascade of loops. Instead of that we have this kernel launch and the kernel is launching that many dimension times dimension number of threads. And those number of threads essentially are executing parallel in the GPU and while executing parallelly in the GPU each of the thread is responsible for doing membership decision for each position here.

And for each position it is doing the it is calling the device function Julia. And if the call returns to one then it is doing a proper coloring of the bitmap here. Of course it is a one then it is writing the red channel otherwise it is writing 0 here and so everything is black with all that when this code is executed the host program finishes we will get a nice pattern here.

(Refer Slide Time 27:04)



Julia Fractal Pattern

Figure: Julia Set

CUDA by Example: An Introduction to General-Purpose GPU Programming by Sanders

GPU Architectures and Programming

Soumyajit Dey, Assistant

As we have said earlier that what pattern you get is also a function of what is the coefficient that you choose that this was our choice of coefficient this as I mean we that we made for the complex constant c for that coefficient choice you would get this kind of pattern. Now this part of the code I will like to mention that this has been adapted from an introduction to general purpose GPU programming by Sanders et al.

And we used that reference to write this program execute them in our environment and reproduce this picture. So this we thought would be a nice example after matrix multiplication in both the examples we are trying to show you how iterative job that the CPU does by having multiple hierarchies of loops can be replace by suitable kernel launches. Where you launch that many number of threads and each thread does some specific job on some specific data element.

None of these codes are optimized with respect to the GPU’s architectural features. But all it does is it exploits the parallelism on based on the job at hand. First we have the matrix multiplication program as we discussed there is lot of parallelism because I can compute each of the matrix into these in parallel. Similarly in the Julia thread Julia fractal case I am trying to take a membership decision about each of the pixels in the display and that also can be done in parallel.

And we thought this would be a nice example because this is also an example of a graphics program through which you are trying to see that how GPU acceleration can provide you good real time graphics performance although we are going to deal more about general purpose graphics processing units but graphics program were the first work loads that were identified for graphic processing units. So with that we like to end this lecture and thank you.