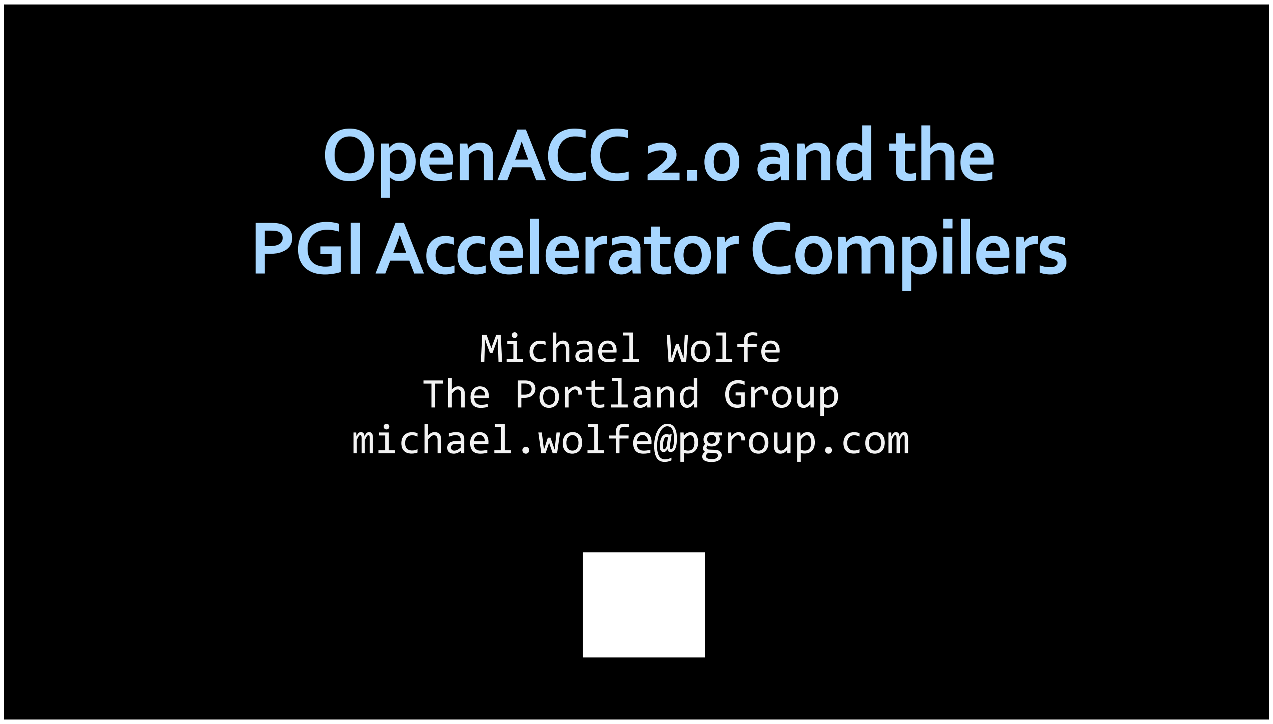
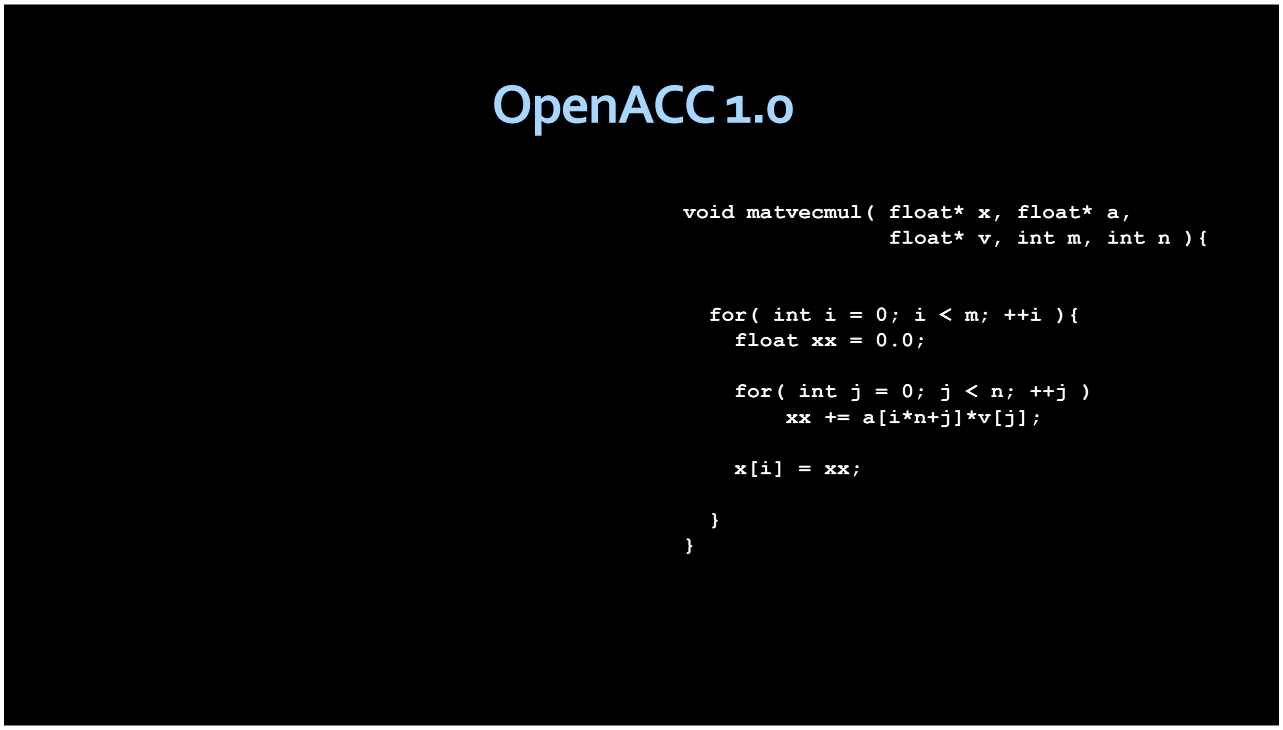


OpenACC 2.0 and the PGI Accelerator Compilers

Michael Wolfe
The Portland Group
michael.wolfe@pgroup.com

This presentation discusses the additions made to the OpenACC API in Version 2.0. I will also present some PGI extensions to OpenACC; PGI's intent is to explore new features before proposing them as part of the OpenACC specification. Most of my examples will be in C, though I will mention some features specific to Fortran.

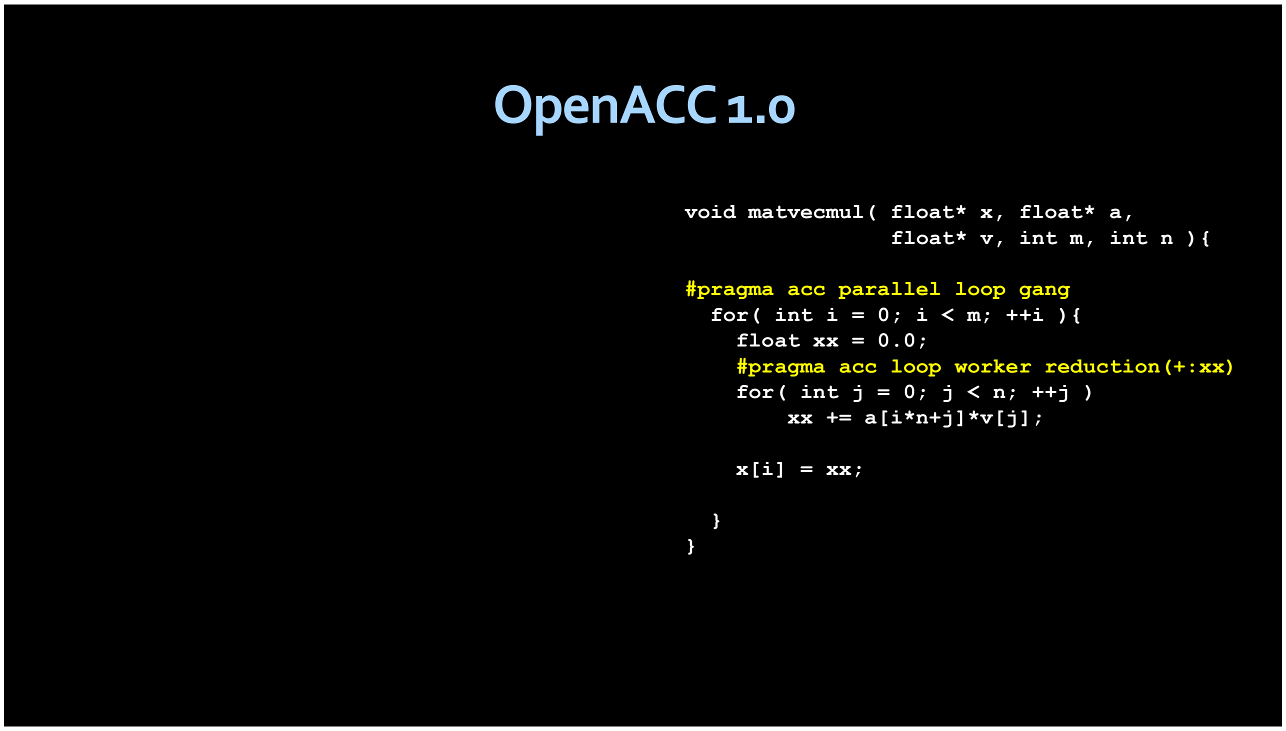
I will start with a very short review of OpenACC; if you are not familiar with OpenACC, you should find an official introduction, as this is a very brief intro. Let me start by stating that OpenACC does not make GPU programming easy. You will hear some vendors or presenters saying that OpenACC or some other language, library or framework makes GPU programming or makes parallel programming easy. Do not believe any of these statements. GPU programming and parallel programming is not easy. It cannot be made easy. However, GPU programming need not be difficult, and certainly can be made straightforward, once you know how to program and know enough about the GPU architecture to optimize your algorithms and data structures to make effective use of the GPU for computing. OpenACC is designed to fill that role.



OpenACC 1.0

```
void matvecmul( float* x, float* a,
                float* v, int m, int n ){


  for( int i = 0; i < m; ++i ){
    float xx = 0.0;

    for( int j = 0; j < n; ++j )
        xx += a[i*n+j]*v[j];

    x[i] = xx;

  }
}
```

This example is a matrix-vector multiplication in a routine. I can port this to a GPU, or an accelerator in general, using OpenACC directives as follows:

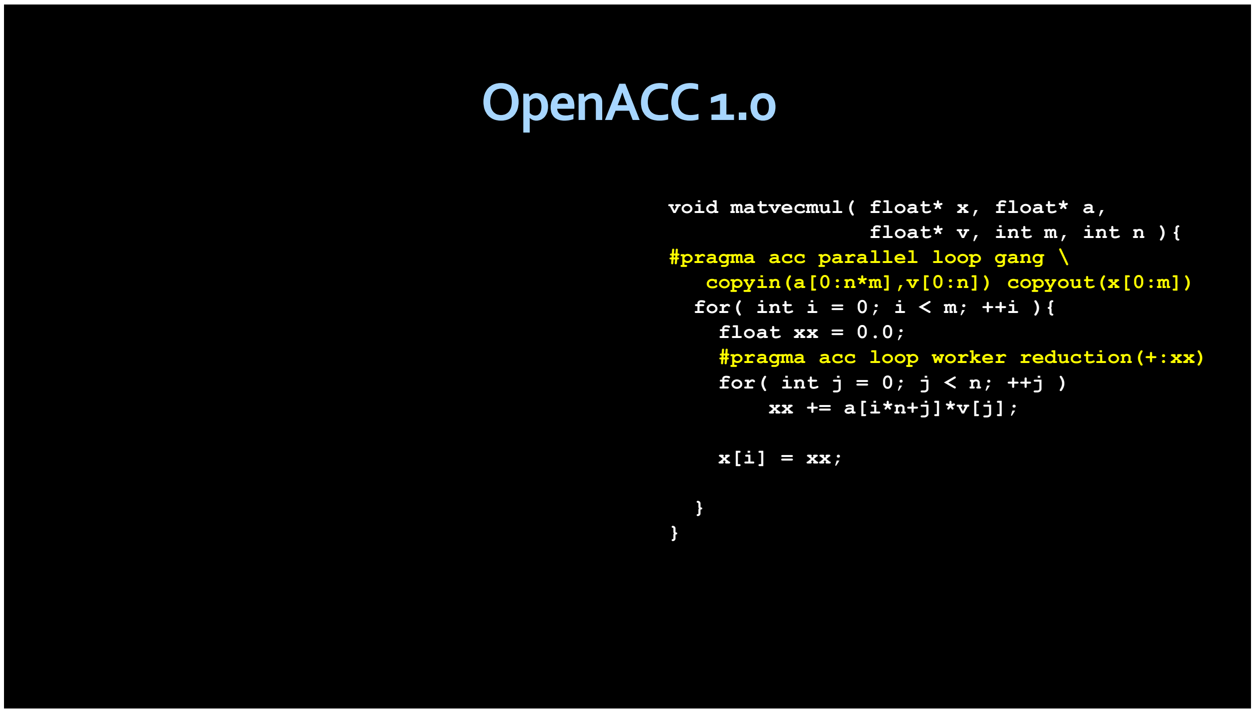


OpenACC 1.0

```
void matvecmul( float* x, float* a,
                float* v, int m, int n ){

#pragma acc parallel loop gang
  for( int i = 0; i < m; ++i ){
    float xx = 0.0;
    #pragma acc loop worker reduction(+:xx)
    for( int j = 0; j < n; ++j )
        xx += a[i*n+j]*v[j];

    x[i] = xx;

  }
}
```

The OpenACC directives are intentionally similar syntactically to OpenMP directives, if you are familiar with OpenMP. The first directive tells the compiler to execute the iterations of the 'i' loop across the parallel gangs; for CUDA programmers, think of gangs as the thread blocks in a grid. The iterations of the 'j' loop are spread across the workers; think of workers as the threads within a thread block. The reduction clause tells the compiler to accumulate the values of xx across the workers into a single sum.

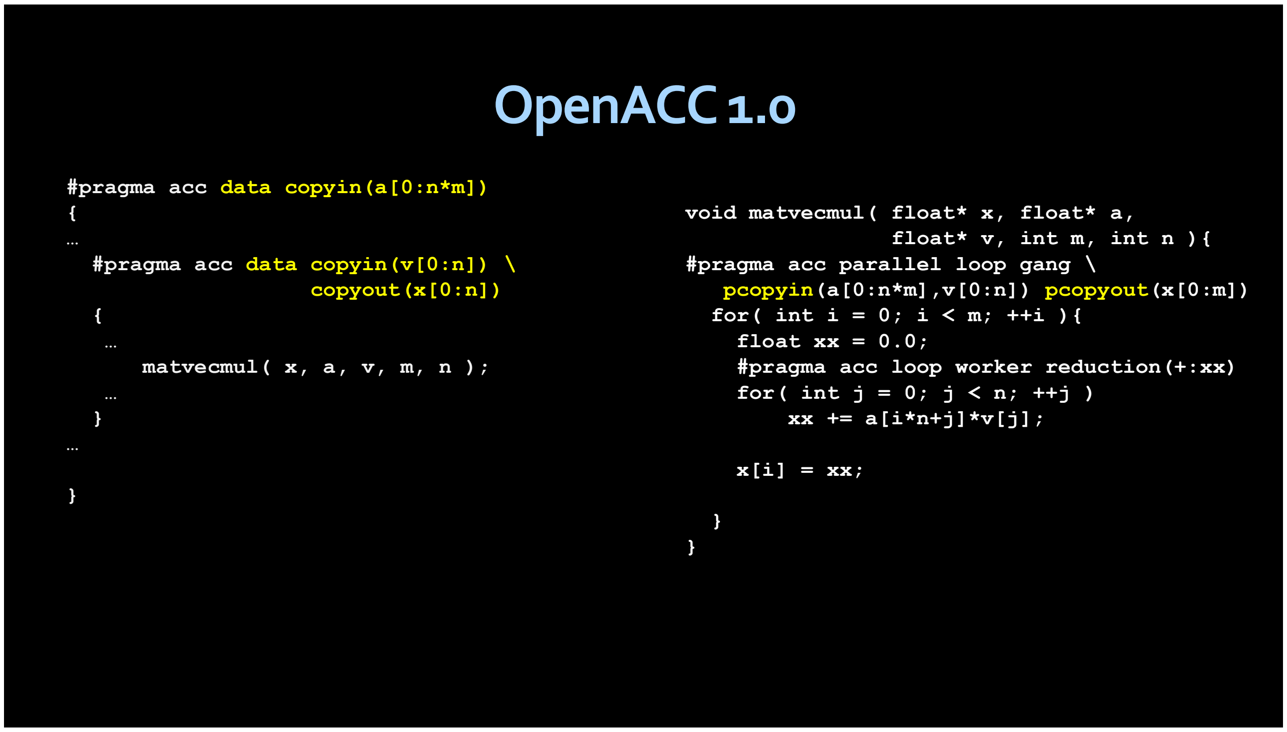
In this case, the compiler has to determine what parts of what arrays need to be moved to and from the GPU. This communication management is something that the PGI compiler can do in a simple case like this, but in many cases you, the programmer, will have to add annotations to manage the data traffic:



OpenACC 1.0

```
void matvecmul( float* x, float* a,
                float* v, int m, int n ){
#pragma acc parallel loop gang \
   copyin(a[0:n*m],v[0:n]) copyout(x[0:m])
  for( int i = 0; i < m; ++i ){
    float xx = 0.0;
    #pragma acc loop worker reduction(+:xx)
    for( int j = 0; j < n; ++j )
        xx += a[i*n+j]*v[j];

    x[i] = xx;

  }
}
```

The copyin clauses tell the compiler to copy the array 'a', starting at element 0 and continuing for 'n*m' elements, in to the GPU; similarly the vector 'v' from element 0 for 'n' elements. The vector 'x' is copied out from the GPU memory to the host memory. The first step in optimizing your GPU application is better management of the traffic between the host and device memories. In a simple case like this, we're moving n*m elements of 'a' and only doing n*m multiplies and adds; the compute intensity, the number of operations relative to the amount of data moved, is way too low, and it's unlikely that this routine will run faster than just leaving all the data on the host. To address this, OpenACC has 'data' constructs, which allow you, the programmer, to float the memory traffic outside of loops and routines to reduce the frequency of data traffic:



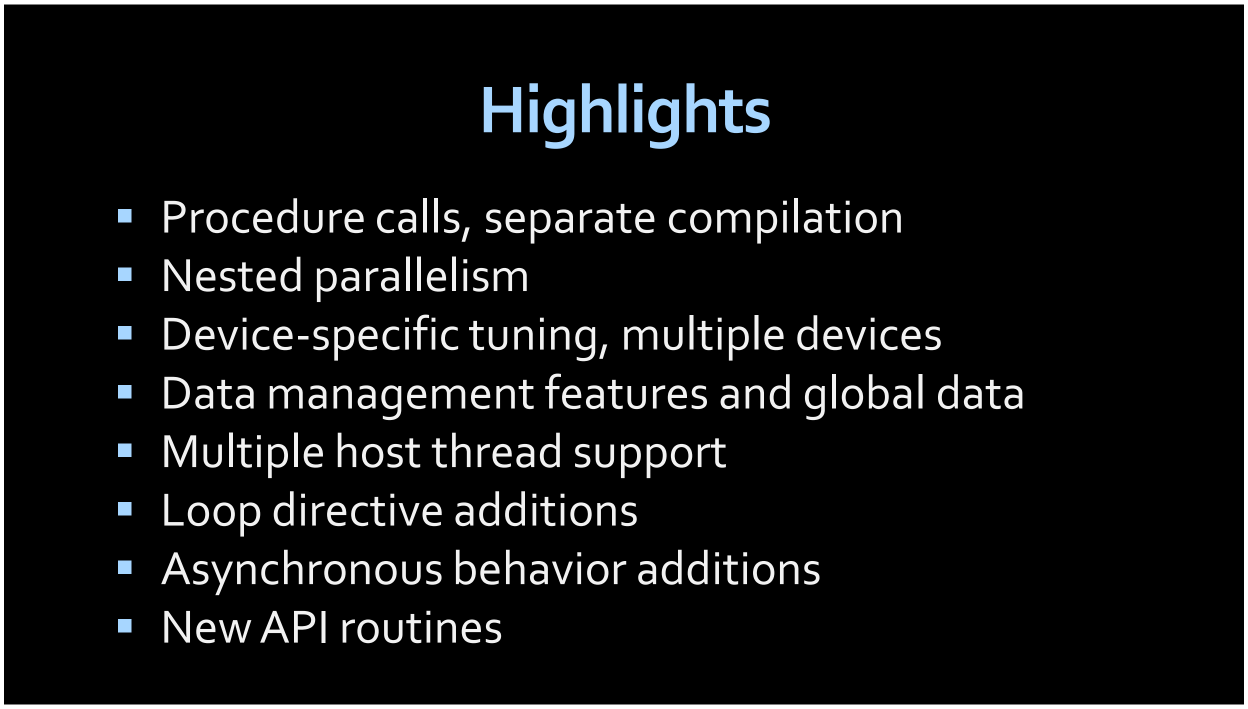
OpenACC 1.0

```
#pragma acc data copyin(a[0:n*m])
{
…
  #pragma acc data copyin(v[0:n]) \
                   copyout(x[0:n])
  {
   …
      matvecmul( x, a, v, m, n );
   …
  }
…

}
```

```
void matvecmul( float* x, float* a,
                float* v, int m, int n ){
#pragma acc parallel loop gang \
   pcopyin(a[0:n*m],v[0:n]) pcopyout(x[0:m])
  for( int i = 0; i < m; ++i ){
    float xx = 0.0;
    #pragma acc loop worker reduction(+:xx)
    for( int j = 0; j < n; ++j )
        xx += a[i*n+j]*v[j];

    x[i] = xx;

  }
}
```

Here, the calling routine does all the data traffic, then eventually calls the routine. The copyin and copyout clauses have changed to pcopyin and pcopyout, which are abbreviations for present_or_copyin and present_or_copyout. This means the implementation will test at runtime whether those arrays are already present on the device; if so, no data is allocated or moved and the routine will use the existing device data. If the arrays are not present, it will behave just like copyin and copyout, so this routine will work, though perhaps not fast, whether it's called from within that data region or not.

With that very brief introduction, let's discuss the additions going into OpenACC 2.0.

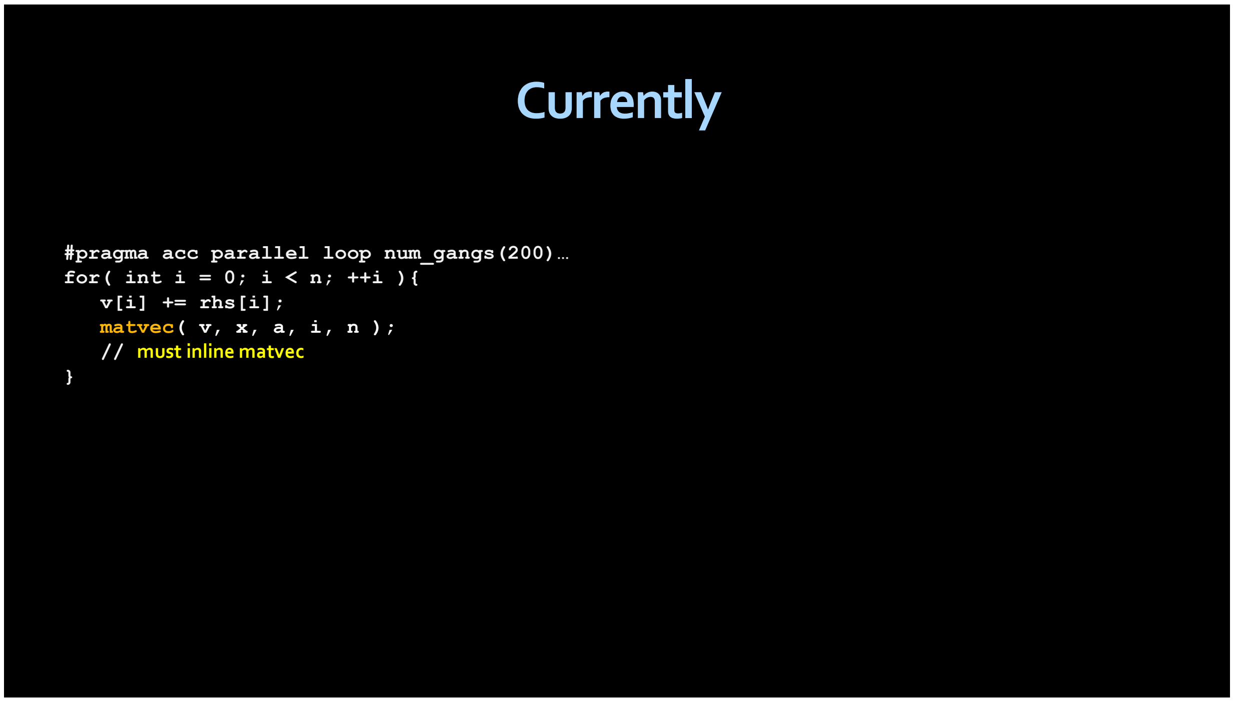


Highlights

- Procedure calls, separate compilation
- Nested parallelism
- Device-specific tuning, multiple devices
- Data management features and global data
- Multiple host thread support
- Loop directive additions
- Asynchronous behavior additions
- New API routines

I'll start by focusing on the first three items, the three most important features, then continue on to the other items. There are other changes and clarifications to the OpenACC spec; this is not intended to be comprehensive.

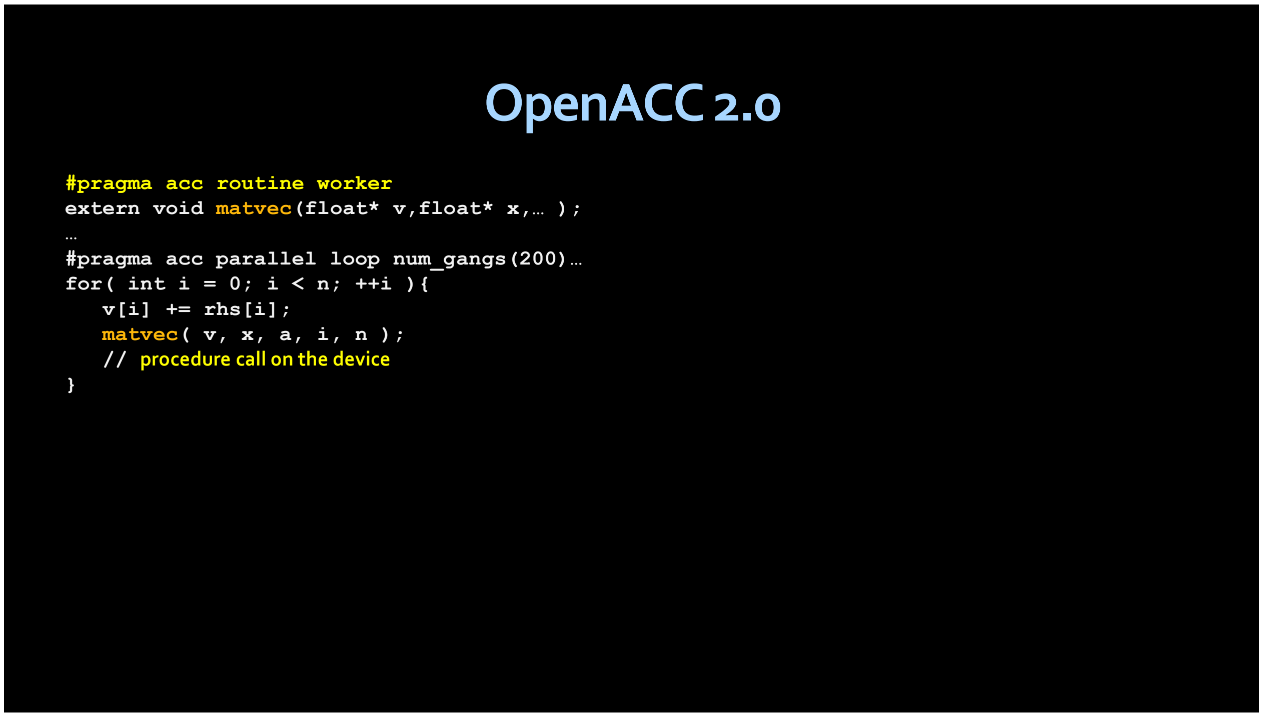
Procedure Calls, Separate Compilation



Currently

```
#pragma acc parallel loop num_gangs(200)…
for( int i = 0; i < n; ++i ){
   v[i] += rhs[i];
   matvec( v, x, a, i, n );
   // must inline matvec
}
```

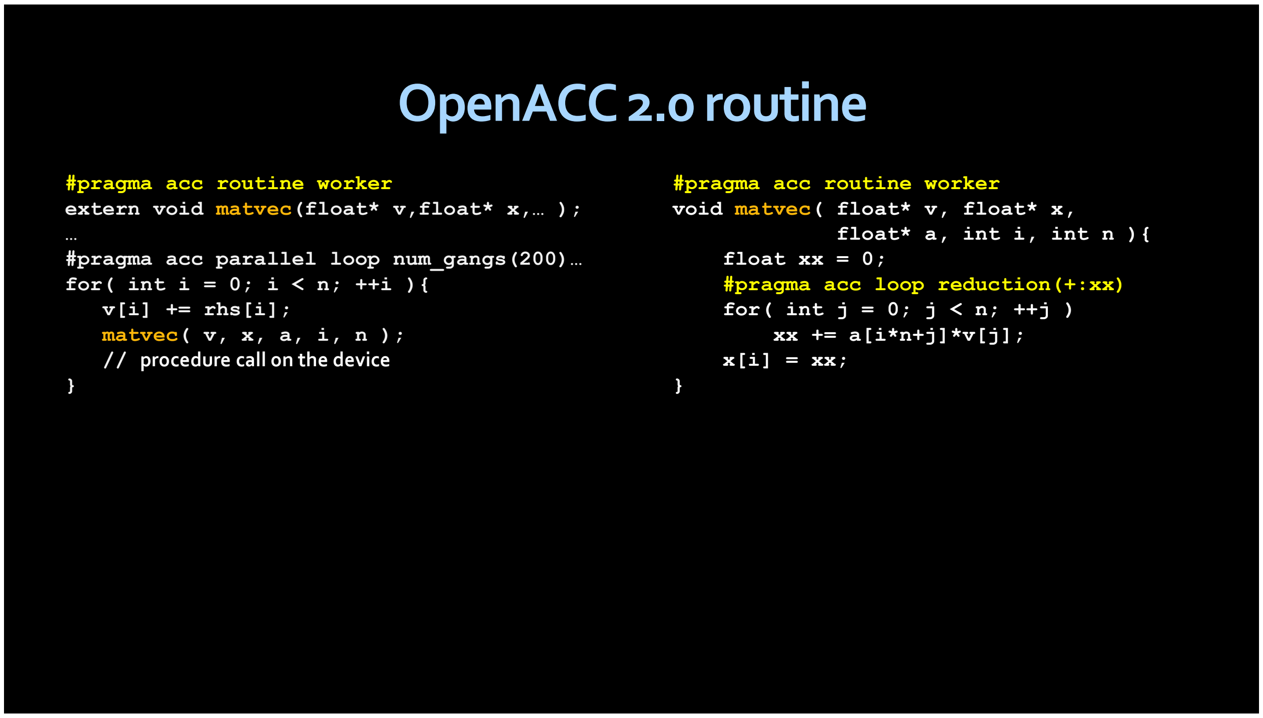
In OpenACC 1.0, procedure calls are very limited. Because the GPU devices had no linker, separate compilation was not supported. In fact, the NVIDIA Tesla (compute capability 1.x) devices didn't have a mechanism for procedure calls, so all procedure calls had to be inlined. This is bad. In fact, this is horrible. Procedures and separation compilation are one of the fundamental features of high level programming; they allow for libraries, modularity, and provide so many benefits that it's hard to justify a programming strategy without real procedure calls. Enabled by CUDA 5.0 features, OpenACC adds support for procedure calls. It's not quite as simple as just removing the restriction. Procedure may have "orphaned" loop directives. The compiler has to know whether that loop directive is to be mapped across gangs or workers or vector lanes. Also, the compiler has to know what procedures are needed on the device. So we've added the 'routine' directive:



OpenACC 2.0

```
#pragma acc routine worker
extern void matvec(float* v,float* x,… );
…
#pragma acc parallel loop num_gangs(200)…
for( int i = 0; i < n; ++i ){
   v[i] += rhs[i];
   matvec( v, x, a, i, n );
   // procedure call on the device
}
```

Here, the routine directive appears just above the prototype for the procedure. This tells the compiler that there will be a device copy of the routine 'matvec.' It also tells the compiler that there may be a loop in 'matvec' that targets the worker-level parallelism, so calls to 'matvec' may not appear within a worker loop. Essentially, it says that worker-level parallelism has to be reserved to the routine 'matvec,' and may not be used by the caller.



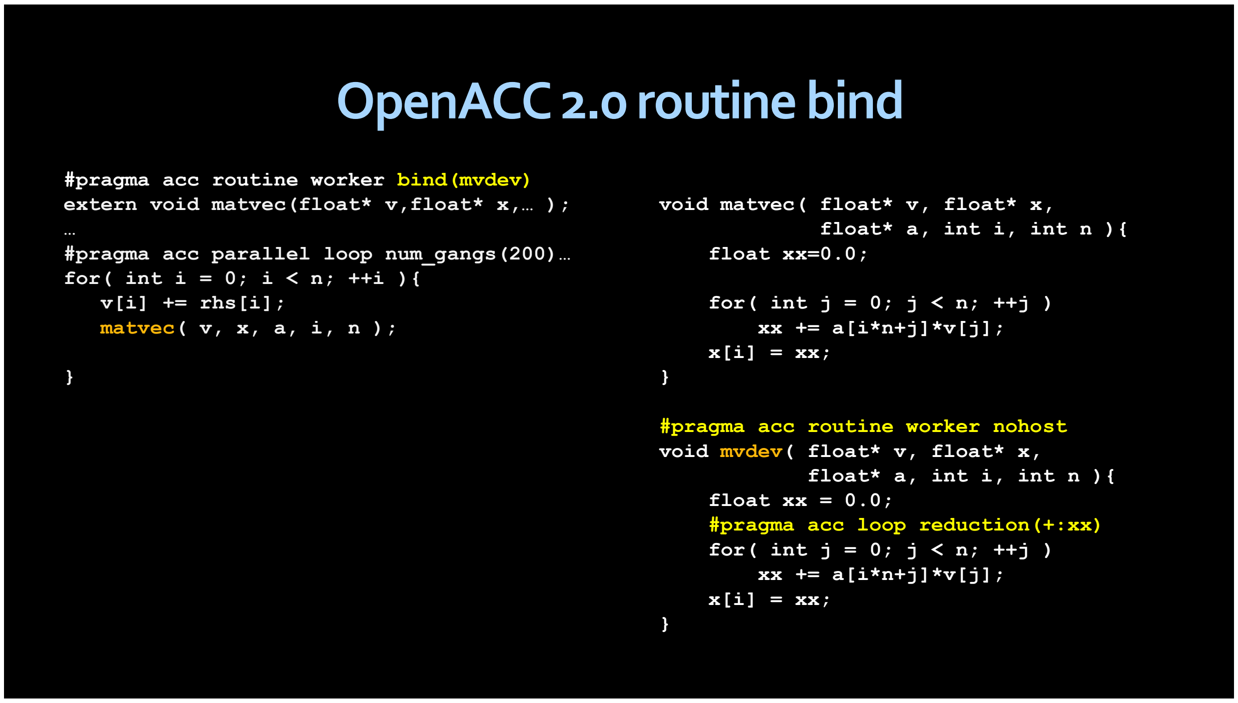
OpenACC 2.0 routine

```
#pragma acc routine worker
extern void matvec(float* v,float* x,… );
…
#pragma acc parallel loop num_gangs(200)…
for( int i = 0; i < n; ++i ){
   v[i] += rhs[i];
   matvec( v, x, a, i, n );
   // procedure call on the device
}
```

```
#pragma acc routine worker
void matvec( float* v, float* x,
             float* a, int i, int n ){
    float xx = 0;
    #pragma acc loop reduction(+:xx)
    for( int j = 0; j < n; ++j )
        xx += a[i*n+j]*v[j];
    x[i] = xx;
}
```

The file containing the actual routine must have a matching ‘routine’ directive. Here we have that routine with a loop directive inside the routine. The compiler will generate code twice for this routine, once for the host and again for the accelerator; when generating code for the accelerator, it will split the iterations of the ‘j’ loop across the workers within a gang.

There are other features of the routine directive as well. In this example, the directive has a ‘bind’ clause, that tells the compiler to call a routine with a different name when calling on the accelerator.



OpenACC 2.0 routine bind

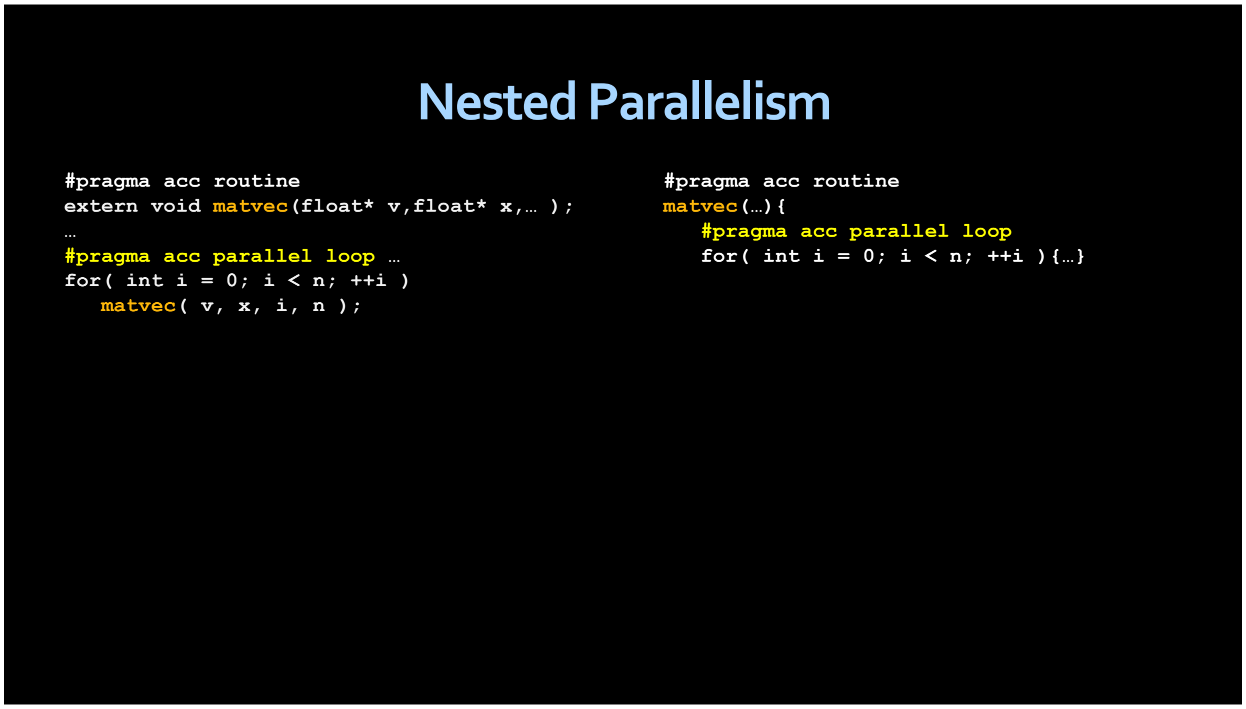
```
#pragma acc routine worker bind(mvdev)
extern void matvec(float* v,float* x,… );
…
#pragma acc parallel loop num_gangs(200)…
for( int i = 0; i < n; ++i ){
   v[i] += rhs[i];
   matvec( v, x, a, i, n );

}
```

```
void matvec( float* v, float* x,
             float* a, int i, int n ){
    float xx=0.0;

    for( int j = 0; j < n; ++j )
        xx += a[i*n+j]*v[j];
    x[i] = xx;
}

#pragma acc routine worker nohost
void mvdev( float* v, float* x,
            float* a, int i, int n ){
    float xx = 0.0;
    #pragma acc loop reduction(+:xx)
    for( int j = 0; j < n; ++j )
        xx += a[i*n+j]*v[j];
    x[i] = xx;
}
```

In this case, the routine ‘matvec’ is only compiled for the host. For the device, the routine ‘mvdev’ is compiled, and because of the ‘nohost’ clause, ‘mvdev’ is not compiled for the host. The call to ‘matvec’ is turned into a call to ‘mvdev’ when compiled for the accelerator.

Nested Parallelism

The second important feature of OpenACC 2.0 is nested parallelism. This takes advantage of the new CUDA feature that allows kernel launches from within a kernel.

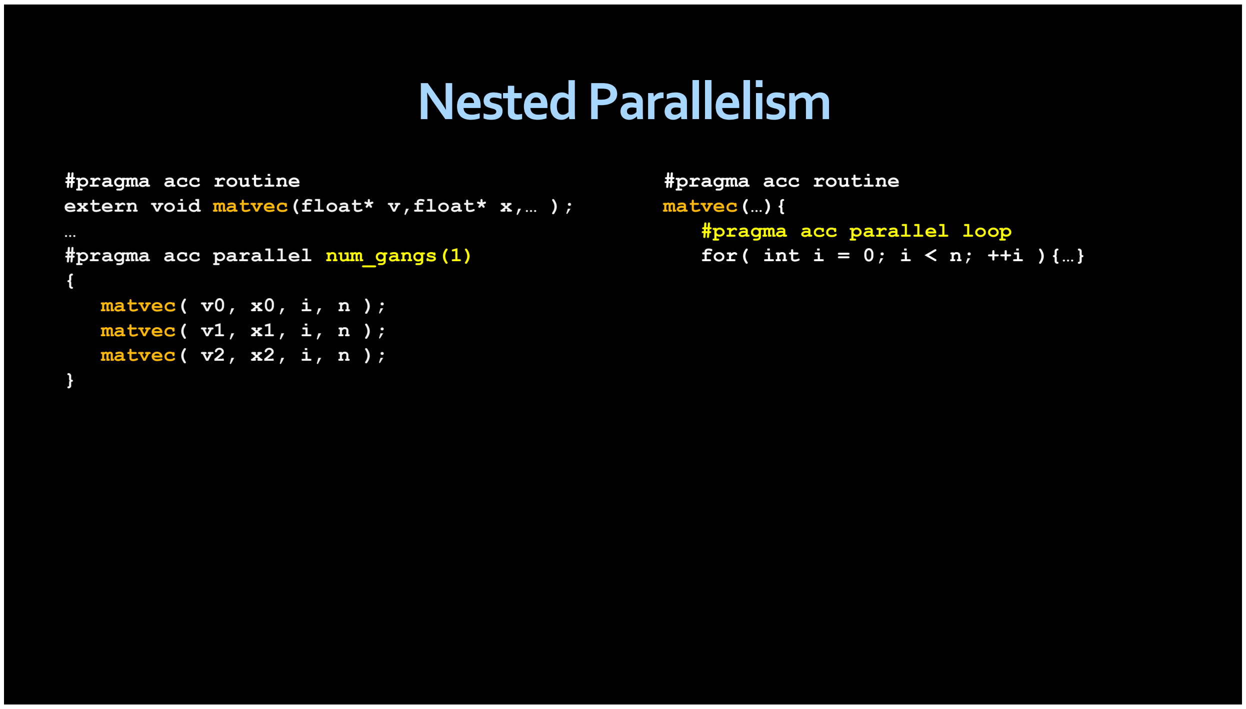


Nested Parallelism

```
#pragma acc routine
extern void matvec(float* v,float* x,… );
…
#pragma acc parallel loop …
for( int i = 0; i < n; ++i )
   matvec( v, x, i, n );
```

```
#pragma acc routine
matvec(…){
   #pragma acc parallel loop
   for( int i = 0; i < n; ++i ){…}
```

This allows a user to write a routine that has OpenACC parallelism and compile that routine for both the host and the accelerator. In this case, the nested parallelism appears within a parallel region, but I think a likely more common mechanism will be to start a single thread on the accelerator and launch all the parallelism from that single device thread. The goal here is to move as much of the application over to the accelerator, to decouple the device from the host and to minimize communication between the two.



Nested Parallelism

```
#pragma acc routine
extern void matvec(float* v,float* x,… );
…
#pragma acc parallel num_gangs(1)
{
   matvec( v0, x0, i, n );
   matvec( v1, x1, i, n );
   matvec( v2, x2, i, n );
}
```

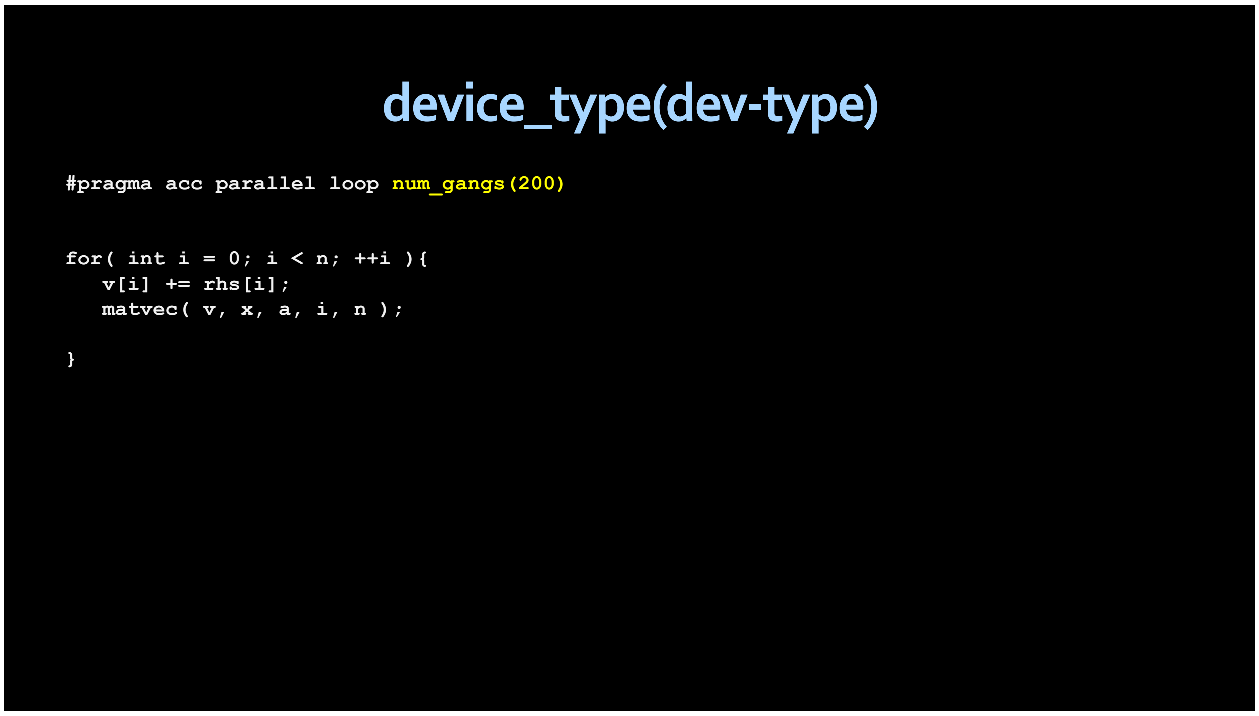
```
#pragma acc routine
matvec(…){
   #pragma acc parallel loop
   for( int i = 0; i < n; ++i ){…}
```

In this example, the outer ‘parallel’ construct only starts a single thread, so perhaps it’s misnamed, but the implementation of the nested device kernels is nested parallelism.

One other change related to nested parallelism is the use of ‘host’ and ‘device’. With nested parallelism, the thread that creates the parallelism may not be the host thread. Some terms, like ‘update host,’ assume a host-centric programming model. Throughout the specification, the term ‘host’ has been reserved to those situations that can only occur on the host thread; otherwise, the term ‘local’ or ‘local thread’ is used. In particular, the ‘update host’ directive has been renamed ‘update local’, though all implementations will still accept ‘update host’ as a synonym.

Device-Specfic Tuning

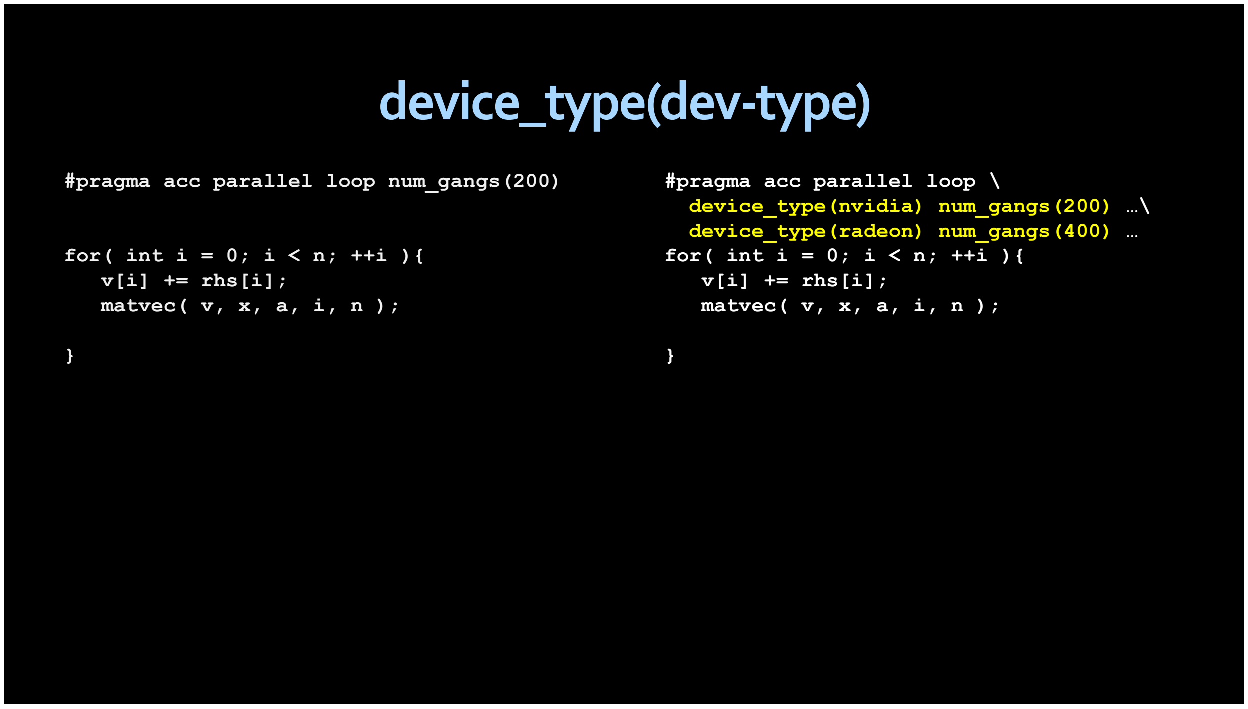
The third important feature is support for tuning for multiple devices in a single program. The tuning clauses are things like the ‘num_gangs’ or ‘vector_length’ clause on the parallel construct, the loop ‘gang’, ‘worker’ and ‘vector’ clauses, and so on. As OpenACC is implemented on more device types, we will want to tune these for each device.



device_type(dev-type)

```
#pragma acc parallel loop num_gangs(200)

for( int i = 0; i < n; ++i ){
   v[i] += rhs[i];
   matvec( v, x, a, i, n );

}
```

The mechanism in OpenACC 2.0 for multiple device types is the ‘device_type’ clause (which may be abbreviated to ‘dtype’).



device_type(dev-type)

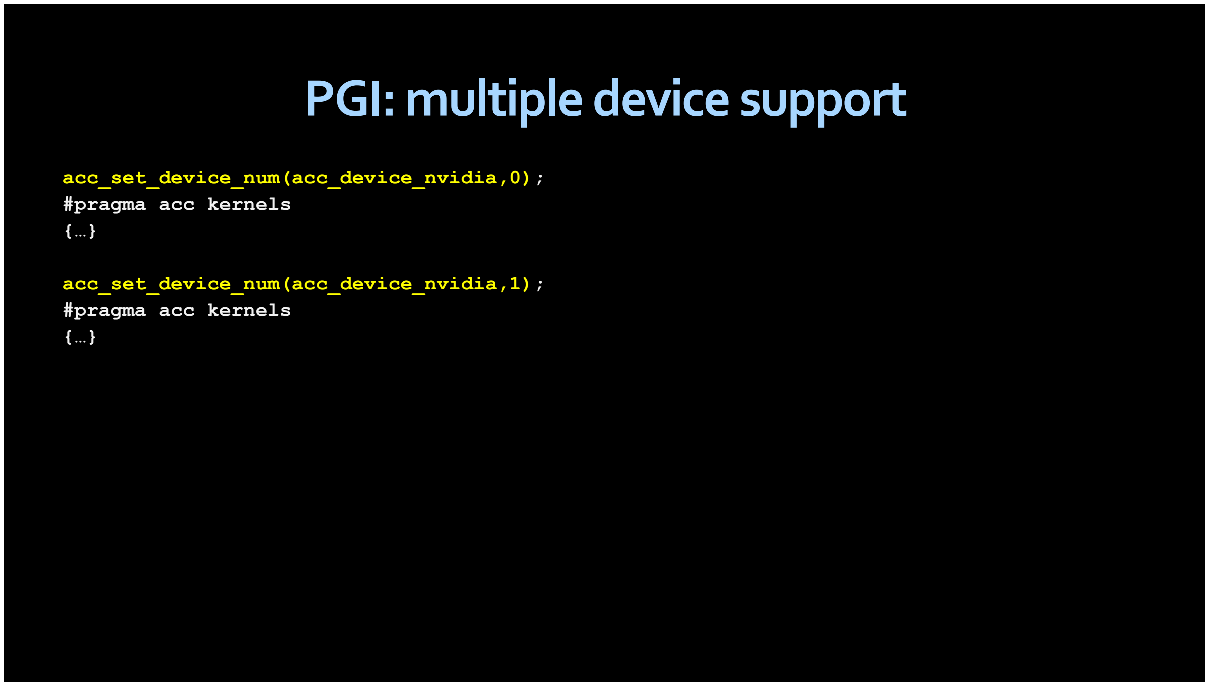
```
#pragma acc parallel loop num_gangs(200)


for( int i = 0; i < n; ++i ){
   v[i] += rhs[i];
   matvec( v, x, a, i, n );

}
```

```
#pragma acc parallel loop \
  device_type(nvidia) num_gangs(200) …\
  device_type(radeon) num_gangs(400) …
for( int i = 0; i < n; ++i ){
   v[i] += rhs[i];
   matvec( v, x, a, i, n );

}
```

Here, this example tells the compiler to create 200 gangs on an NVIDIA target, but 400 gangs on a RADEON target. A clause that follows a ‘device_type’ clause only applies to that device type, up to the next ‘device_type’ clause. A ‘device_type(*)’ gives default values for any device type not explicitly named. Note that the data clauses (copyin, copyout, present) may not be device-specific, so may not follow a device_type clause.

Multiple Devices (PGI-specific)

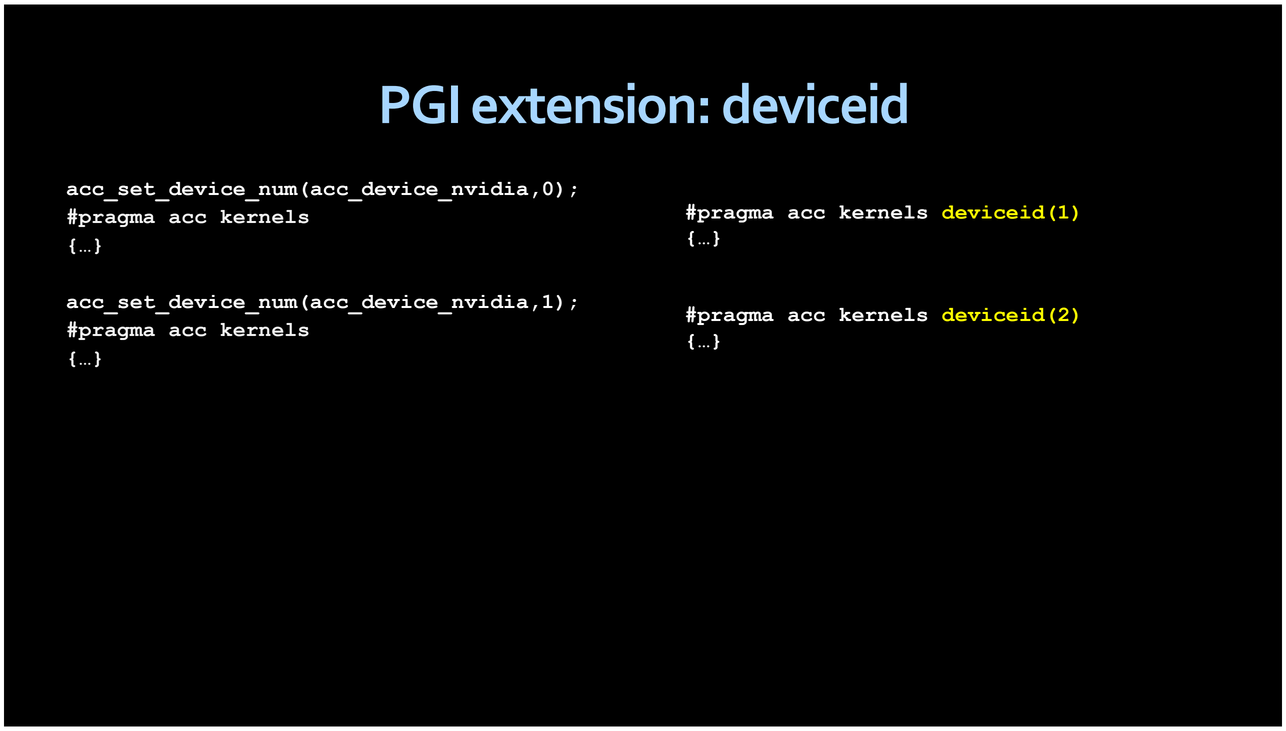
I’m going to take a short break here to introduce some PGI-specific features, having to do with multiple devices and device types. First the 2013 PGI compilers will support multiple devices from a single thread or program. The thread will have a ‘current’ accelerator. All data traffic and parallel regions will be launched to the current accelerator. The current accelerator can change dynamically with a call to the API routine:



PGI: multiple device support

```
acc_set_device_num(acc_device_nvidia,0);
#pragma acc kernels
{…}

acc_set_device_num(acc_device_nvidia,1);
#pragma acc kernels
{…}
```

PGI has added an extension to these directives, a ‘deviceid’ clause, which lets you change the device to use right on the directive.

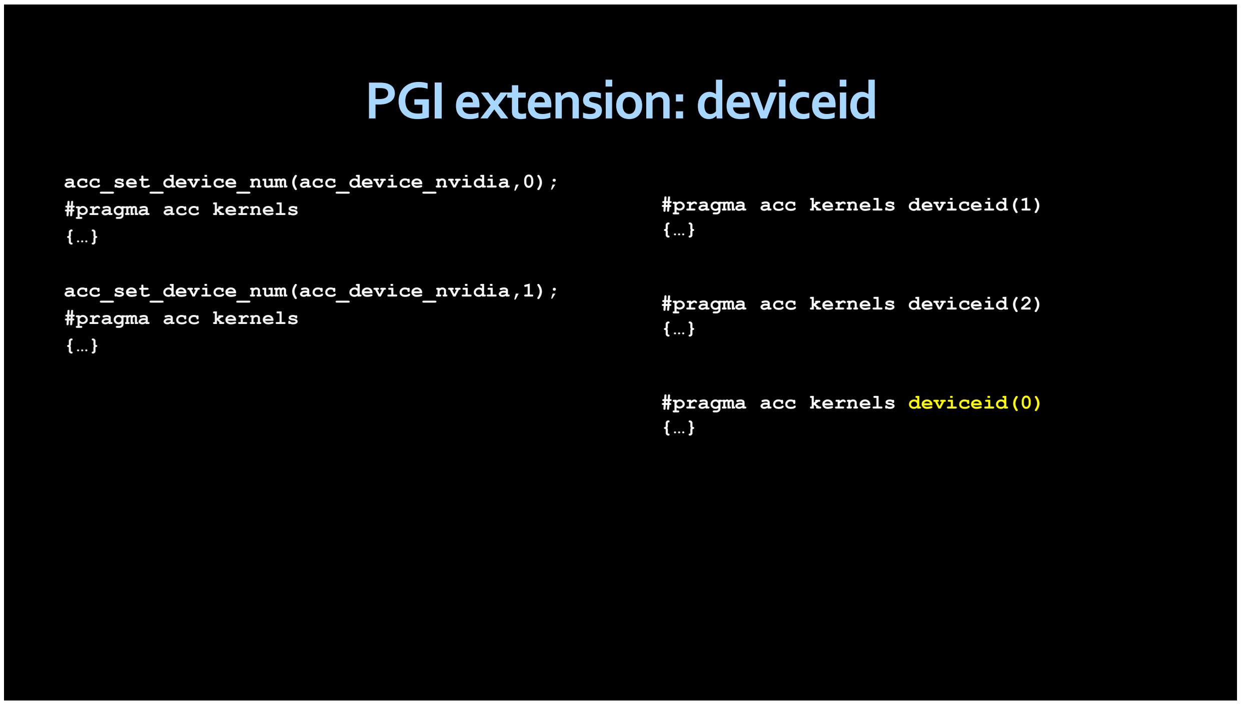


PGI extension: deviceid

```
acc_set_device_num(acc_device_nvidia,0);
#pragma acc kernels
{…}

acc_set_device_num(acc_device_nvidia,1);
#pragma acc kernels
{…}
```

```
#pragma acc kernels deviceid(1)
{…}

#pragma acc kernels deviceid(2)
{…}
```

Note that the numbering for device ID starts at one, so is not the same as the CUDA device numbering. Device ID zero is special; deviceid(0) means to use the ‘current’ deviceid for this thread.

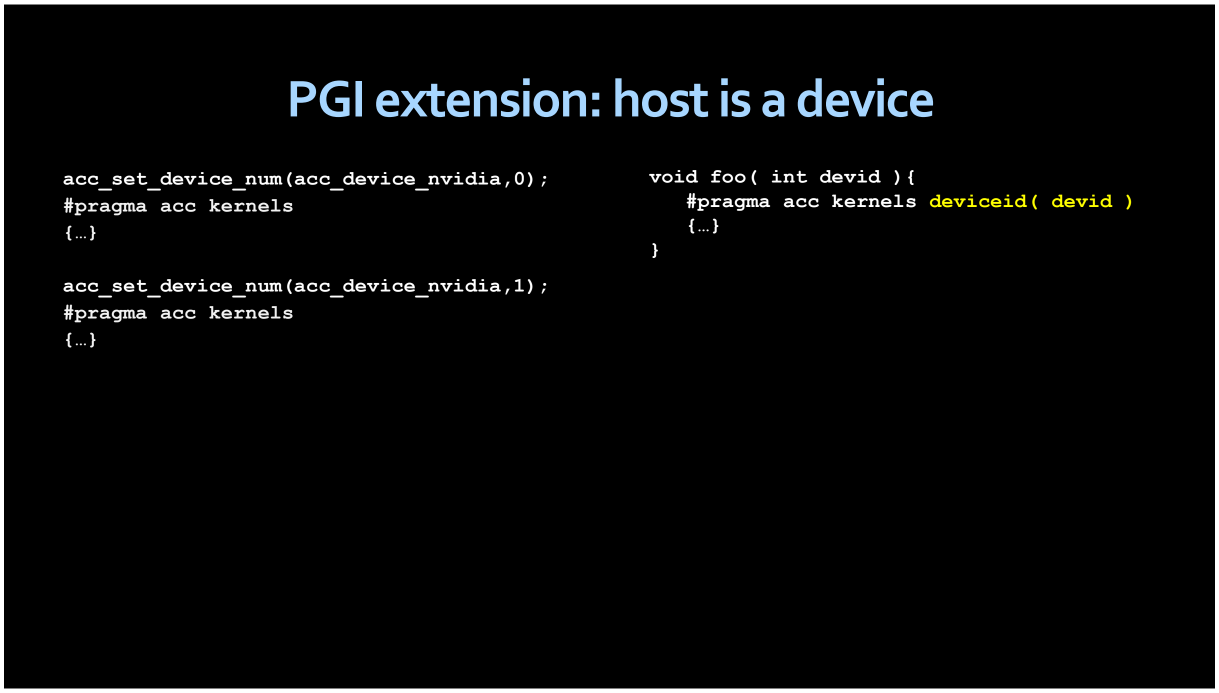


PGI extension: deviceid

```
acc_set_device_num(acc_device_nvidia,0);
#pragma acc kernels
{…}

acc_set_device_num(acc_device_nvidia,1);
#pragma acc kernels
{…}
```

```
#pragma acc kernels deviceid(1)
{…}

#pragma acc kernels deviceid(2)
{…}

#pragma acc kernels deviceid(0)
{…}
```

In addition, the host itself appears as a device and can be selected dynamically:



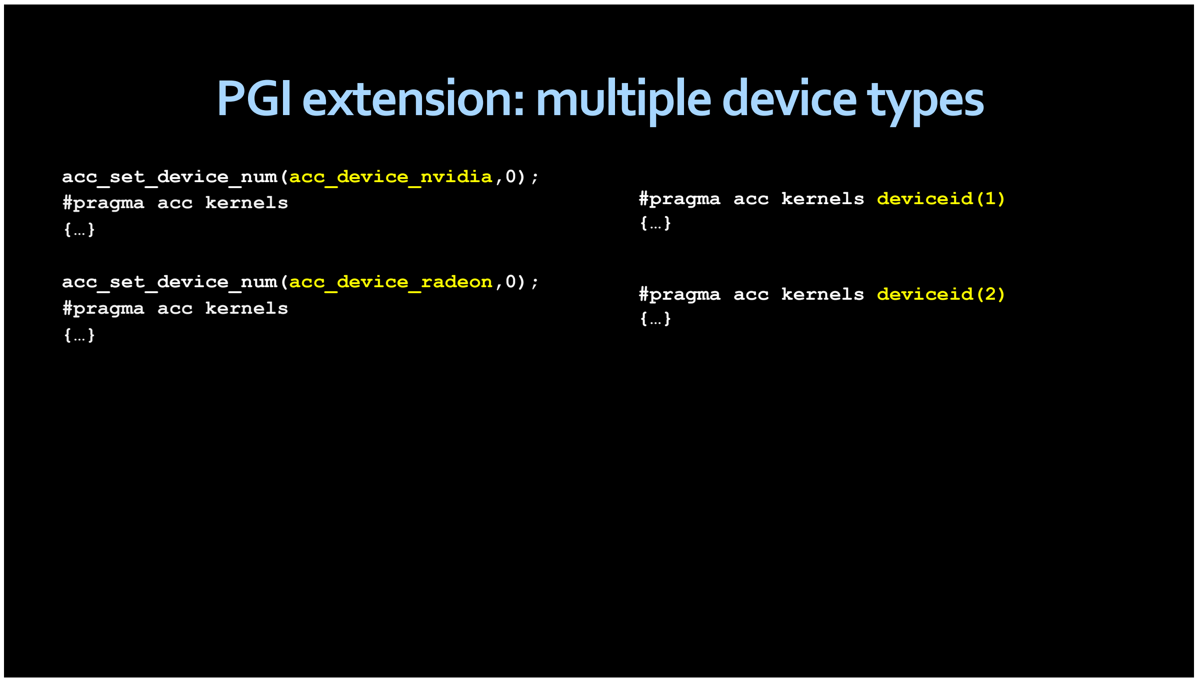
PGI extension: host is a device

```
acc_set_device_num(acc_device_nvidia,0);
#pragma acc kernels
{…}

acc_set_device_num(acc_device_nvidia,1);
#pragma acc kernels
{…}
```

```
void foo( int devid ){
   #pragma acc kernels deviceid( devid )
   {…}
}
```

The routine ‘foo’ is called with the argument that specifies the device ID to use for that call.

PGI is also adding support for multiple device types, in a single executable program. This allows you to either deliver a single binary that will work for different environments, or even a single binary that can split work across multiple devices, even from different vendors.

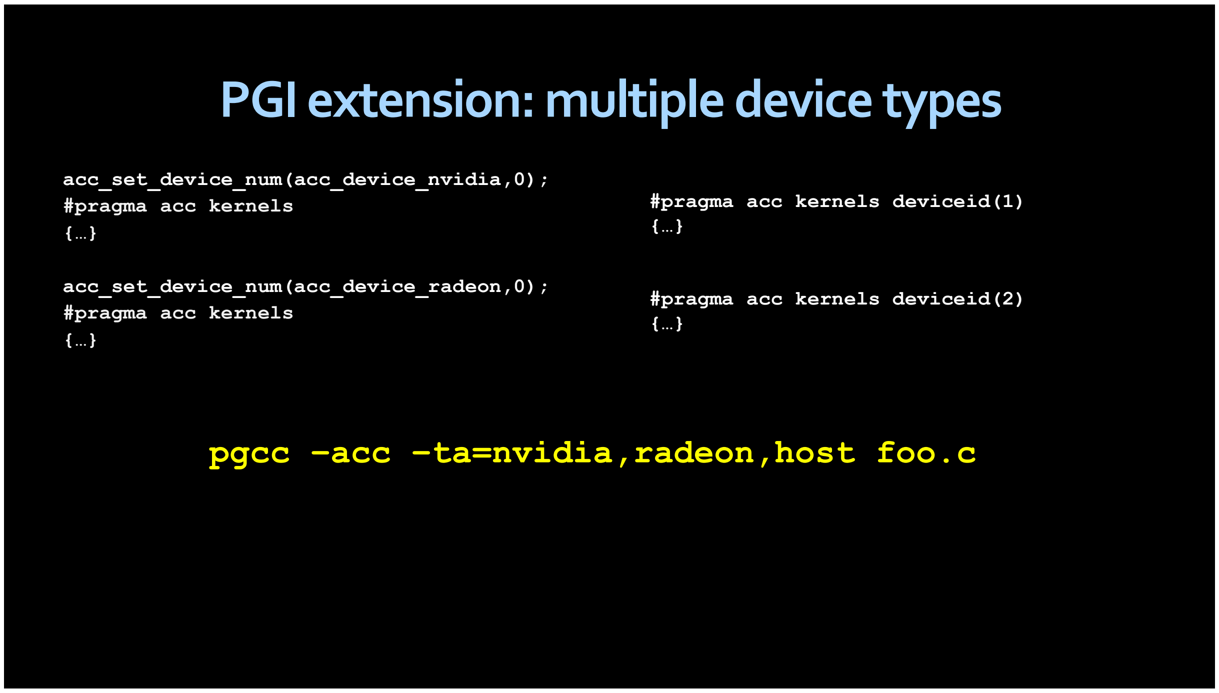


PGI extension: multiple device types

```
acc_set_device_num(acc_device_nvidia,0);
#pragma acc kernels
{…}

acc_set_device_num(acc_device_radeon,0);
#pragma acc kernels
{…}
```

```
#pragma acc kernels deviceid(1)
{…}

#pragma acc kernels deviceid(2)
{…}
```

To preempt an obvious question, there is no predefined numbering of device vendor and device number to device ID. It depends on how the operating system orders the devices, and can be modified by setting environment variables appropriately. Compiling for multiple devices will be as easy as setting the PGI 'target-accelerator' flag:



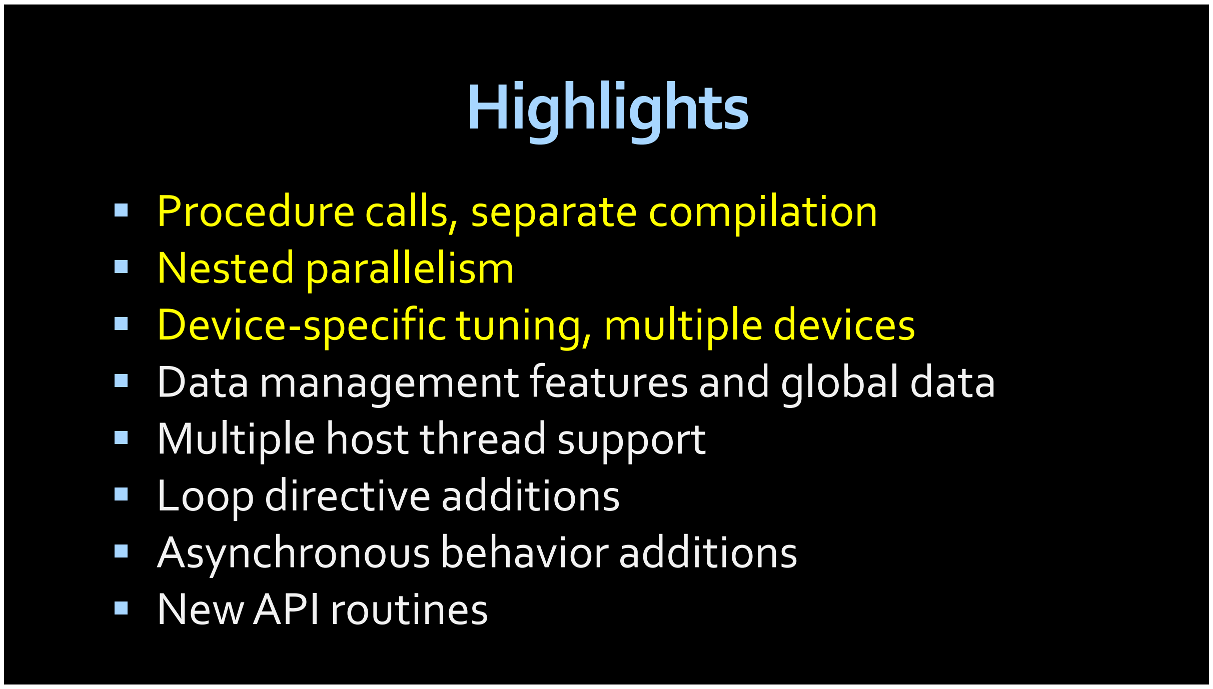
PGI extension: multiple device types

```
acc_set_device_num(acc_device_nvidia,0);
#pragma acc kernels
{…}

acc_set_device_num(acc_device_radeon,0);
#pragma acc kernels
{…}
```

```
#pragma acc kernels deviceid(1)
{…}

#pragma acc kernels deviceid(2)
{…}
```

```
pgcc –acc –ta=nvidia,radeon,host foo.c
```

Now we're through the three important new features. Let's look at some other new features:

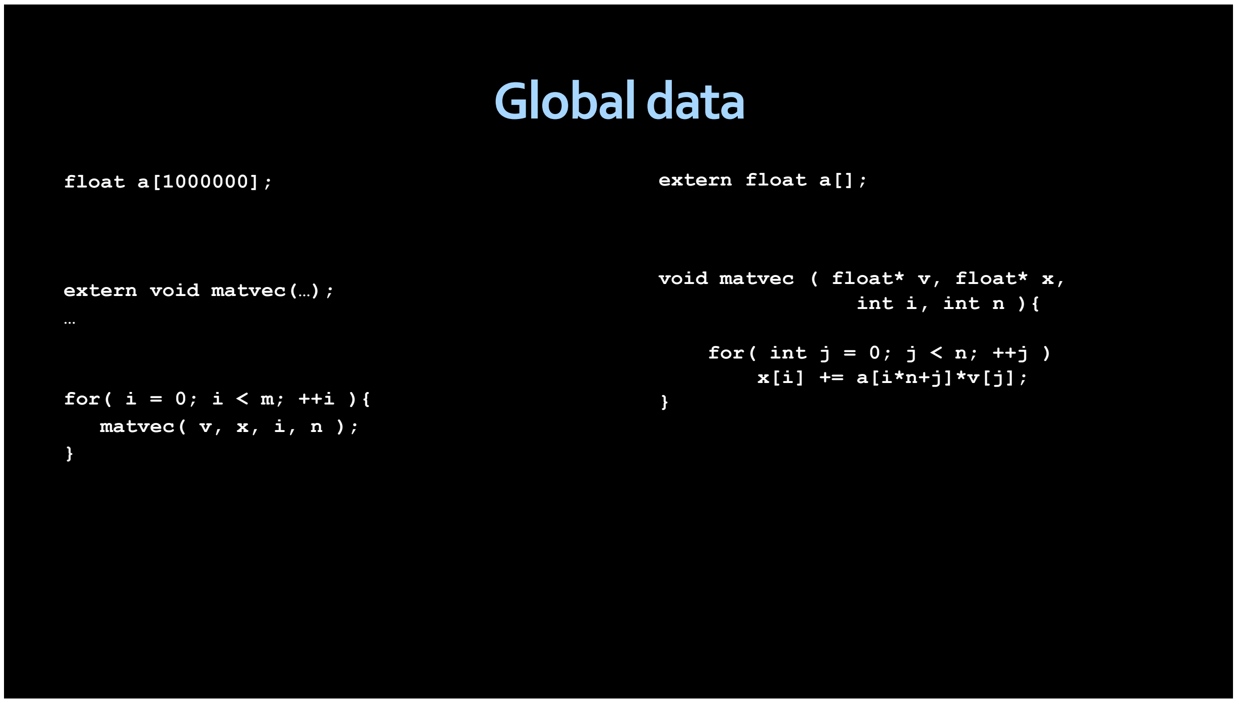


Highlights

- Procedure calls, separate compilation
- Nested parallelism
- Device-specific tuning, multiple devices
- Data management features and global data
- Multiple host thread support
- Loop directive additions
- Asynchronous behavior additions
- New API routines

Global Data

When we add separate compilation, we need support for global (external) variables.

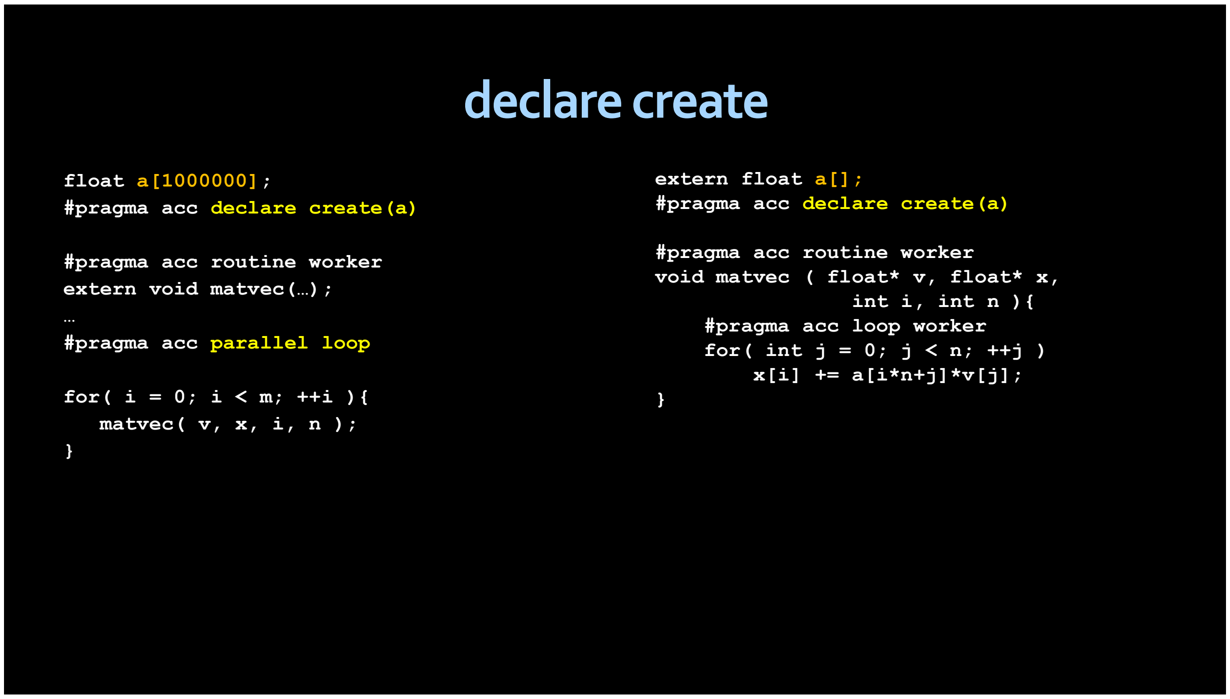


Global data

```
float a[1000000];


extern void matvec(…);
…


for( i = 0; i < m; ++i ){
   matvec( v, x, i, n );
}
```

```
extern float a[];


void matvec ( float* v, float* x,
                int i, int n ){

    for( int j = 0; j < n; ++j )
        x[i] += a[i*n+j]*v[j];
}
```

OpenACC adds four mechanisms to tune global data access across procedures using the ‘declare’ directive. The first three are for static global data.

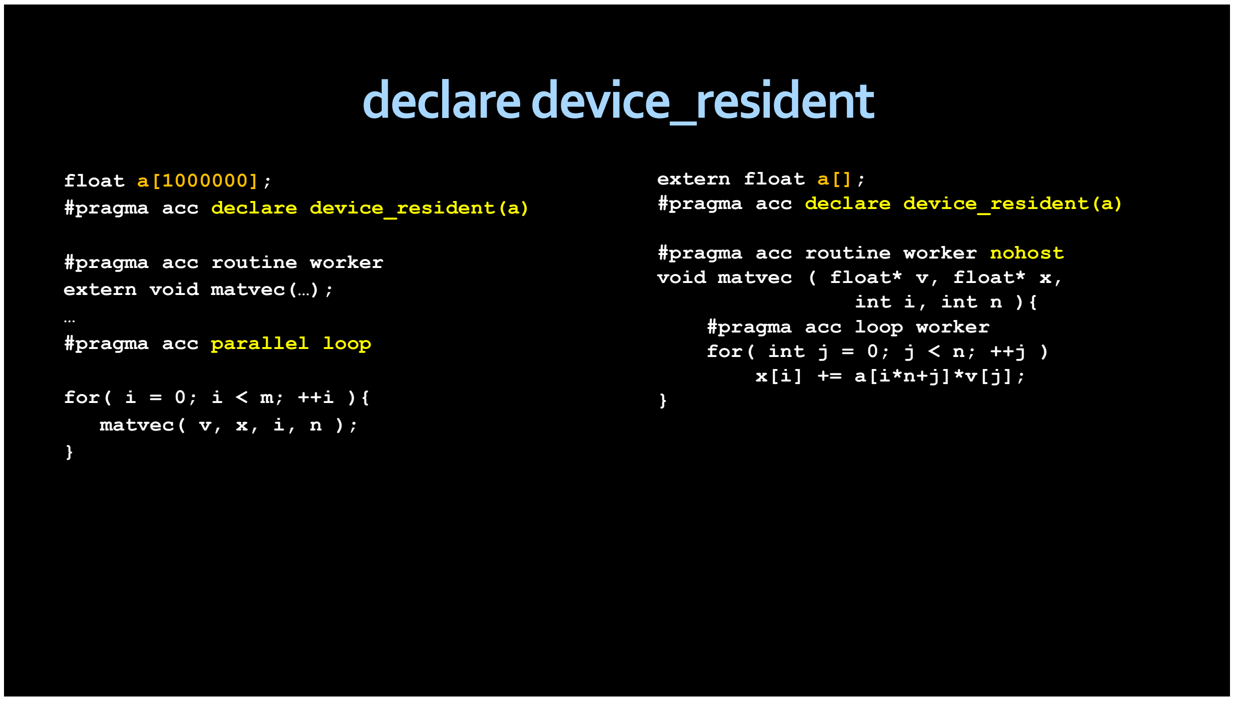


declare create

```
float a[1000000];
#pragma acc declare create(a)

#pragma acc routine worker
extern void matvec(…);
…
#pragma acc parallel loop

for( i = 0; i < m; ++i ){
   matvec( v, x, i, n );
}
```

```
extern float a[];
#pragma acc declare create(a)

#pragma acc routine worker
void matvec ( float* v, float* x,
                int i, int n ){
    #pragma acc loop worker
    for( int j = 0; j < n; ++j )
        x[i] += a[i*n+j]*v[j];
}
```

The ‘declare create’ directive tells the compiler to create the static data on the device as well as on the host. If there are multiple devices, each device will get a copy. The host and device copies of the array may have to be kept coherent using ‘ update’ directives.

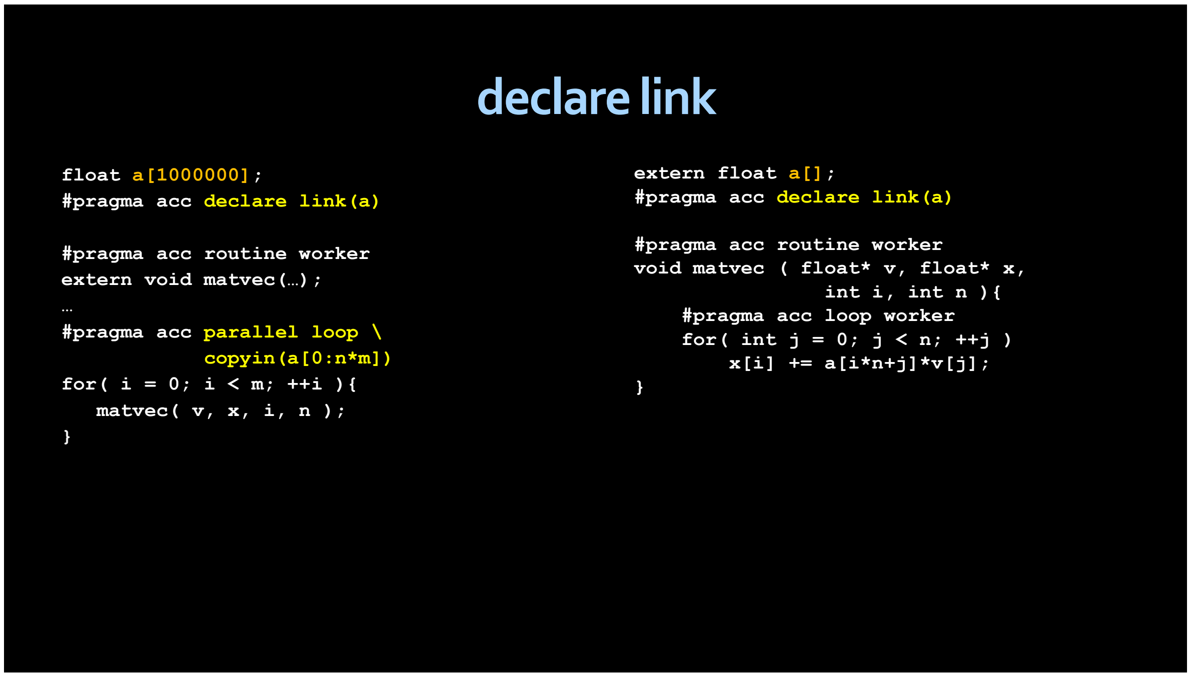


declare device_resident

```
float a[1000000];
#pragma acc declare device_resident(a)

#pragma acc routine worker
extern void matvec(…);
…
#pragma acc parallel loop

for( i = 0; i < m; ++i ){
   matvec( v, x, i, n );
}
```

```
extern float a[];
#pragma acc declare device_resident(a)

#pragma acc routine worker nohost
void matvec ( float* v, float* x,
                int i, int n ){
    #pragma acc loop worker
    for( int j = 0; j < n; ++j )
        x[i] += a[i*n+j]*v[j];
}
```

The ‘declare device_resident’ directive tells the compiler to create a device copy of the data, but not to allocate the data on the host. This data can only be accessed in device routines (‘nohost’) or inside compute regions.

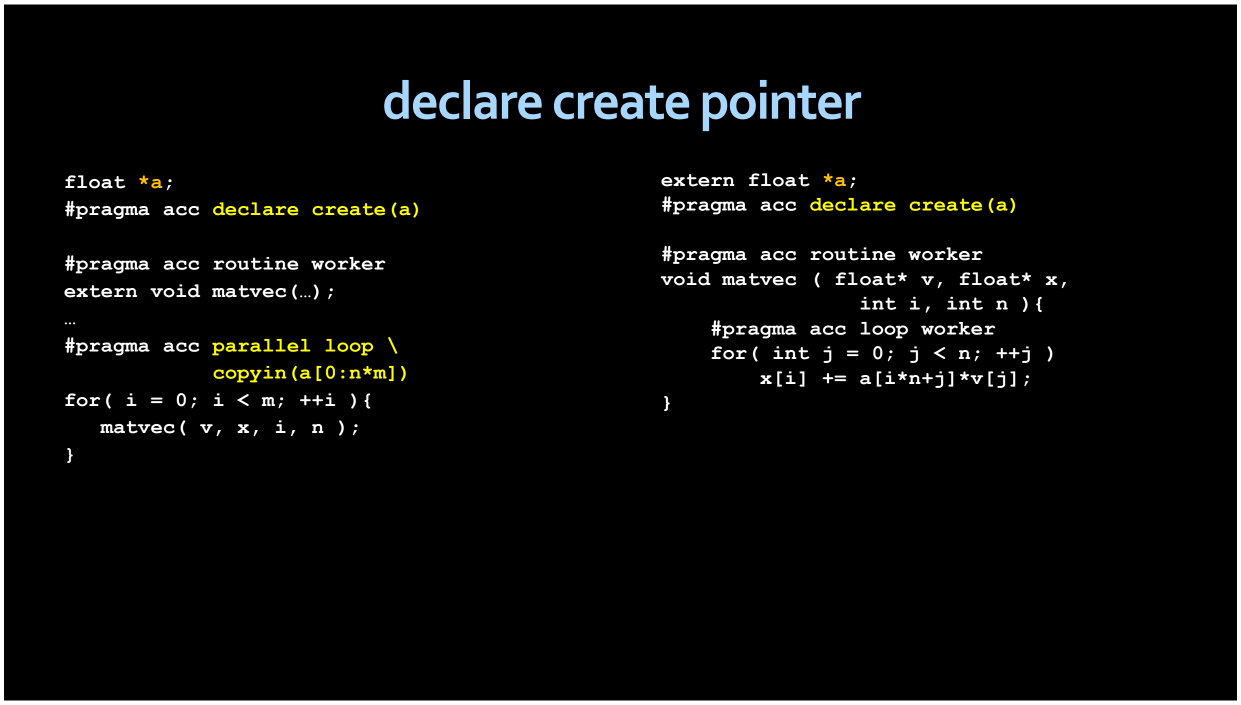


declare link

```
float a[1000000];
#pragma acc declare link(a)

#pragma acc routine worker
extern void matvec(…);
…
#pragma acc parallel loop \
            copyin(a[0:n*m])
for( i = 0; i < m; ++i ){
   matvec( v, x, i, n );
}
```

```
extern float a[];
#pragma acc declare link(a)

#pragma acc routine worker
void matvec ( float* v, float* x,
                int i, int n ){
    #pragma acc loop worker
    for( int j = 0; j < n; ++j )
        x[i] += a[i*n+j]*v[j];
}
```

The ‘declare link’ directive tells the compiler to generate the data on the host and to generate a link to the data on the device. When generating code for the device, accesses to the array will be generated as if the array were accessed through a pointer. The pointer gets filled by OpenACC data clauses, such as the ‘copyin’ in the example above. When a data clause is given for global data with ‘declare link’, the data is allocated and copied, as before, and the link on the device is filled in with the address to the data.



declare create pointer

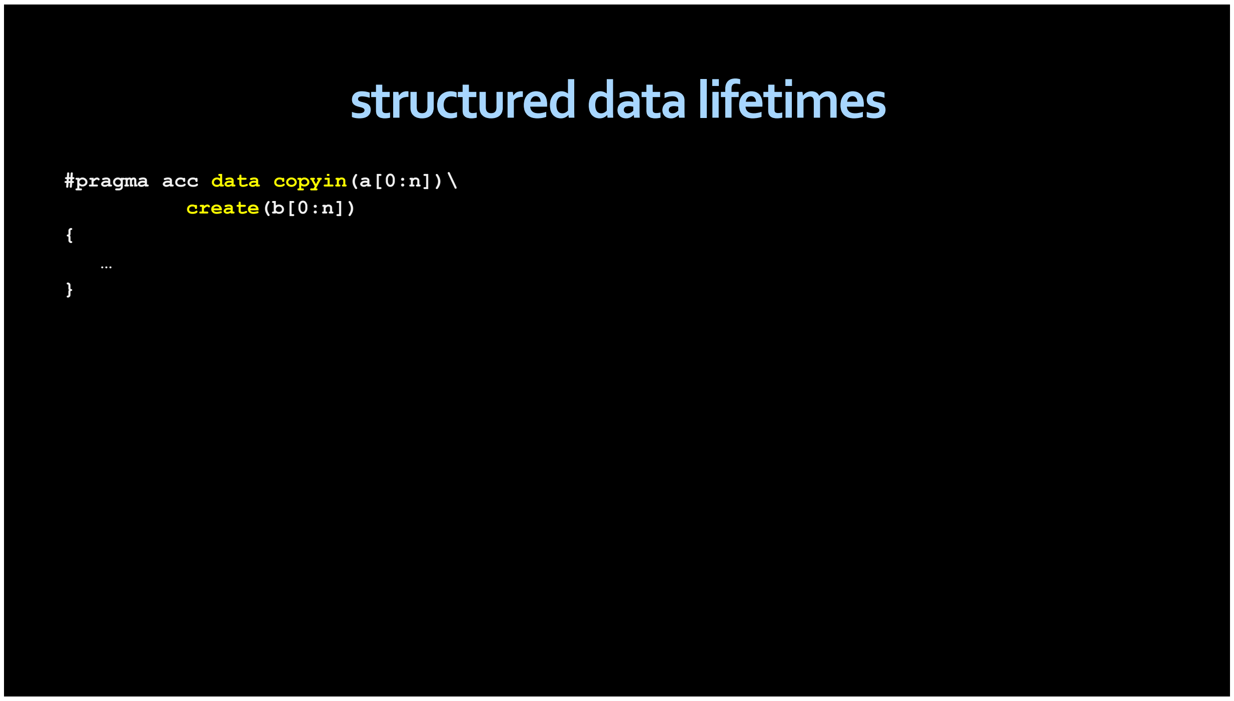
```
float *a;
#pragma acc declare create(a)

#pragma acc routine worker
extern void matvec(…);
…
#pragma acc parallel loop \
            copyin(a[0:n*m])
for( i = 0; i < m; ++i ){
   matvec( v, x, i, n );
}
```

```
extern float *a;
#pragma acc declare create(a)

#pragma acc routine worker
void matvec ( float* v, float* x,
                int i, int n ){
    #pragma acc loop worker
    for( int j = 0; j < n; ++j )
        x[i] += a[i*n+j]*v[j];
}
```

The final mechanism is used for global pointers; we want the pointer itself to be allocated statically on both the host and device. Just as the programmer is responsible for allocating and filling the data on the host, the programmer is responsible for managing the device copy of the data. Here, when a global pointer is used in a data clause, the compiler generates code to allocate and copy the data, and to fill in the global pointer on the device.

Data Management

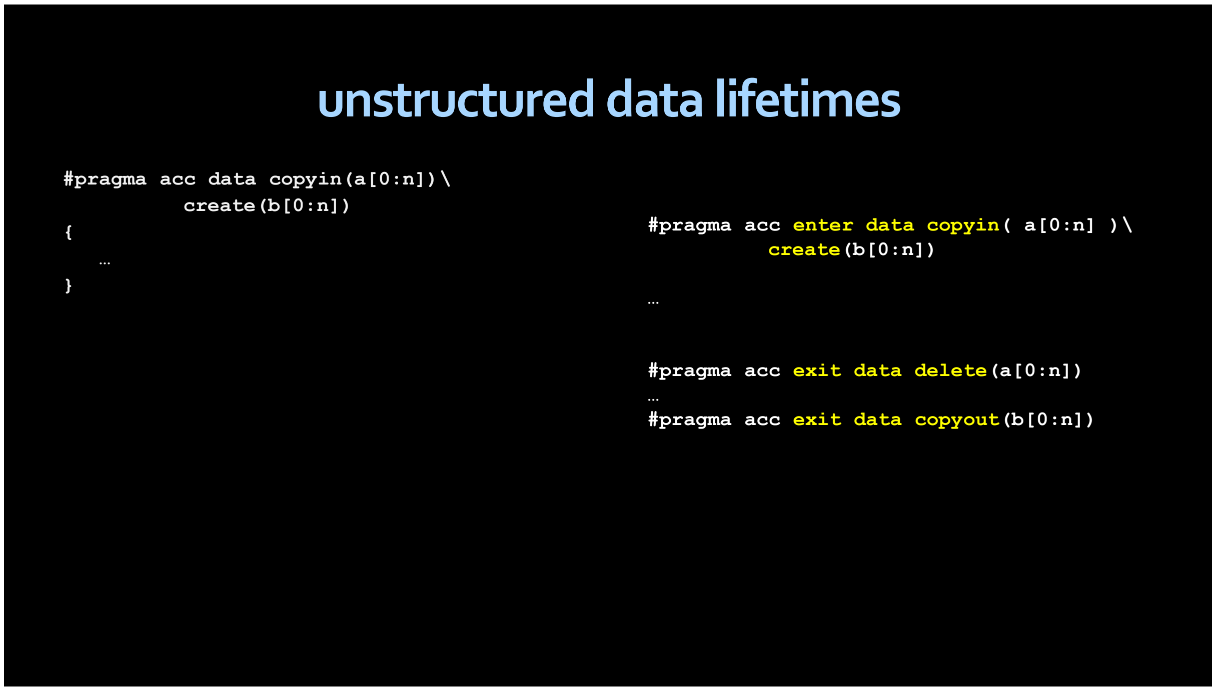
OpenACC uses the data construct to allow the programmer to manage data lifetimes on the device. This promotes structured programming, but not all data lifetimes are easily amenable to structured lifetimes.



structured data lifetimes

```
#pragma acc data copyin(a[0:n])\
          create(b[0:n])
{
   …
}
```

OpenACC 2.0 adds two new directives, ‘enter data’ and ‘exit data,’ which act much like the beginning and the end of a data region.

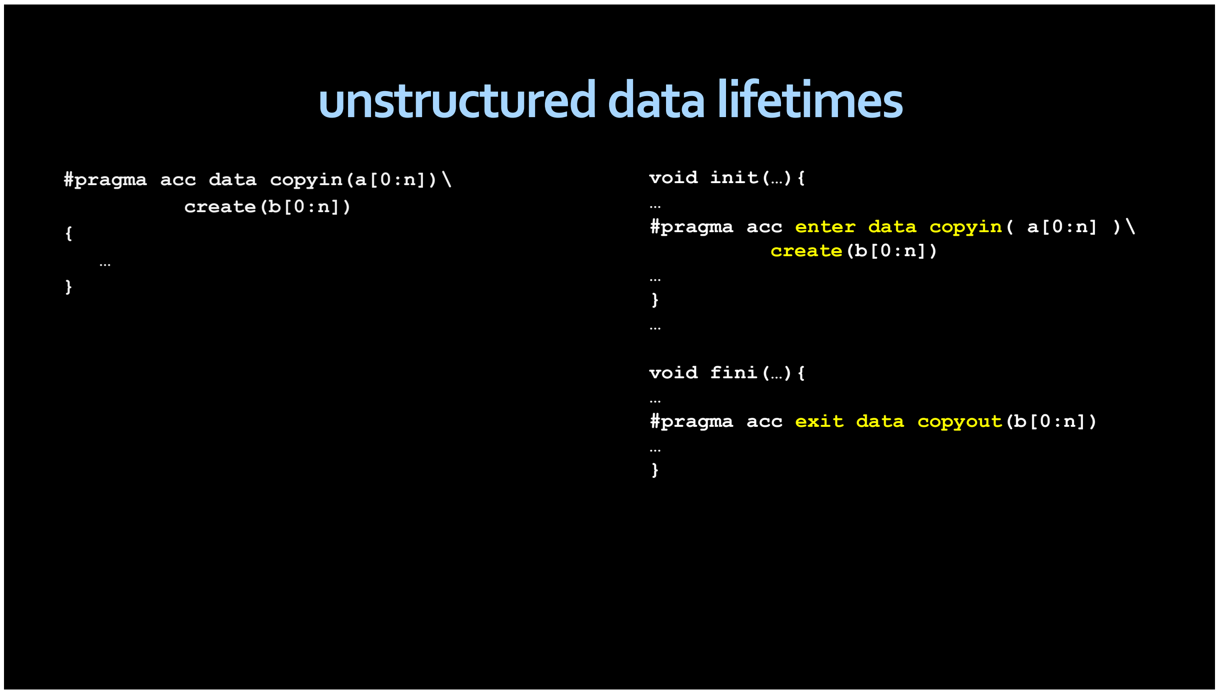


unstructured data lifetimes

```
#pragma acc data copyin(a[0:n])\
         create(b[0:n])
{
   …
}
```

```
#pragma acc enter data copyin( a[0:n] )\
         create(b[0:n])

…


#pragma acc exit data delete(a[0:n])
…
#pragma acc exit data copyout(b[0:n])
```

The simple use case for these directives is when the data lifetime actually begins inside some initialization routine. We want to create the device copy of the data in the initialization routine, and not defer it to some parent routine.



unstructured data lifetimes

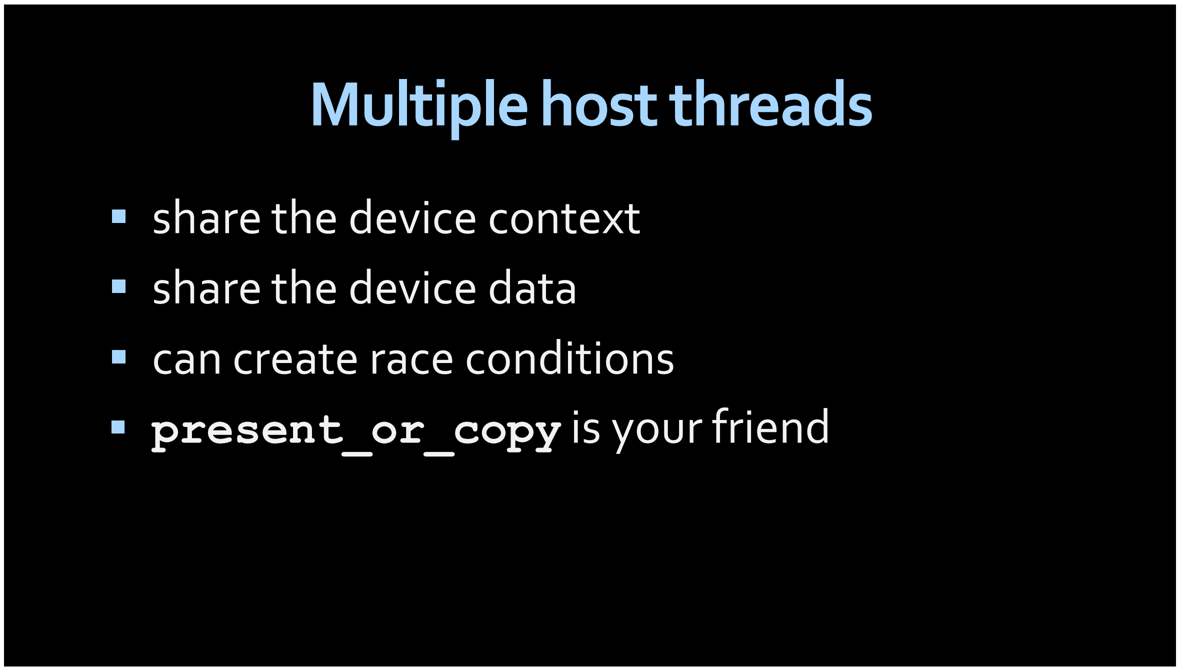
```
#pragma acc data copyin(a[0:n])\
         create(b[0:n])
{
   …
}
```

```
void init(…){
…
#pragma acc enter data copyin( a[0:n] )\
         create(b[0:n])
…
}
…

void fini(…){
…
#pragma acc exit data copyout(b[0:n])
…
}
```

If there is no ‘exit data’ directive, the data lifetime continues until the end of the program.

Multiple Host Threads

OpenACC 2.0 defines the behavior with multiple host threads. The multiple threads may share the same accelerator device, and if so, they share the device context and device data. The PGI 2013 compilers implement this already; this is a significant change from the PGI 2012 compilers, where each host thread had its own context, even when the threads shared a device.



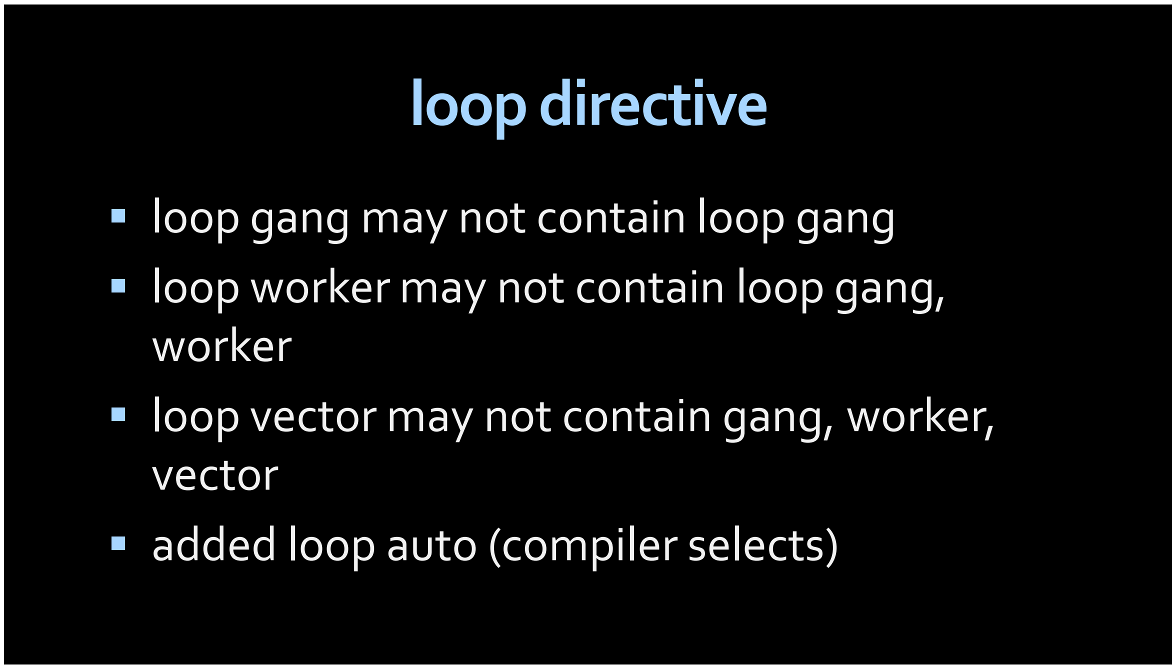
Multiple host threads

- share the device context
- share the device data
- can create race conditions
- **`present_or_copy`** is your friend

The situation to avoid is having multiple threads trying to copy the same data to the same device; the second thread will find that the data already exists on the device and will get a runtime error. Instead, have each thread use 'present_or_copy,' so the second thread will find the data already present and will then continue immediately.

Loop Directives

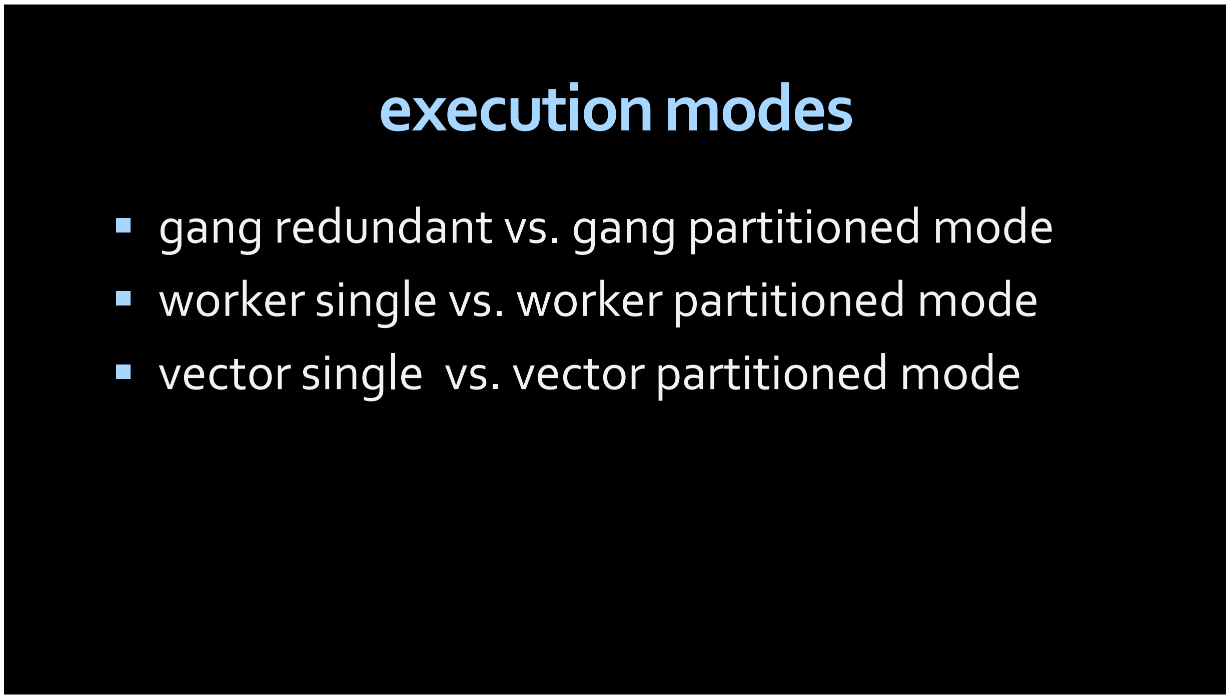
OpenACC is more strict about loop nestings. There are three levels of parallelism supported in OpenACC: gang, worker, vector. The loose mapping to CUDA terms are block, warp, thread, though this is not strict. In OpenACC 2.0, gang must be outermost, vector must be innermost. A new loop type, 'auto,' tells the compiler to determine for this target device the best mapping.



loop directive

- loop gang may not contain loop gang
- loop worker may not contain loop gang, worker
- loop vector may not contain gang, worker, vector
- added loop auto (compiler selects)

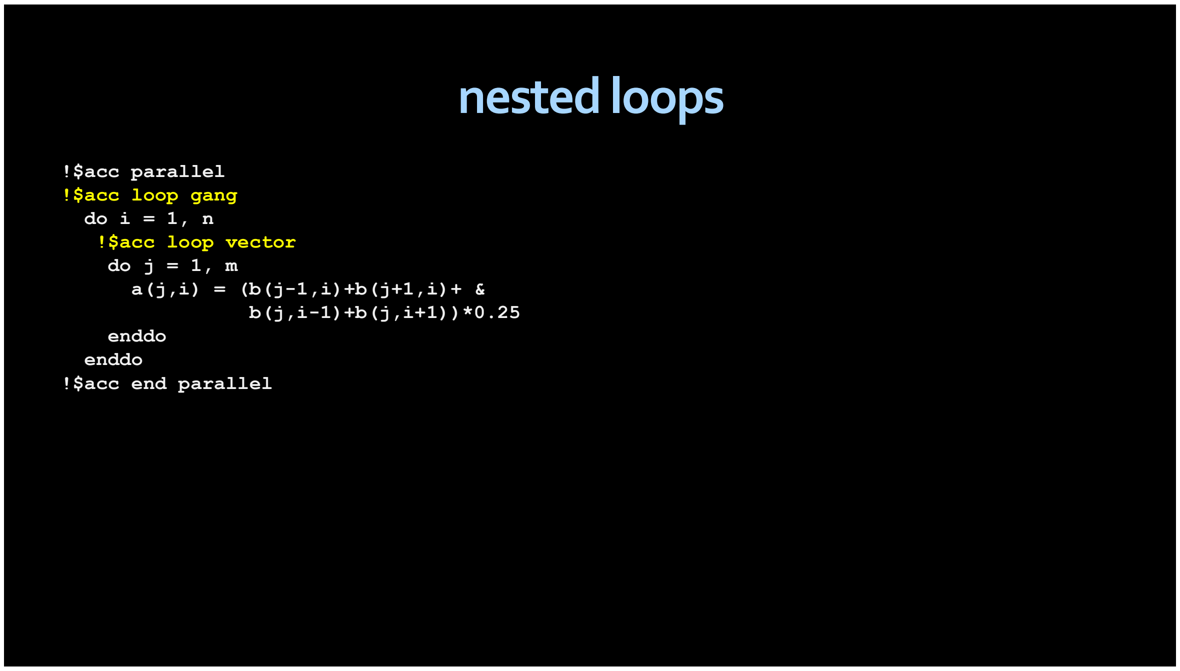
We've also added some terminology to discuss execution modes. When a parallel region starts, all the gangs are executing the same code redundantly (gang redundant mode). When it reaches a gang loop, the iterations are partitioned across the gangs (gang partitioned mode). When a parallel region starts, only a single worker is active (worker-single mode). When a worker loop is reached, the iterations are partitioned across the workers (worker-partitioned mode). Similarly, when a parallel region starts, only a single vector lane is active (vector-single mode). When a vector loop is reached, the iterations are partitioned across the vector lanes (vector-partitioned mode). Finally, we define an accelerator thread as a single vector lane of a single worker of a single gang.



execution modes

- gang redundant vs. gang partitioned mode
- worker single vs. worker partitioned mode
- vector single vs. vector partitioned mode

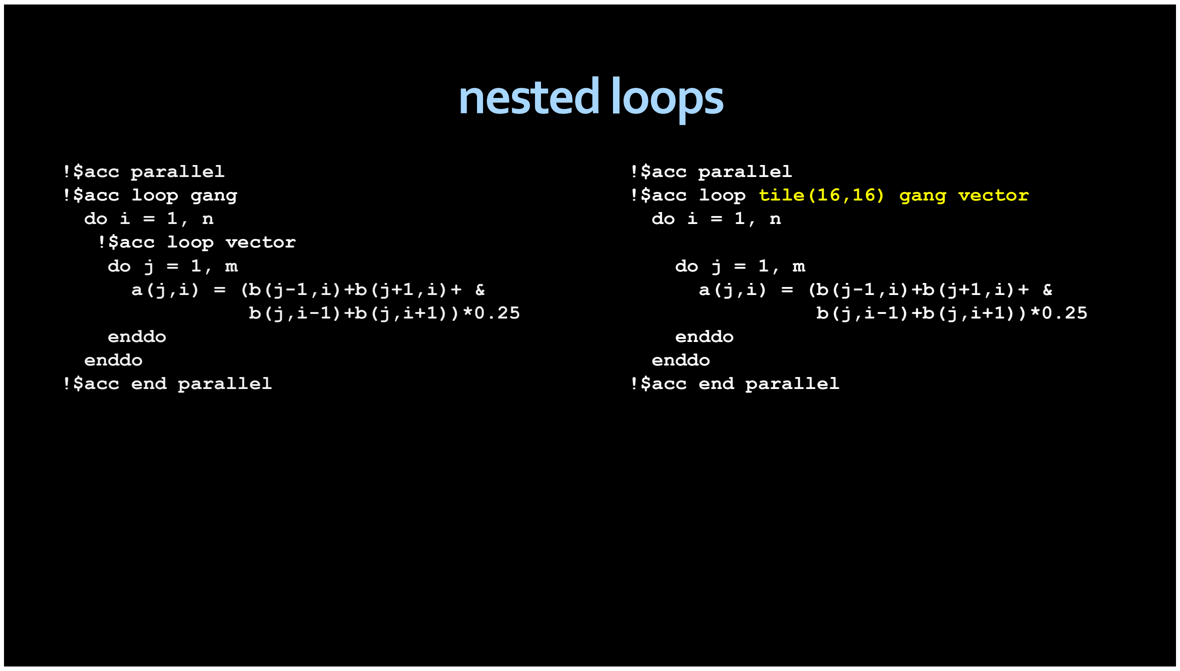
Currently, OpenACC allows nested loops with nested loop directives, but there’s no way to get true tiling.



nested loops

```
!$acc parallel
!$acc loop gang
  do i = 1, n
   !$acc loop vector
    do j = 1, m
      a(j,i) = (b(j-1,i)+b(j+1,i)+ &
                b(j,i-1)+b(j,i+1))*0.25
    enddo
  enddo
!$acc end parallel
```

OpenACC 2.0 adds a ‘tile’ clause, which lets you specify that you want, say, a 16x16 tile of iterations to execute across workers, with the tiles executing in parallel across gangs:

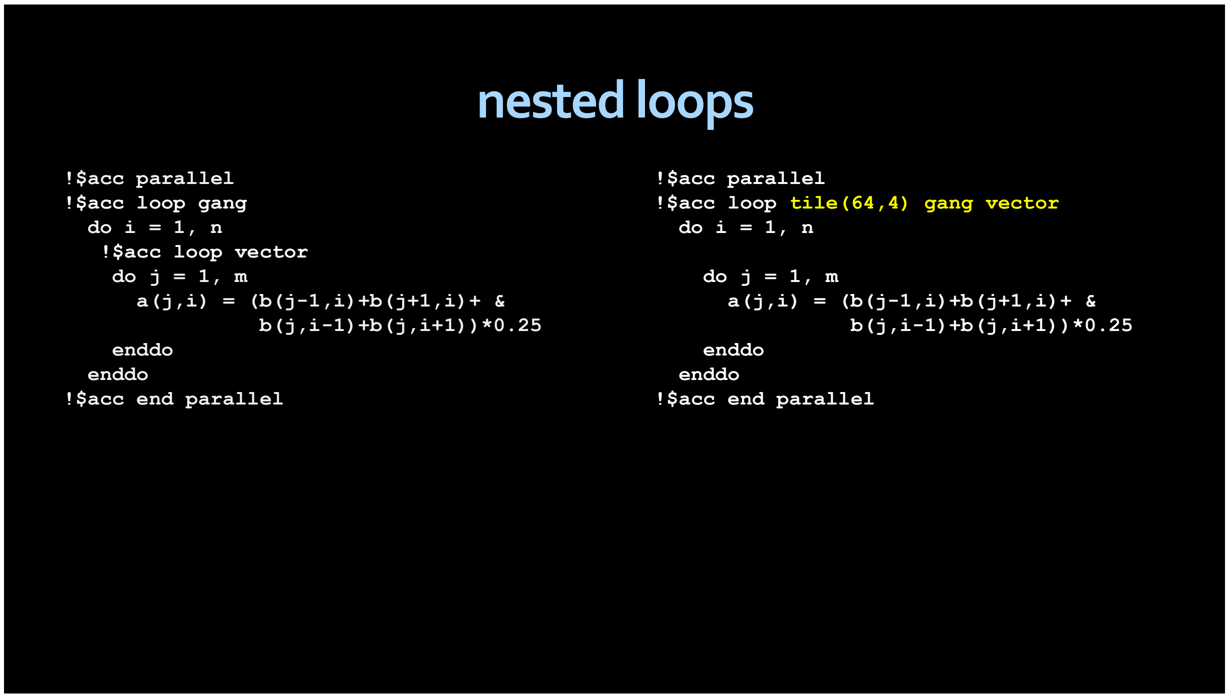


nested loops

```
!$acc parallel
!$acc loop gang
  do i = 1, n
   !$acc loop vector
    do j = 1, m
      a(j,i) = (b(j-1,i)+b(j+1,i)+ &
                b(j,i-1)+b(j,i+1))*0.25
    enddo
  enddo
!$acc end parallel
```

```
!$acc parallel
!$acc loop tile(16,16) gang vector
  do i = 1, n

    do j = 1, m
      a(j,i) = (b(j-1,i)+b(j+1,i)+ &
                b(j,i-1)+b(j,i+1))*0.25
    enddo
  enddo
!$acc end parallel
```

It’s important to note that the first element in the tile clause is the tile size of the inner loop, the second element is the tile size for the next outer loop, and so on. So the following example has a 64-long tile in the ‘j’ loop, and a 4-wide tile in the ‘i’ loop:


nested loops

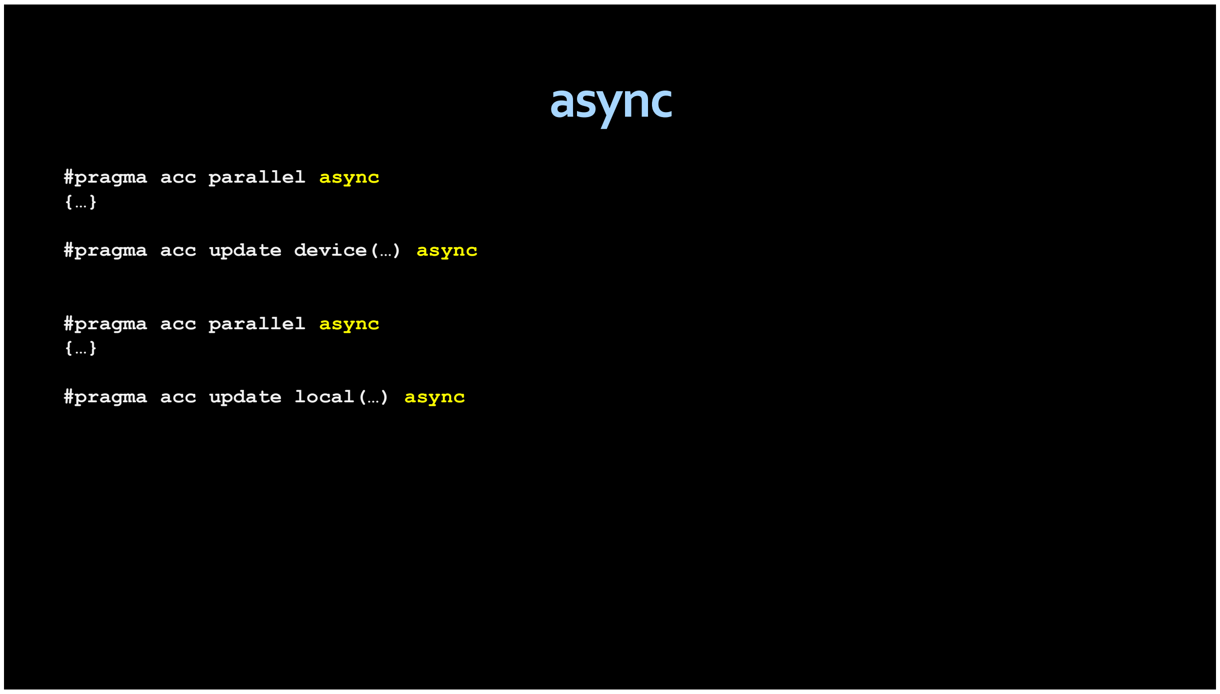
```
!$acc parallel
!$acc loop gang
  do i = 1, n
   !$acc loop vector
    do j = 1, m
      a(j,i) = (b(j-1,i)+b(j+1,i)+ &
               b(j,i-1)+b(j,i+1))*0.25
    enddo
  enddo
!$acc end parallel
```

```
!$acc parallel
!$acc loop tile(64,4) gang vector
  do i = 1, n

    do j = 1, m
      a(j,i) = (b(j-1,i)+b(j+1,i)+ &
               b(j,i-1)+b(j,i+1))*0.25
    enddo
  enddo
!$acc end parallel
```

There are rules about the loop modes allowed and how they interact, but the common cases are easily described.

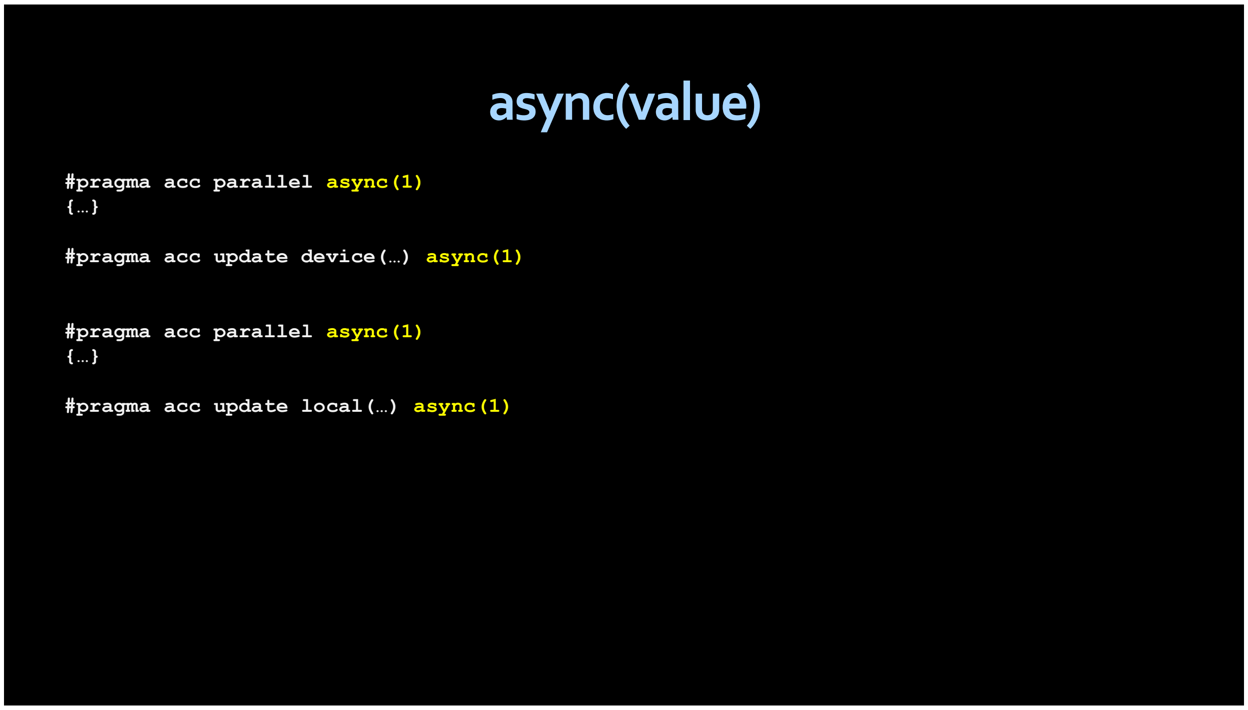
Asynchronous behavior

OpenACC allows for asynchronous compute regions and data updates with the async clause:


async

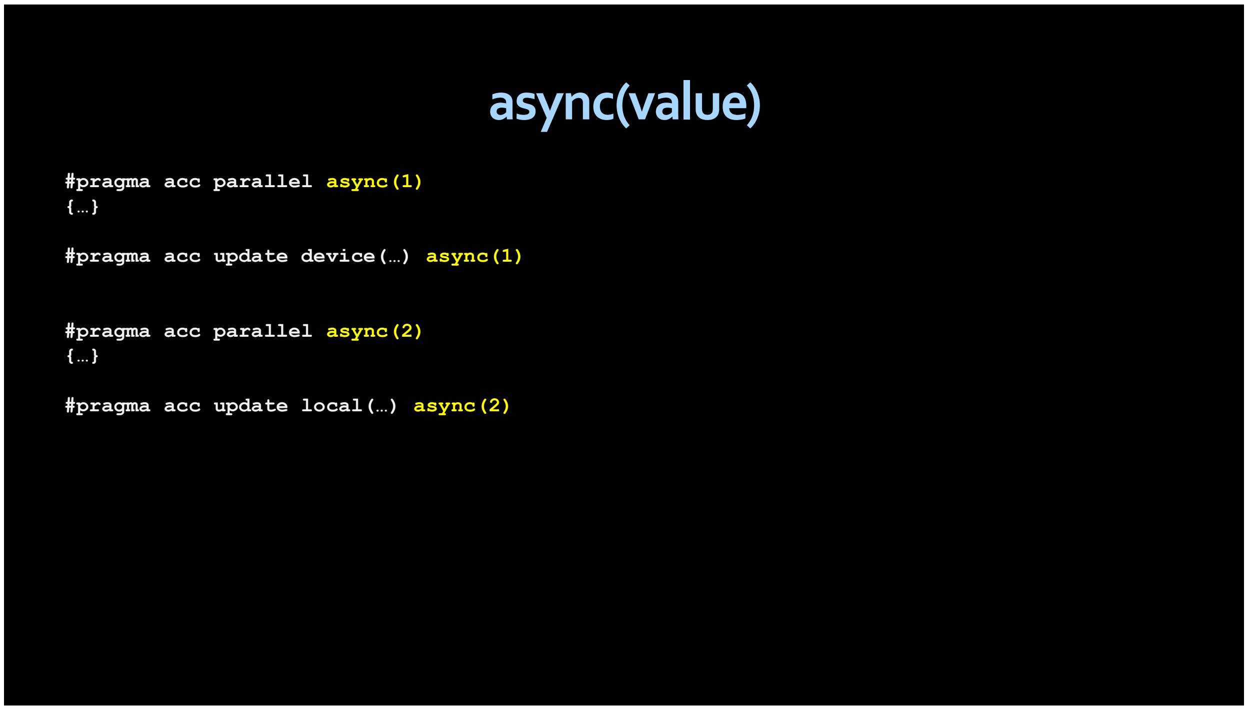
```
#pragma acc parallel async
{…}

#pragma acc update device(…) async


#pragma acc parallel async
{…}

#pragma acc update local(…) async
```

Your OpenACC program can have a number of async queues by adding a value expression to the async clause:


async(value)

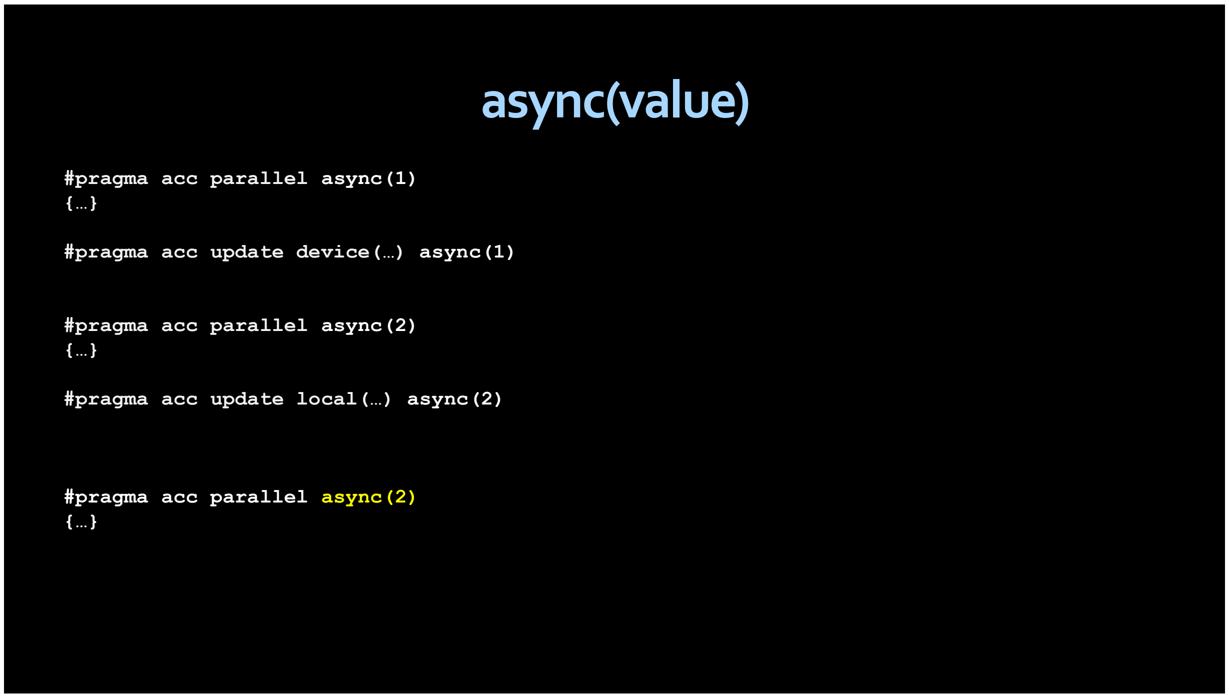
```
#pragma acc parallel async(1)
{…}

#pragma acc update device(…) async(1)


#pragma acc parallel async(1)
{…}

#pragma acc update local(…) async(1)
```

Asynchronous activities with the same async value will be executed that they are enqueued by the host thread. Activities with different async value can execute in any order. Async queues correspond roughly to CUDA streams.


async(value)

```
#pragma acc parallel async(1)
{…}

#pragma acc update device(…) async(1)


#pragma acc parallel async(2)
{…}

#pragma acc update local(…) async(2)
```

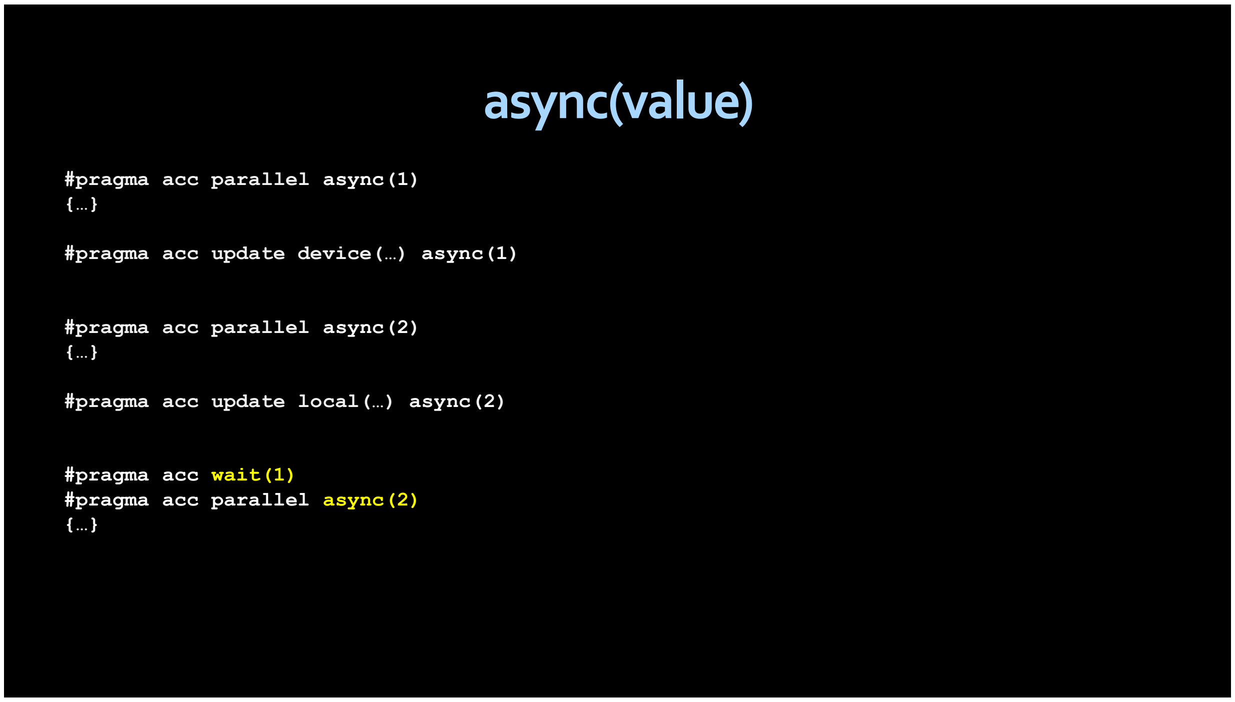
What do you do if you have two async queues, then you have one operation that depends on activities from both queues, as the last parallel construct below:



async(value)

```
#pragma acc parallel async(1)
{…}

#pragma acc update device(…) async(1)


#pragma acc parallel async(2)
{…}

#pragma acc update local(…) async(2)



#pragma acc parallel async(2)
{…}
```

One option, shown below, is to add a wait directive to wait for queue 1 to finish, then put the operation on queue 2 (or, equivalently, wait for queue 2 then put the operation on queue 1).

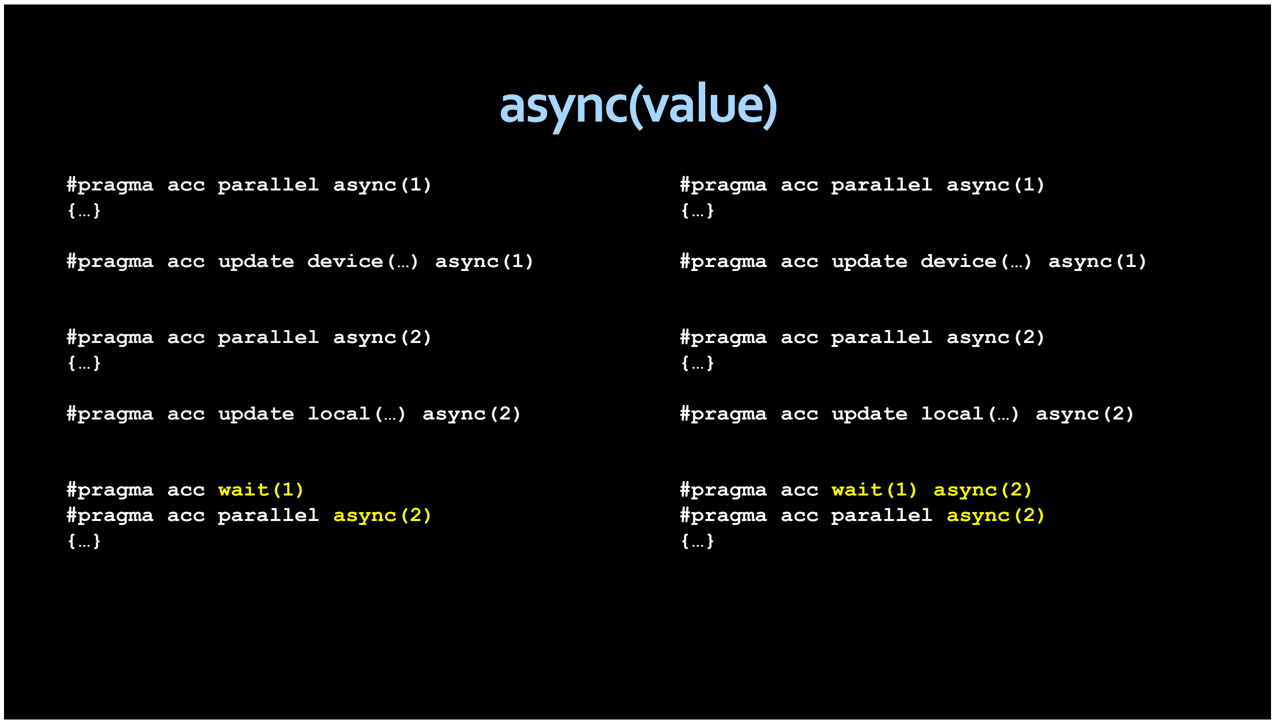


async(value)

```
#pragma acc parallel async(1)
{…}

#pragma acc update device(…) async(1)


#pragma acc parallel async(2)
{…}

#pragma acc update local(…) async(2)


#pragma acc wait(1)
#pragma acc parallel async(2)
{…}
```

This has the unfortunate effect of making the host thread wait for the device queue to flush, losing the decoupling between host and device.

OpenACC 2.0 allows for adding a wait operation on another device queue:



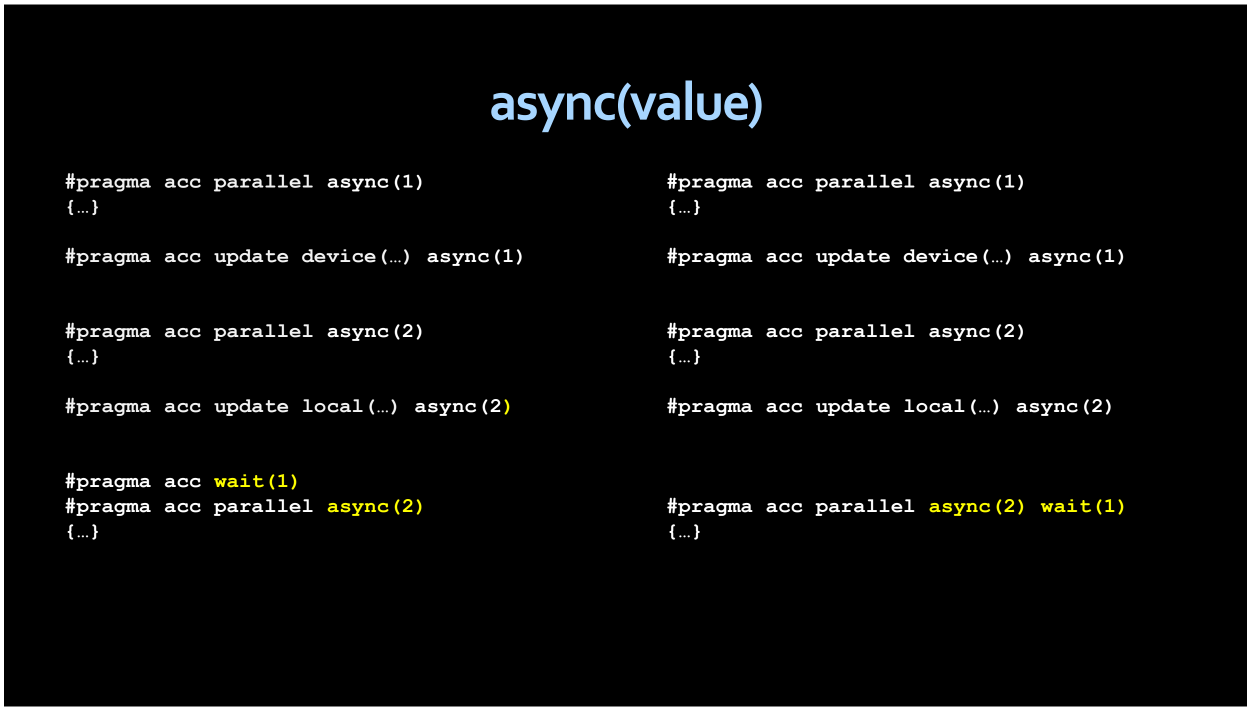
async(value)

```
#pragma acc parallel async(1)
{…}

#pragma acc update device(…) async(1)


#pragma acc parallel async(2)
{…}

#pragma acc update local(…) async(2)


#pragma acc wait(1)
#pragma acc parallel async(2)
{…}
```

```
#pragma acc parallel async(1)
{…}

#pragma acc update device(…) async(1)


#pragma acc parallel async(2)
{…}

#pragma acc update local(…) async(2)


#pragma acc wait(1) async(2)
#pragma acc parallel async(2)
{…}
```

This corresponds roughly to adding a CUDA event on the stream for queue 1 and a wait for that event on the stream for queue 2. The host thread can then continue executing without waiting for either queue to flush. Of course, a fallback implementation may still have the host wait for queue 1, but if the device supports multiple queues with synchronization, it's more efficient to put all the synchronization on the device itself.

A simpler way to specify the same thing is to put a wait clause on the async compute construct itself; in this case, putting the wait on the parallel construct makes the parallel operation wait until all the events in queue 1 are complete before issuing the parallel region.

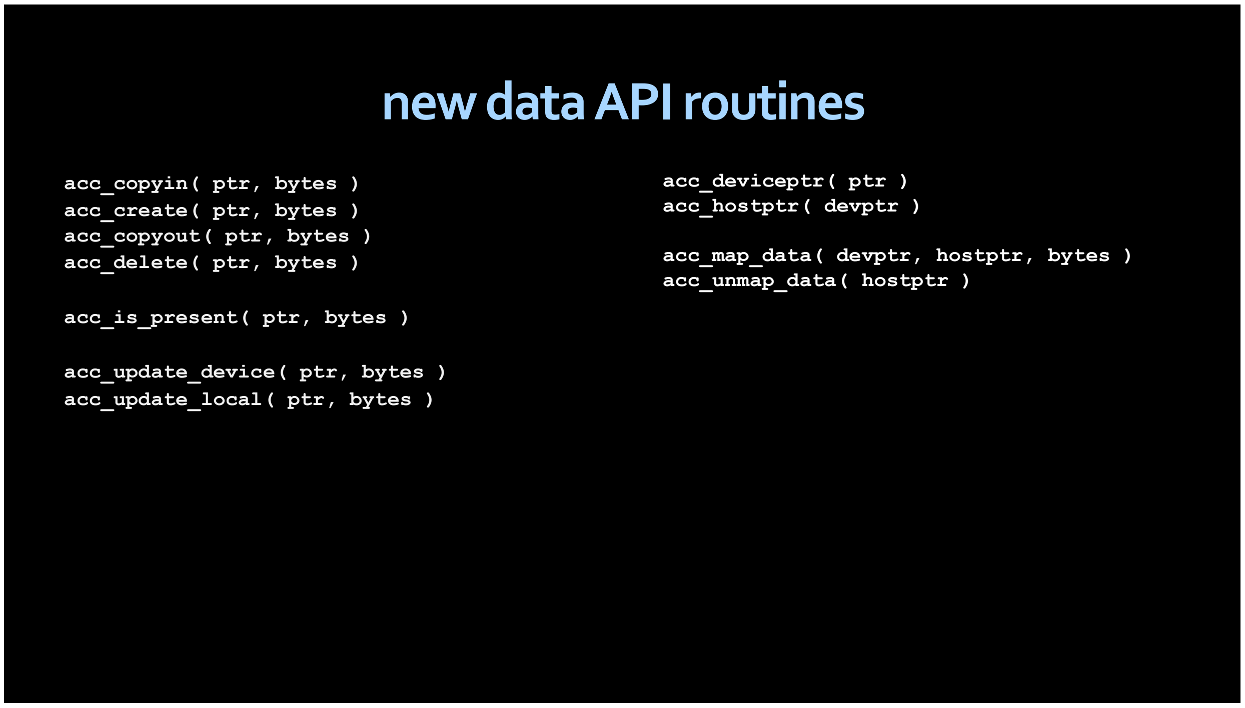


async(value)

```
#pragma acc parallel async(1)                #pragma acc parallel async(1)
{…}                                          {…}

#pragma acc update device(…) async(1)        #pragma acc update device(…) async(1)


#pragma acc parallel async(2)                #pragma acc parallel async(2)
{…}                                          {…}

#pragma acc update local(…) async(2)         #pragma acc update local(…) async(2)


#pragma acc wait(1)
#pragma acc parallel async(2)                #pragma acc parallel async(2) wait(1)
{…}                                          {…}
```

New API Routines

OpenACC 2.0 adds a number of new API routines. I mention only a few here that have to do with explicit data management. Some of these were present in previous PGI releases, and some were introduced by the other OpenACC vendors.



new data API routines

```
acc_copyin( ptr, bytes )                     acc_deviceptr( ptr )
acc_create( ptr, bytes )                     acc_hostptr( devptr )
acc_copyout( ptr, bytes )
acc_delete( ptr, bytes )                     acc_map_data( devptr, hostptr, bytes )
                                             acc_unmap_data( hostptr )

acc_is_present( ptr, bytes )

acc_update_device( ptr, bytes )
acc_update_local( ptr, bytes )
```

acc_copyin, acc_create, acc_copyout and acc_delete act very like the relevant data clauses, more or less just like a 'enter data' or 'exit data' directive.

acc_is_present tests whether all the given data is already present on the device.

acc_update_device and acc_update_local act very like the corresponding 'update' directive.

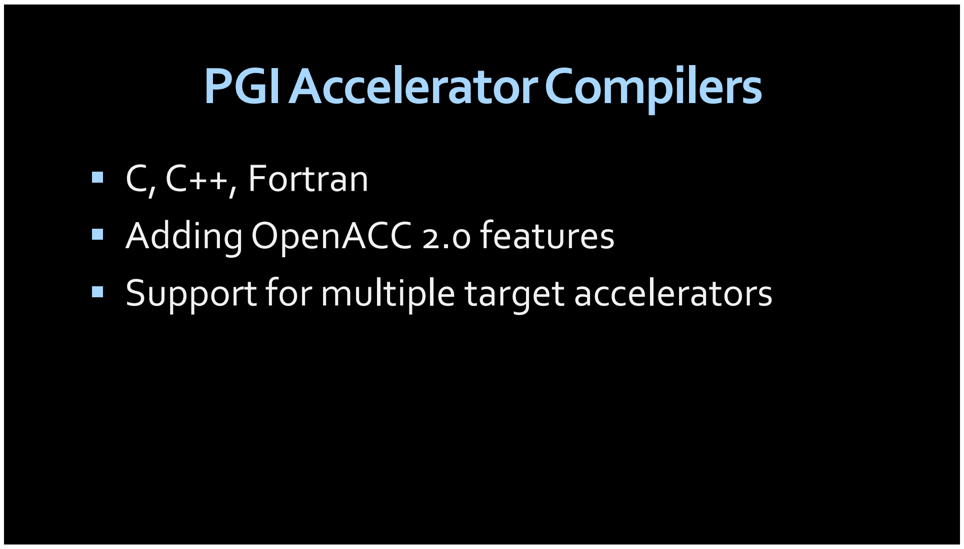
acc_deviceptr returns the device pointer corresponding to the given host address; this could be used, for instance, to pass to an explicit CUDA kernel.

acc_hostptr does the inverse, returns the host address corresponding to a device address.

acc_map_data takes some device address that was directly allocated and adds the entry to the OpenACC runtime 'present' table that this device address corresponds to the given host address. acc_unmap_data undoes this effect.

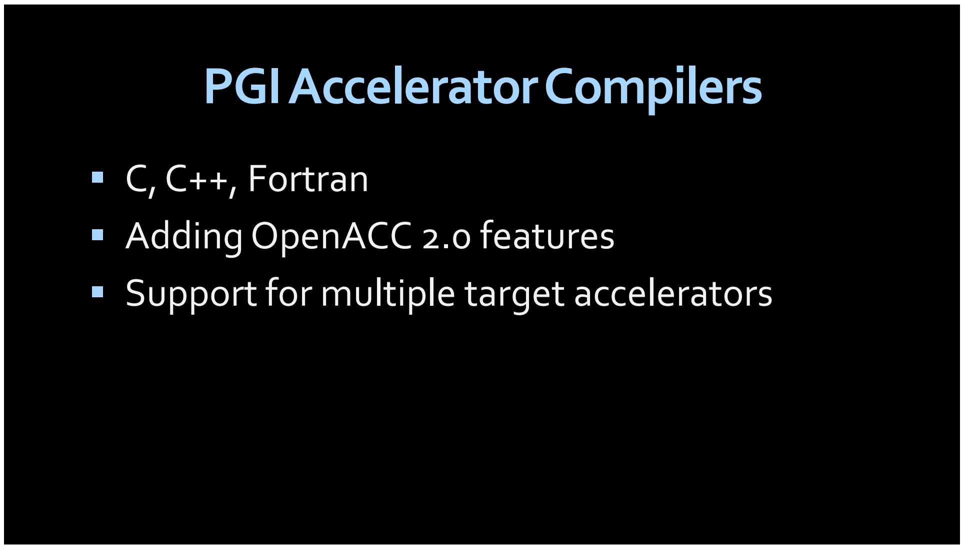
PGI Compilers

I close with a short discussion of the PGI compilers and tools. PGI is working on adding the OpenACC 2.0 features, starting with some of the simpler data features and separate compilation. The PGI compilers already support some of the new features such as multiple accelerators in a single program. The PGI OpenACC Fortran compiler is already well-integrated with CUDA Fortran, allowing Fortran programmers to mix high-level OpenACC with detailed and manually-optimized CUDA kernels.



PGI Accelerator Compilers

- C, C++, Fortran
- Adding OpenACC 2.0 features
- Support for multiple target accelerators



PGI Accelerator Compilers

- C, C++, Fortran
- Adding OpenACC 2.0 features
- Support for multiple target accelerators