EPICURE

OpenACC programming
on NVIDIA GPUs

Epicure hackathon @ CINECA, Casalecchio di Reno (BO) Italy
28-31 October 2024

CINECA

Co-funded by the European Union

EuroHPC Joint Undertaking

This project has received funding from the High Performance Computing Joint Undertaking under grant agreement No 101139786

Why OpenACC?

Incremental approach

The runtime can handle data movements, compiler optimizes the offload for the underlying hardware

Avoid source code duplication

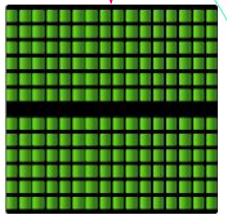
Based on compiler directives,

acc activated via compilation flags

More portable

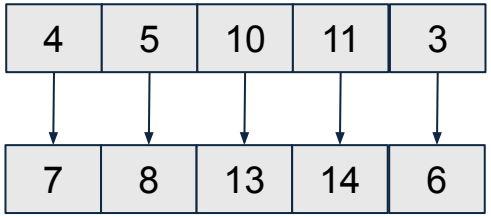
Supported also by AMD GPUs

```
< 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 >
}

< sequential code >
```



Coding with OpenACC

```
< cpu code >
#pragma acc parallel loop
for( i = 0; i < N; i++ )
     {
     output[i] = input[i] + scalar
}
< cpu code >
```



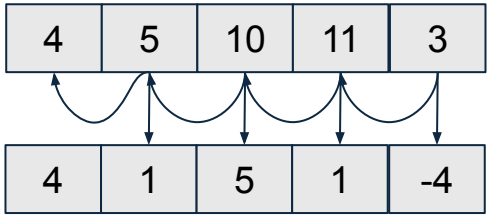
There must be no **data dependencies inside the loop** (each element computation at i is independent)

The compiler will compile the loop for the GPU

Each GPU thread will execute the operation on a subset of the iteration range

Data dependencies

```
< cpu code >
#pragma acc parallel loop
for( i = 1; i < N; i++ )
     {
     output[i] = output[i-1] + scalar
}
< cpu code >
```



Not all loops are parallel!

If there dependencies between elements in the array, the GPU thread might access an element after it has changed

This provides **wrong results**

OpenACC syntax

C/C++

```
#pragma acc directive clauses
<code>
```

Fortran

```
!$acc directive clauses
<code>
```

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.

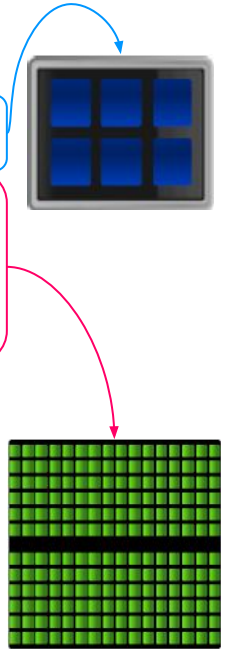
"***acc***" informs the compiler that what will come is an OpenACC directive

Directives are commands in OpenACC for altering our code.

Clauses are specifiers or additions to directives.

The parallel directive

```
< sequential code >

#pragma acc parallel
{
            <code for gpu>
}
```



`parallel` instructs the compiler to **create parallel *gangs***

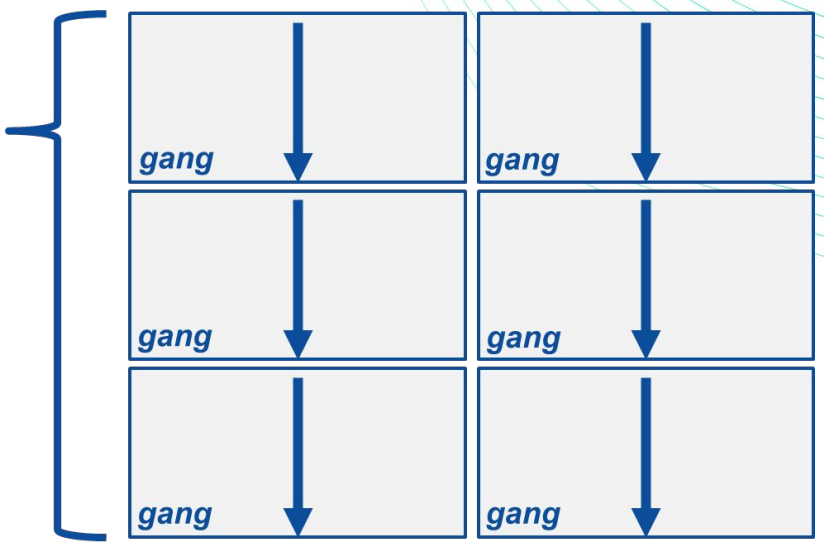
Gangs are **independent groups of worker threads** on the accelerator

The code contained within a parallel directive is executed redundantly by all parallel gangs

The parallel directive

```
!$acc parallel

    < code for gpu >

!$acc end parallel
```

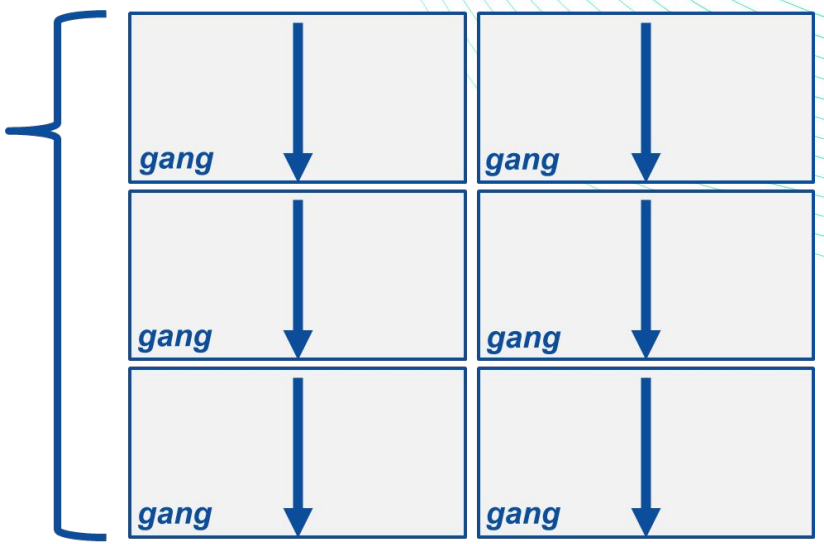
The compiler will generate *1 or more parallel* ***gangs***, which execute redundantly.



The loop directive

```
!$acc parallel

do i = 1, N
   < some operation >
end do

!$acc end parallel
```

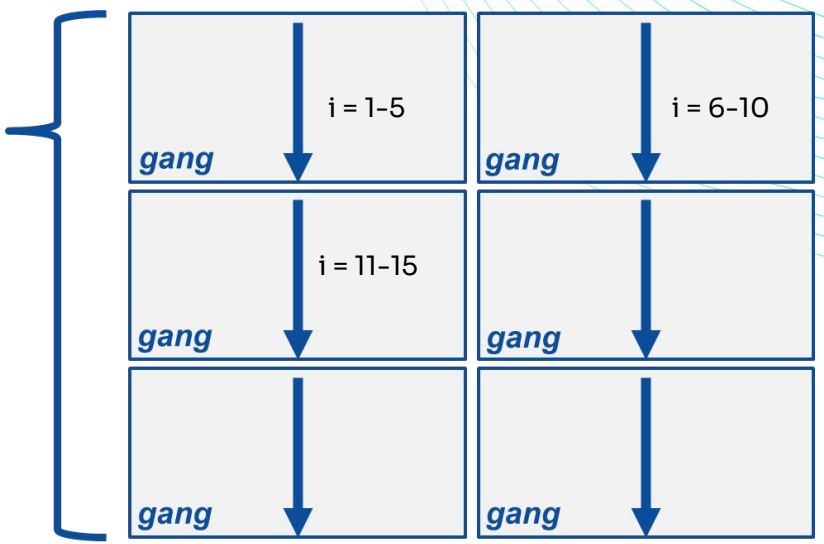
this way, the same loop is executed by each gang. We want to distribute loop iterations among gangs!



The loop directive

```
!$acc parallel
!$acc loop
do i = 1, N
   < some operation >
end do
!$acc end loop
!$acc end parallel
```

the loop directive is used to distribute loop iterations among gangs



The parallel loop directive

C/C++

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

Fortran

```
!$acc parallel
        !$acc loop
        do i = 1, N
                a(i) = 0
        end do
!$acc end parallel
```

parallel marks a region of code where you parallel execution should occur

The **loop** directive is used to instruct the compiler to parallelize the iterations of the next loop to run across the parallel gangs

The parallel loop directive

C/C++

```
#pragma acc parallel loop
for(int i = 0; j < N; i++)
        a[i] = 0;
```

Fortran

```
!$acc parallel loop
do i = 1, N
        a(i) = 0
end do
```

parallel marks a region of code where you parallel execution should occur

The **loop** directive is used to instruct the compiler to parallelize the iterations of the next loop to run across the parallel gangs

parallel loop can also be fused in a single direcive

The kernels directive

```
#pragma acc kernels
{
  for(int i = 0; i < N; i++)
  a[i] = 0;

  for(int j = 0; j < M; j++)
  b[i] = 0;
}
```

```
!$acc kernels
a(:) = 1
b(:) = 2
c(:) = a(:) + b(:)
!$acc end kernels
```

The **kernels** directive instructs the compiler to search for parallel loops in the code

The compiler will analyze the loops and **parallelize those it finds safe** and profitable to do so

The kernels directive can be applied to regions containing multiple loop nests

Supports Fortran array syntax

Loop nests

C/C++

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

Fortran

```
!$acc parallel loop
do i = 1, N
          !$acc loop
          do i = 1, M
                    a(i,j) = 0
          end do
end do
```

loop directives can be nested to parallelize multi-dimensional loops

This **allows the compiler to implement more levels of parallelism**, and increase performance, **if resources are available**

If more levels are not available, the inner loop directives will be ignored

Reductions

The inner-most loop is not parallelizable, multiple threads could attempt to write to tmp → we should expect to receive erroneous results

```
do k = 1, size
  do j = 1, size
    tmp = 0.0
    !$acc parallel loop
    do i = 1, size
      tmp = tmp + a(i,k) * b(k,j)
    end do
    c(i,j) = tmp
  end do
end do
```

Reductions

The inner-most loop is not parallelizable, multiple threads could attempt to write to tmp → we should expect to receive erroneous results

To fix this, we should use the reduction clause

```
do k = 1, size
  do j = 1, size
    tmp = 0.0
    !$acc parallel loop reduction(+:tmp)
    do i = 1, size
      tmp = tmp + a(i,k) * b(k,j)
    end do
    c(i,j) = tmp
  end do
end do
```

Reductions

The inner-most loop is not parallelizable, multiple threads could attempt to write to tmp → we should expect to receive erroneous results

To fix this, we should use the reduction clause

```
do k = 1, size
  do j = 1, size
    tmp = 0.0
    !$acc parallel loop reduction(+:tmp)
    do i = 1, size
      tmp = tmp + a(i,k) * b(k,j)
    end do
    c(i,j) = tmp
  end do
end do
```

Each thread group will its own private copy of the reduction variable and perform a **partial reduction** on their loop iterations

Reductions

The inner-most loop is not parallelizable, multiple threads could attempt to write to tmp → we should expect to receive erroneous results

To fix this, we should use the reduction clause

```
do k = 1, size
  do j = 1, size
    tmp = 0.0
    !$acc parallel loop reduction(+:tmp)
    do i = 1, size
      tmp = tmp + a(i,k) * b(k,j)
    end do
    c(i,j) = tmp
  end do
end do
```

Each thread group will its own private copy of the reduction variable and perform a **partial reduction** on their loop iterations

After the loop, a final reduction will be performed to produce a **single global result**

Reduction operators

Operator	Description	Example
+	Addition/Summation	reduction(+:sum)
*	Multiplication/Product	reduction(*:product)
max	Maximum value	reduction(max:maximum)
min	Minimum value	reduction(min:minimum)
&	Bitwise and	reduction(&:val)
|	Bitwise or	reduction(|:val)
&&	Logical and	reduction(&&:val)
||	Logical or	reduction(||:val)

Reductions

```
a[0] = 0;
#pragma parallel acc loop \
        reduction(+:a[0])
for( i = 0; i < 100; i++ )
        a[0] += i;
```

The reduction variable may not be an array element

```
v.val = 0;
#pragma acc parallel loop \
        reduction(+:v.val)
for( i = 0; i < v.n; i++ )
        v.val += i;
```

The reduction variable may not be a C struct member, a C++ class or struct member, or a Fortran derived type member

Sequential

```
#pragma acc parallel loop
for( i = 0; i < size; i++ )
   #pragma acc loop
   for( j = 0; j < size; j++ )
      #pragma acc loop seq
      for( k = 0; k < size; k++ )
         c[i][j] += a[i][k] * b[k][j];
```

The **seq clause will tell the compiler to run the loop sequentially**

The compiler will parallelize the outer loops across the parallel threads, but each thread will run the inner-most loop sequentially

The compiler may automatically apply the seq clause to loops as well

Privatizations

```
double tmp[3];
#pragma acc kernels loop private(tmp[0:3])
for( i = 0; i < size; i++ )
{
  tmp[0] = <value>;
  tmp[1] = <value>;
  tmp[2] = <value>;
}
// note that the host value of “tmp”
// remains unchanged.
```

Each thread can have a private copy of every variable

- **private** variables are uninitialized.
- **firstprivate** private values are initialized to the same value used on the host.

Unless doing a reduction, the value on the host outside the parallel region is unchanged

Privatizations

```
double tmp[3];
#pragma acc kernels loop private(tmp[0:3])
for( i = 0; i < size; i++ ) {
  // the tmp array is private to each
iteration of the outer loop
  tmp[0] = <value>;
  tmp[1] = <value>;
  tmp[2] = <value>;
  #pragma acc loop
  for ( j = 0; j < size2; j++) {
      // but tmp is shared amongst the threads
    // in the inner loop
      array[i][j] = tmp[0]+tmp[1]+tmp[2];
  }
}
```

Variables in **private** or **firstprivate** clause are private to the loop level on which the clause appears.

Private variables on an outer loop are shared within inner loops

Scalars

By default, scalars are **firstprivate** when used in a parallel region and **private** when used in a kernels region.

Except in some cases, scalars do not need to be added to a private clause. These cases may include but are not limited to:

1. Scalars with global storage such as global variables in C/C++, Module variables in Fortran
2. When the scalar is passed by reference to a device subroutine
3. When the scalar is used as an rvalue after the compute region, aka “live-out”

<u>Note that putting scalars in a private clause may actually hurt performance!</u>

Collapse clause

collapse(N) combines the next N tightly nested loops

Can turn a multidimensional loop nest into a single-dimension loop

This can be extremely useful for increasing **memory locality**, as well as creating larger loops to **expose more parallelism**

```
#pragma acc parallel loop collapse(2)
for( i = 0; i < size; i++ ){
  for( j = 0; j < size; j++ ){
    double tmp = 0.0f;
    #pragma acc loop reduction(+:tmp)
    for( k = 0; k < size; k++ ){
      tmp += a[i][k] * b[k][j];
    }
  }
  c[i][j] = tmp;
}
```

Compiling for GPUs

```
nvc -fast -acc -gpu=cc80,cuda12.3 -Minfo=accel main.c
nvc++ -fast -acc -gpu=cc80,cuda12.3 -Minfo=accel main.cpp
nvfortran -fast -acc -gpu=cc80,cuda12.3 -Minfo=accel main.f90
```

```
daxpy:
     19, Generating NVIDIA GPU code
         20, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
     19, Generating implicit
copyout(y(1:2147483647),x(1:2147483647),d(1:2147483647)) [if not already
present]
     26, Generating NVIDIA GPU code
         27, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
     26, Generating implicit copyin(y(1:2147483647),x(1:2147483647)) [if not
already present]
         Generating implicit copyout(d(1:2147483647)) [if not already present]
```

Compiling for multicore

```
nvc -fast -acc -ta=multicore -Minfo=accel main.c
nvc++ -fast -acc -ta=multicore -Minfo=accel main.cpp
nvfortran -fast -acc -ta=multicore -Minfo=accel main.f90
```

```
daxpy:
     19, Generating Multicore code
         20, !$acc loop gang
     26, Generating Multicore code
         27, !$acc loop gang
```

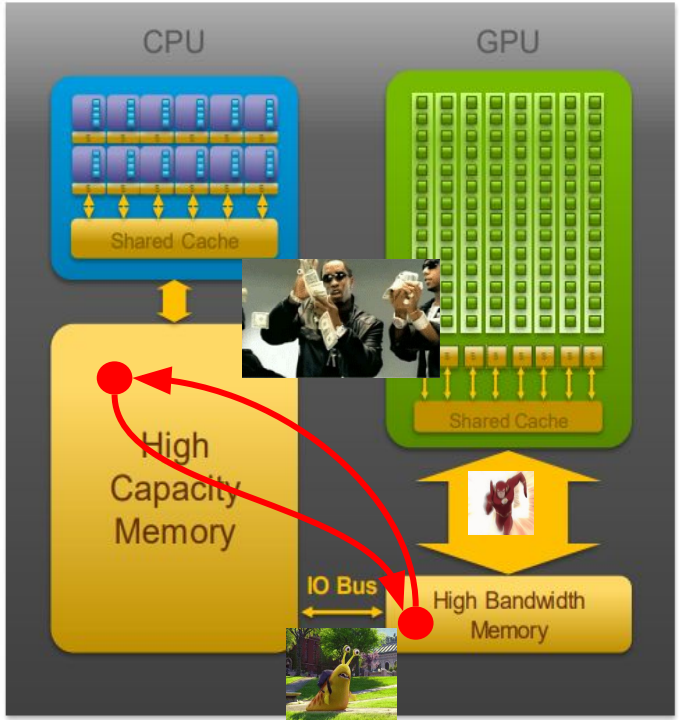
Data management

Data must be visible on the **device** when running **parallel** code

Data must be visible on the **host** when running **sequential** code

When the host and device don't share memory, data movement must occur

To maximize performance, the programmer should avoid all unnecessary data transfers



Implicit data management

```
do i = 1, N
    D(i) = A* X(i) + Y(i)
 end do
```

Implicit data management

```
!$acc parallel do
do i = 1, N
    D(i) = A* X(i) + Y(i)
 end do
!$acc end parallel do
```

Implicit data management

```
!$acc parallel do

do i = 1, N

    D(i) = A* X(i) + Y(i)

 end do

!$acc end parallel do
```

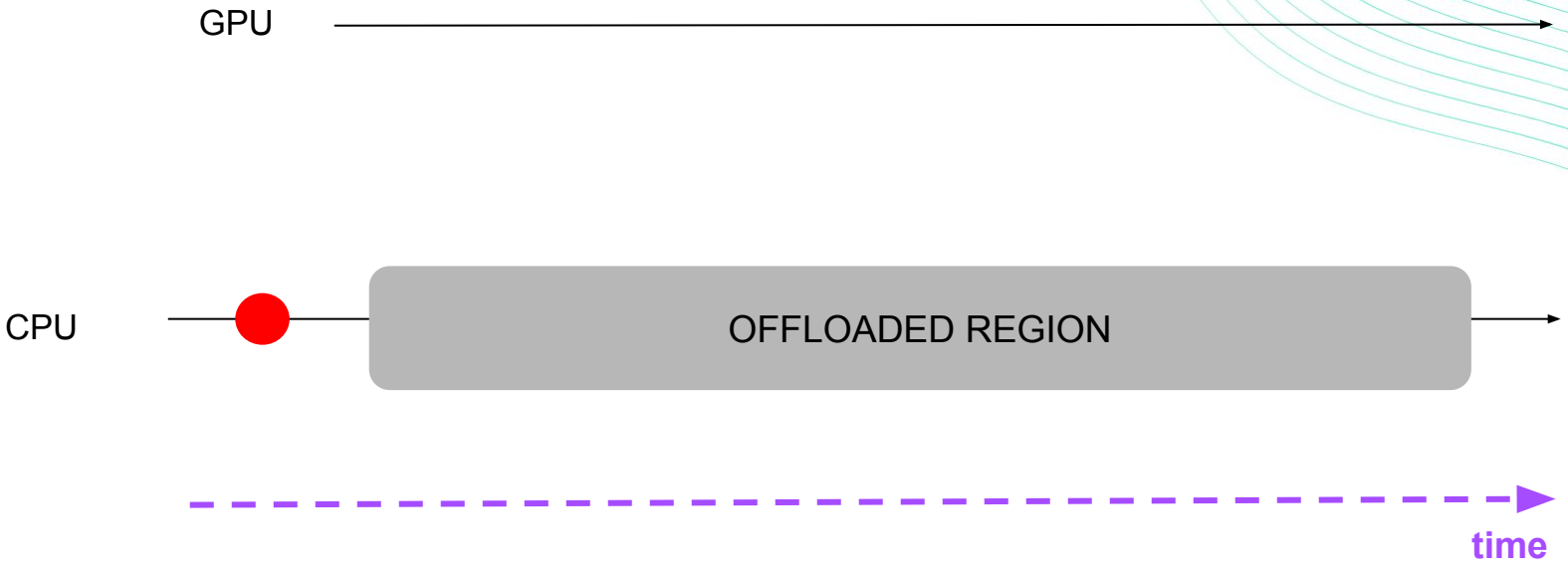
```
pgfortran -fast -acc -gpu=cc80,cuda12.3 -Minfo=accel -o
binary daxpy.f90
daxpy:
```

Implicit data management

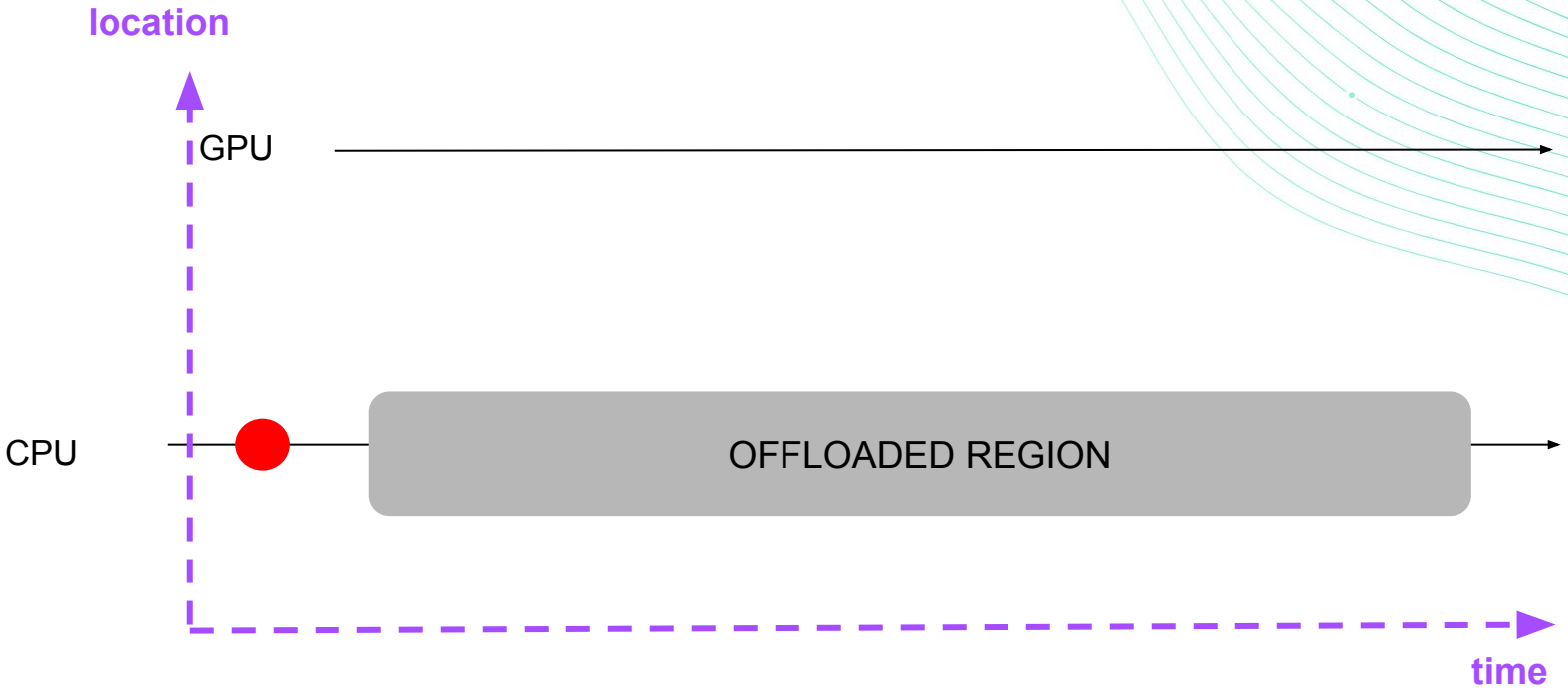
```
!$acc parallel do

do i = 1, N

    D(i) = A* X(i) + Y(i)

 end do

!$acc end parallel do
```

```
pgfortran -fast -acc -gpu=cc80,cuda12.3 -Minfo=accel -o
binary daxpy.f90
daxpy:
     19, Generating NVIDIA GPU code
         20, !$acc loop gang, vector(128) ! blockidx%x
threadidx%x
     19, Generating implicit
copyout(y(1:2147483647),x(1:2147483647),d(1:2147483647)) [if
not already present]
     26, Generating NVIDIA GPU code
         27, !$acc loop gang, vector(128) ! blockidx%x
threadidx%x
     26, Generating implicit
copyin(y(1:2147483647),x(1:2147483647)) [if not already
present]
         Generating implicit copyout(d(1:2147483647)) [if not
already present]
```

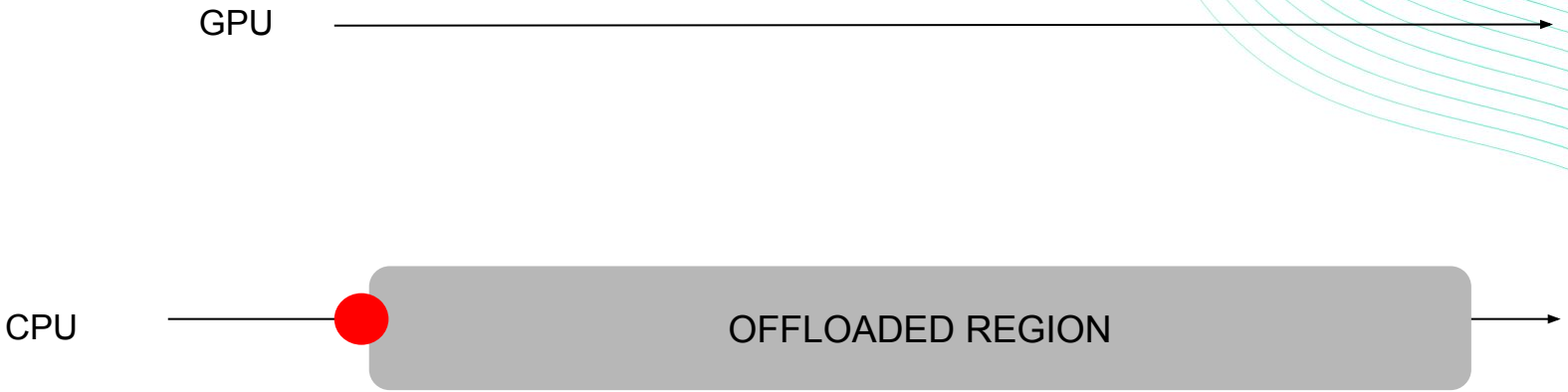
Traces in heterogeneous programs



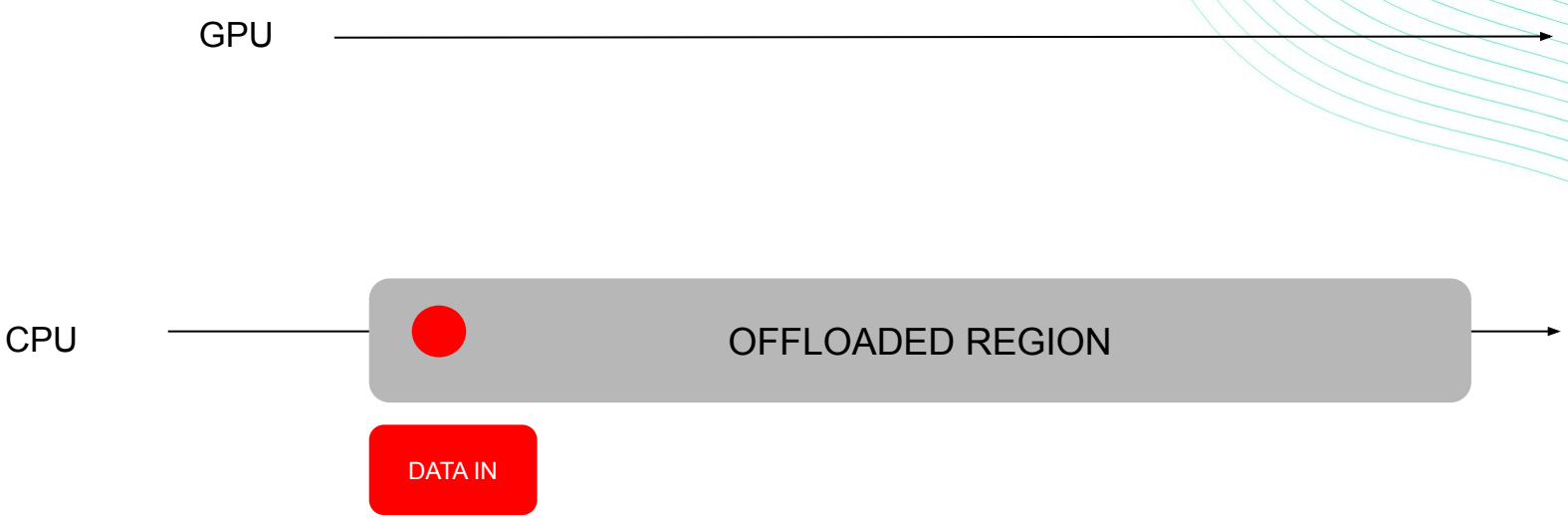
Traces in heterogeneous programs



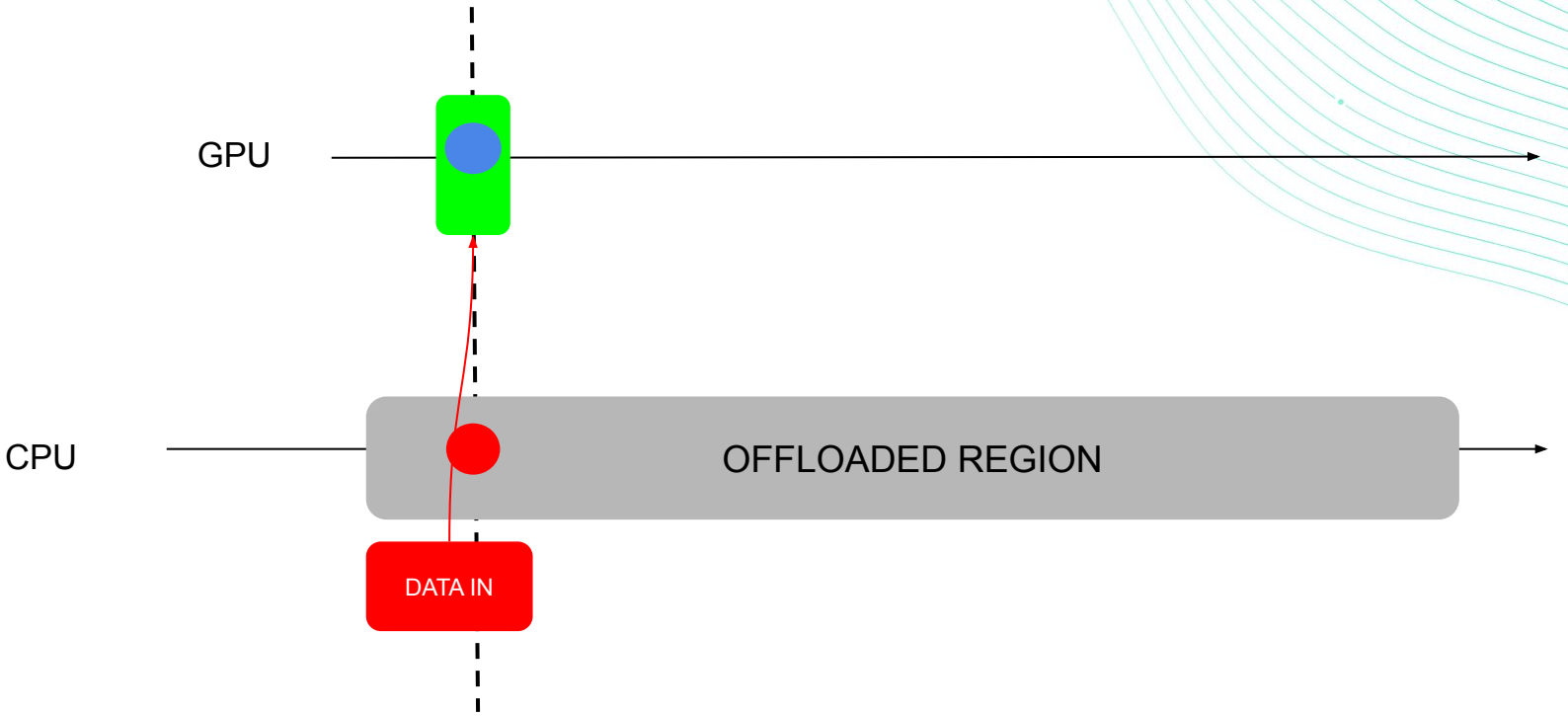
Traces in heterogeneous programs



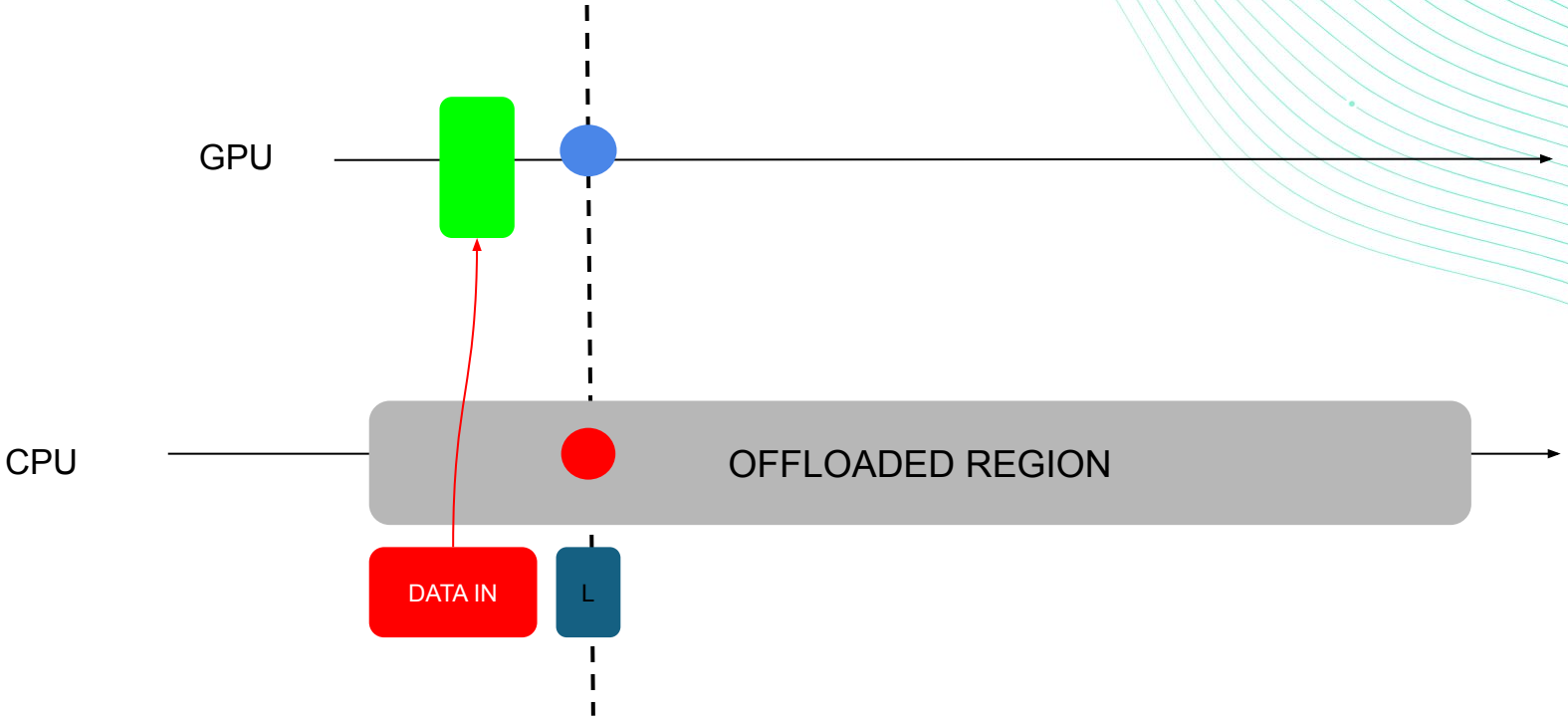
Traces in heterogeneous programs



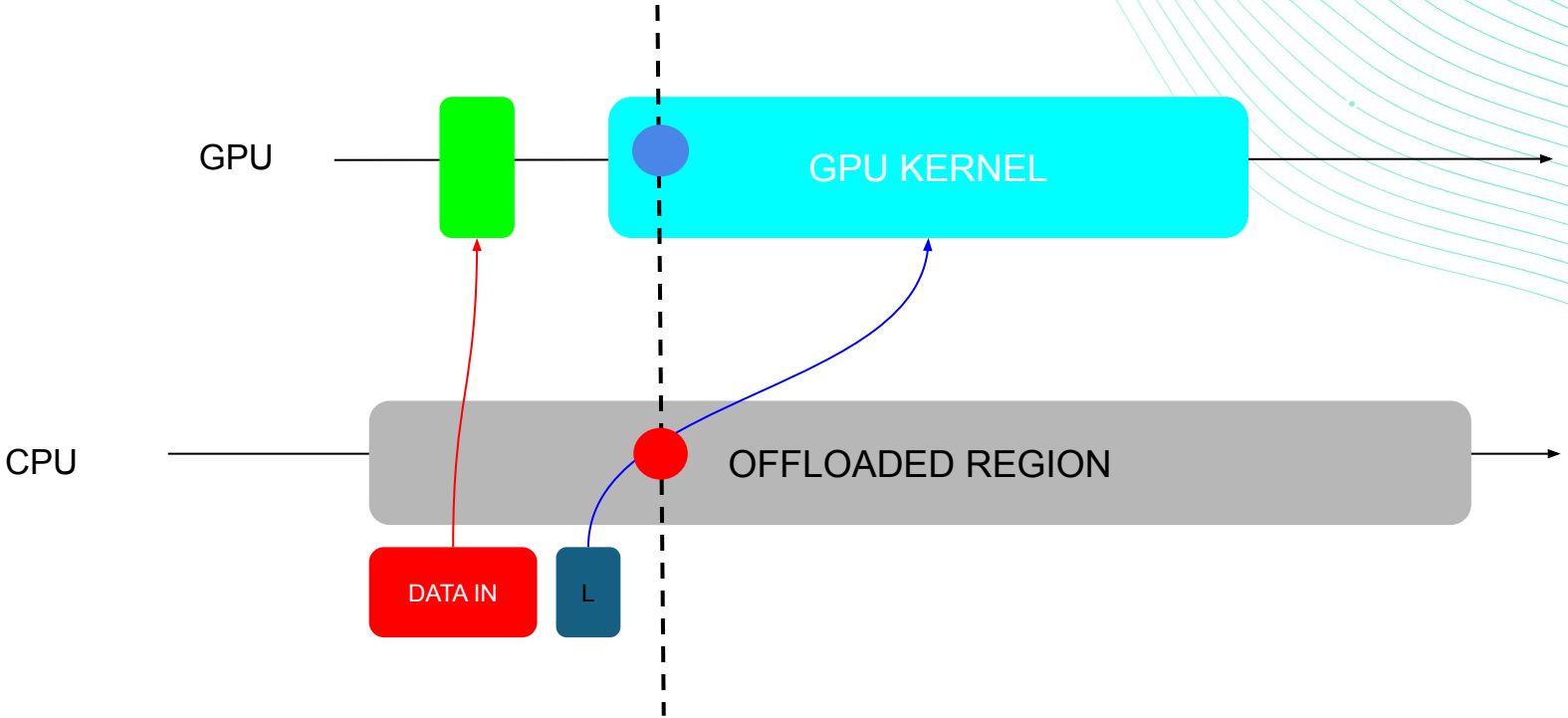
Traces in heterogeneous programs



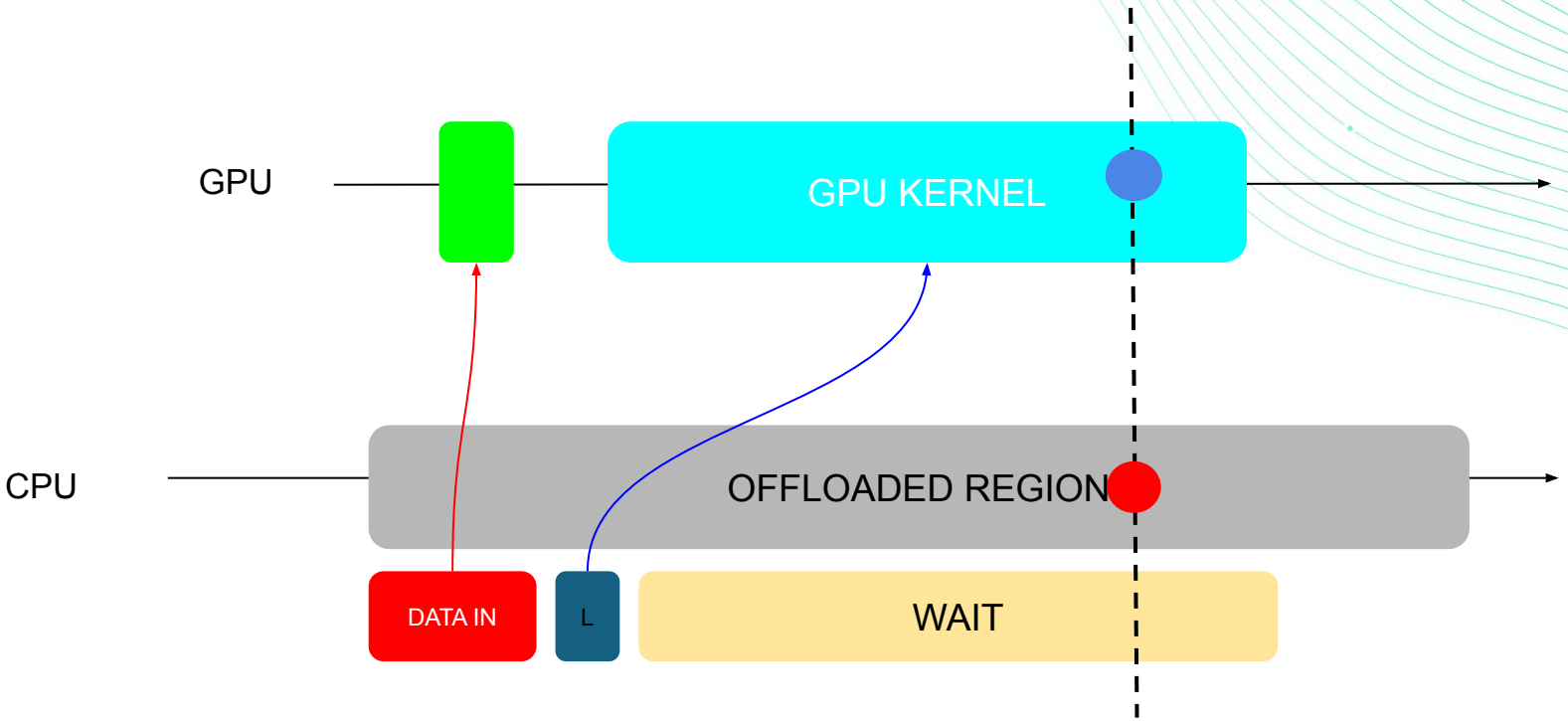
Traces in heterogeneous programs



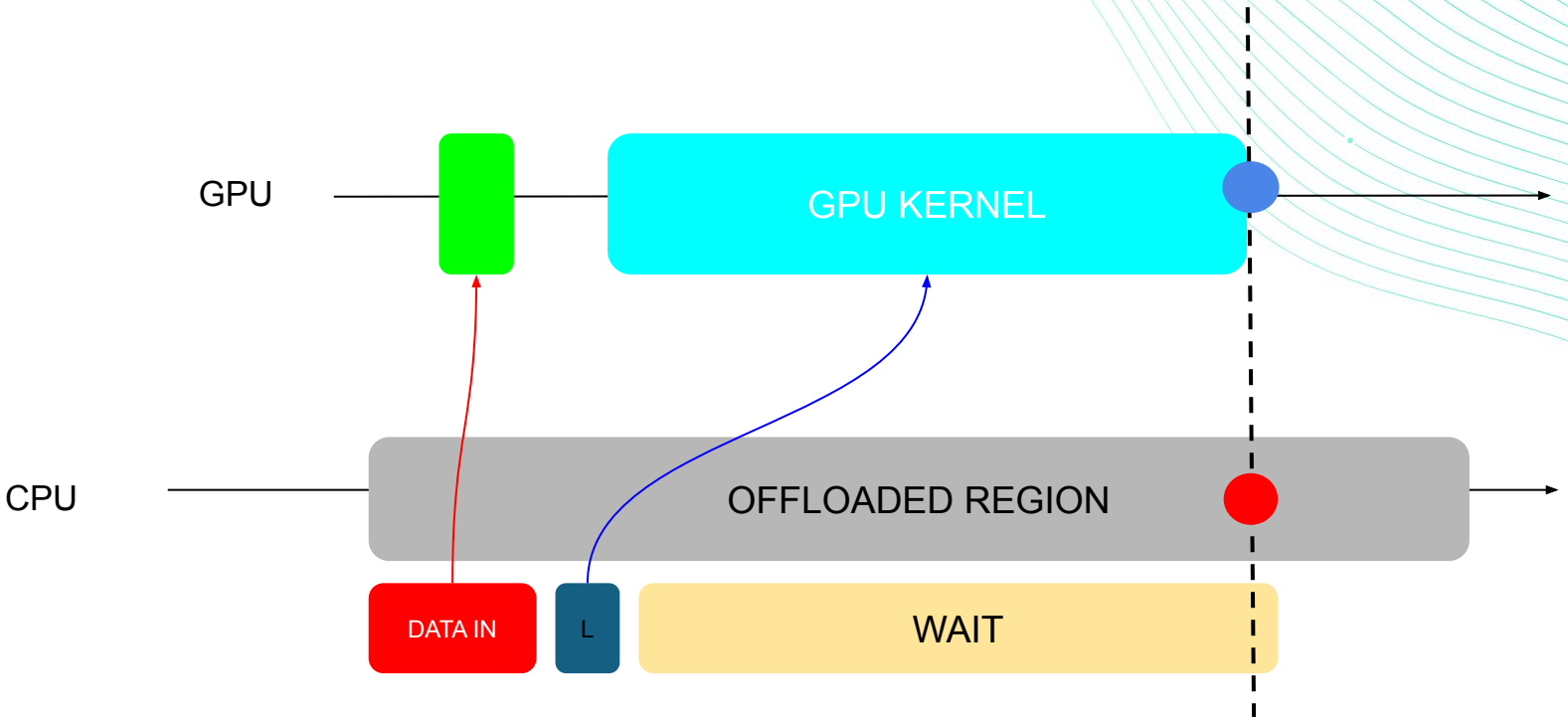
Traces in heterogeneous programs



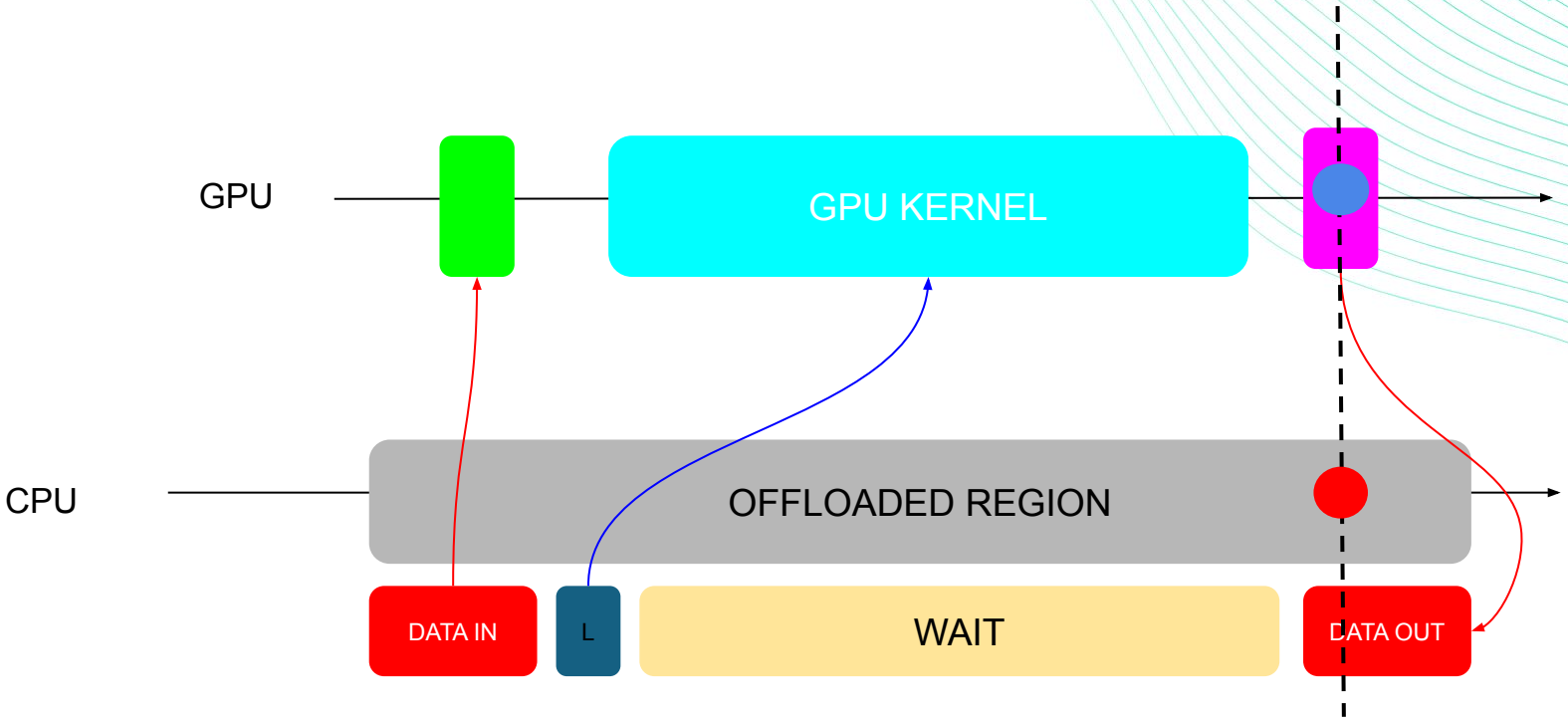
Traces in heterogeneous programs



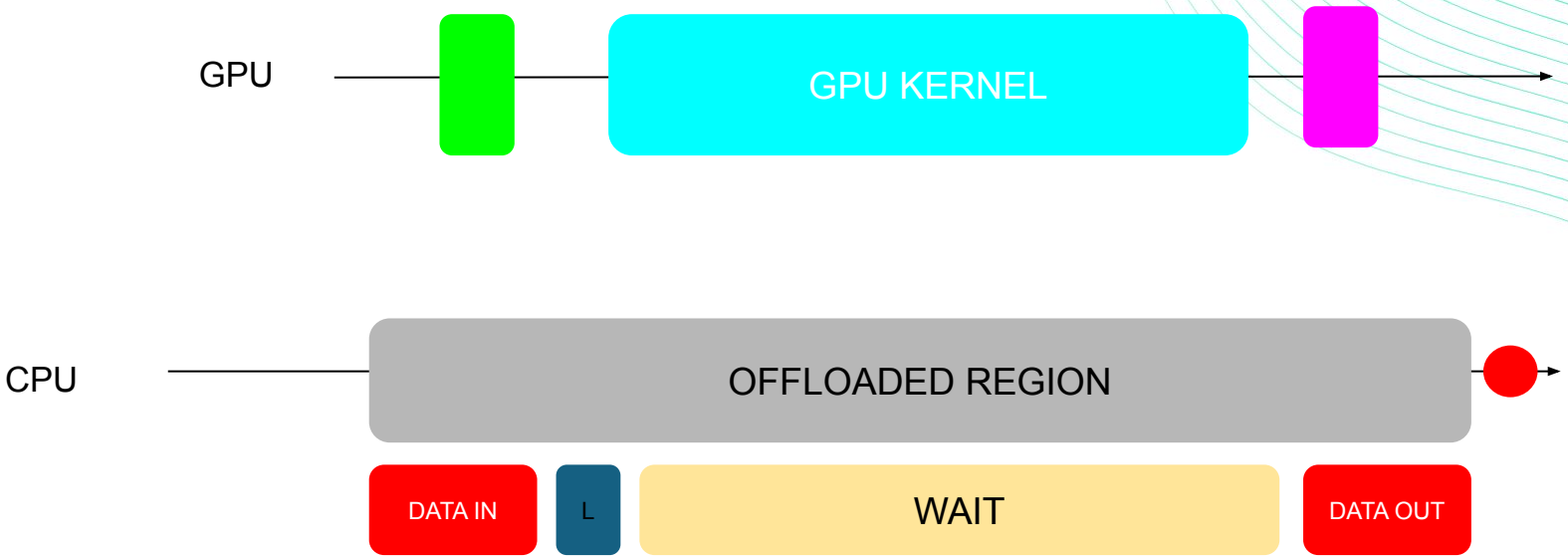
Traces in heterogeneous programs



Traces in heterogeneous programs



Traces in heterogeneous programs



Data clauses

Data clauses allow the programmer to tell the compiler **which data to move and when**

Fortran

```
!$acc parallel loop copyout(a(1:N))
do i = 1, N
     a(i) = 0
end do
```

I don't need the initial value of a, so I'll only copy it out of the region at the end.

```
copy(array[starting_index:length])
```

C/C++

```
copy(array(starting_index:ending_index))
```

Fortran

- Fortran programmers can rely on the self-describing nature of Fortran arrays
- C/C++ programmers will frequently need to give additional information to the compiler so that it will know **how large an array to allocate on the device and how much data needs to be copied**

e.g. copy(array[:]) copy(array[:]) copy(array[:N])

Data clauses

Data clauses allow the programmer to tell the compiler **which data to move and when**

Fortran

```
!$acc parallel loop copyout(a(1:N))
do i = 1, N
     a(i) = 0
end do
```

I don't need the initial value of a, so I'll only copy it out of the region at the end.

```
copy(array[starting_index:length])
```

C/C++

```
copy(array(starting_index:ending_index))
```

Fortran

- Fortran programmers can rely on the self-describing nature of Fortran arrays
- C/C++ programmers will frequently need to give additional information to the compiler so that it will know **how large an array to allocate on the device and how much data needs to be copied**

e.g. copy(array[:]) copy(array[:]) copy(array[:N])

array shaping: **can move also chunks** of arrays (minimize data movements)

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:** A result that isn't overwriting the input data structure.
`create( list )`	**Allocates memory on GPU but does not copy.** **Principal use:** Temporary arrays.

Implicit data management

```
while ( error > tol && iter < iter_max )
{
  error = 0.0;
  #pragma acc kernels
  {
    for( int j = 1; j < n-1; j++)
    {
       for( int i = 1; i < m-1; i++ )
       {
       Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]+ A[j-1][i] + A[j+1][i]);
       error = fmax( error, fabs(Anew[j][i] - A[j][i]));
       }
    }
    for( int j = 1; j < n-1; j++)
    {
       for( int i = 1; i < m-1; i++ )
       {
       A[j][i] = Anew[j][i];
       }
    }
  }
}
```

The runtime **automatically handles data** movements

For any parallel or kernels construct, it will move data in and out the GPU automatically

This might be very inefficient if the parallel region is inside a loop → data would be moved in/out the GPU for each “kernels”

Structured data directives

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

a, b, c must be visible in the device memory

a → in
b → in
c → out

Structured data directives

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

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

 #pragma acc end data
```

a, b, c must be visible in the device memory

a → in
b → in
c → out

we can explicitly tell the compiler to copyin(a,b) and copyout(c) for a given region of the source code

The acc data directives opens a **data region**

Structured data directives

```
#pragma acc parallel loop
for(int i = 0; i < N; i++){
    a[i] = a[i] + 1.0;
 }
#pragma acc parallel loop
for(int i = 0; i < N; i++){
    b[i] = 2.0;
 }
#pragma acc parallel loop
for(int i = 0; i < N; i++){
    c[i] = a[i] + b[i];
 }
```

a copied in an out

b copied in and out

a,b copied in and c copied out

Structured data directives

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

#pragma acc parallel loop
for(int i = 0; i < N; i++){
    a[i] = a[i] + 1.0;
 }
#pragma acc parallel loop
for(int i = 0; i < N; i++){
    b[i] = 2.0;
 }
#pragma acc parallel loop
for(int i = 0; i < N; i++){
    c[i] = a[i] + b[i];
 }

 #pragma acc end data
```

Inside the data region, the runtime knows a,b,c are in GPU memory

Enclosing parallel/kernels in the same data region reduces the number of data copies

```
a,b copied in

...gpu work...

c copied out
```

Implicit data region

```
#pragma acc parallel loop [copy(a[0:N])]
for(int i = 0; i < N; i++){
     a[i] = a[i] + 1.0;
}
```

parallel and kernels open an implicit data region

```
#pragma acc data copy(a[0:N]){
  #pragma acc parallel loop
  for(int i = 0; i < N; i++){
     a[i] = a[i] + 1.0;
  }
}
```

The data region extends for the extension of the parallel/kernel region

Unstructured data directives

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

#pragma acc parallel loop
for(int i = 0; i < N; i++){
    a[i] = a[i] + 1.0;
 }
#pragma acc parallel loop
for(int i = 0; i < N; i++){
    b[i] = 2.0;
 }
#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])
```

enter/exit data are used to create/upload or delete/download data

enter data

- + create
- + copyin

exit data

- + delete
- + copyout

enter/exit data perform only data movement, do not open any data region

Unstructured data directives

Unstructured

- Can have multiple starting/ending points
- Can branch across multiple functions
- Memory exists until explicitly deallocated

```
#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)
```

Structured

- Must have explicit start/end points
- Must be within a single function
- Memory only exists within the data region

```
#pragma acc data copyin(a[0:N],b[0:N]) \
          copyout(c[0:N])
{
          #pragma acc parallel loop
          for(int i = 0; i < N; i++){
                    c[i] = a[i] + b[i];
          }

}
```

Unstructured data directives

```
int* allocate(int size)
{
    int *ptr = (int*) malloc(size * sizeof(int));

    return ptr;
}

void deallocate(int *ptr)
{

    free(ptr);
}

int main()
{
    int *ptr = allocate(100);

    for( int i = 0; i < 100; i++ )
    {
        ptr[i] = 0;
    }

    deallocate(ptr);
}
```

Data lifetime might not be restricted to a single routine

data is created

Unstructured data directives

```
int* allocate(int size)
{
    int *ptr = (int*) malloc(size * sizeof(int));

    return ptr;
}

void deallocate(int *ptr)
{

    free(ptr);
}

int main()
{
    int *ptr = allocate(100);

    for( int i = 0; i < 100; i++ )
    {
        ptr[i] = 0;
    }

    deallocate(ptr);
}
```

Data lifetime might not be restricted to a single routine

data is created

data is used

Unstructured data directives

```
int* allocate(int size)
{
    int *ptr = (int*) malloc(size * sizeof(int));

    return ptr;
}

void deallocate(int *ptr)
{

    free(ptr);
}

int main()
{
    int *ptr = allocate(100);

    for( int i = 0; i < 100; i++ )
    {
        ptr[i] = 0;
    }

    deallocate(ptr);
}
```

Data lifetime might not be restricted to a single routine

data is created

data is used

data is deallocated

Unstructured data directives

```
int* allocate(int size)
{
    int *ptr = (int*) malloc(size * sizeof(int));
    #pragma acc enter data create(ptr[0:size])
    return ptr;
}

void deallocate(int *ptr)
{
    #pragma acc exit data delete(ptr)
    free(ptr);
}

int main()
{
    int *ptr = allocate(100);

    #pragma acc parallel loop
    for( int i = 0; i < 100; i++ )
    {
        ptr[i] = 0;
    }

    deallocate(ptr);
}
```

Data lifetime might not be restricted to a single routine

data is created

data is used

data is deallocated

Unstructured data directives

Be careful when you manage data across multiple routines.

If you try copying data that is already PRESENT on the GPU, <u>the copy is not done.</u>

< a modified on the host >

#pragma acc enter data copyin(a[0:N]) → host and device copies are in sync

< a modified on the host > → host and device copies are out of sync

#pragma acc data copyin(a[0:N]) **! copy is ignored**

< a used on the GPU > → **host and device copies are out of sync**

Update directive

Data is already on the GPU and you need to update the value on the CPU or device

```
do_something_on_device()
#pragma acc update host(a)
```

→ Copy “a” from GPU to CPU

```
do_something_on_host()
#pragma acc update device(a)
```

→ Copy “a” from CPU to GPU

! The **update can also be partial** (shaping) , e.g. `#pragma acc update host(a[0:N/2])`

Update directive

update: Explicit transfers data between the host and device

Useful when you want to synchronise data in the middle of a data region

self / host makes host data agree with device data

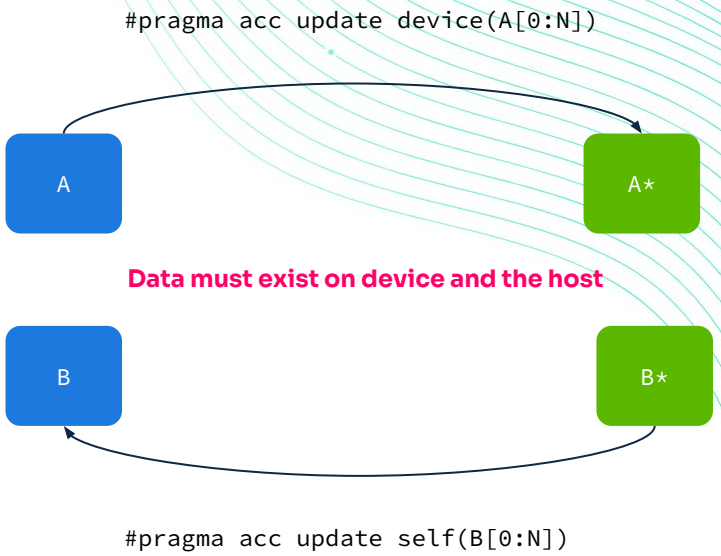
device makes device data agree with host data

C/C++

```
#pragma acc update self(x[0:count])
#pragma acc update device(x[0:count])
```

Fortran

```
!$acc update self(x[1:count])
!$acc update device(x[1:count])
```



#pragma acc declare create(a)

Declare directive

The declare directive specifies that a variable or array **has to be allocated in the device memory for the duration of the implicit data region of function**.

- Used in the declaration section of function
- May specify whether the data have to be transferred and how (create, copy, etc)
- If referring to **global variables, the implicit region is the whole program**

C/C++

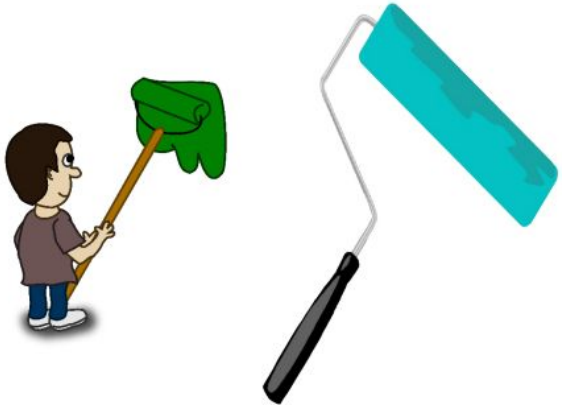
```
#pragma acc declare create(a[0:N])
```

Fortran

```
real a(100)
!$acc declare create(a)
```

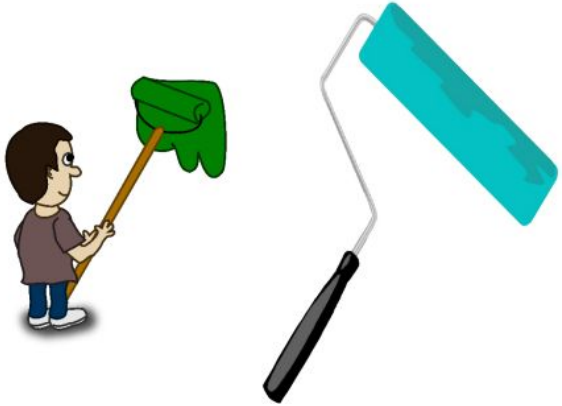
Gang, worker, vector

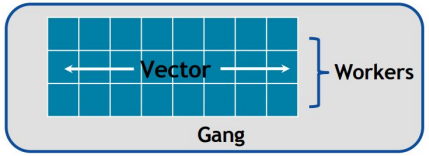
Gang, worker, vector



Gang, worker, vector

Gang, worker, vector





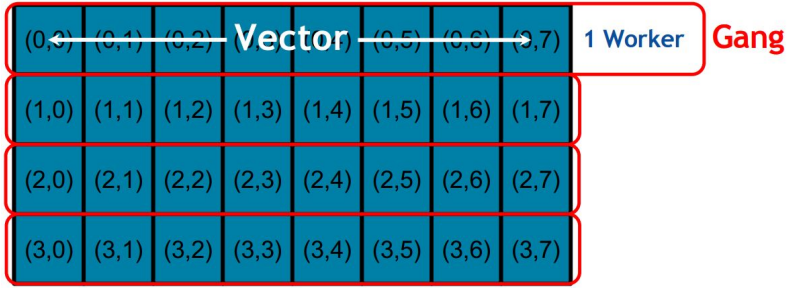
Gangs do not share resources, do not synchronize, are independent groups of working units

Workers in a gang can share resources, can synch, each one having a roller of a given size (vector length)

A **vector** has the ability to run a single instruction on multiple data elements

Gang, worker, vector

```
#pragma acc parallel loop gang
for( i = 0; i < N; i++ )
   #pragma acc loop vector
   for( j = 0; j < M; j++ )
      < loop code >
```



The gang parallelism applies to the outer loop

A vector is the lowest level of parallelism, and every gang will have at least 1 vector

Usually the compiler generates N gangs with one worker and a vector of size 128 on NVIDIA gpus

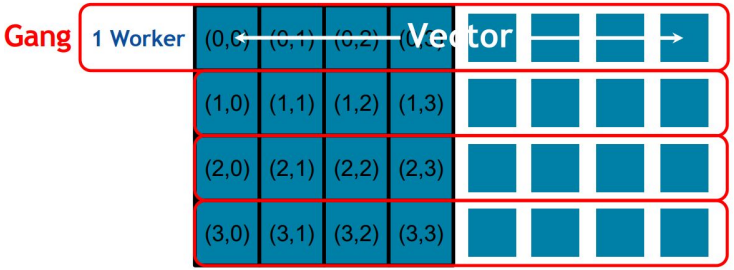
check with -Minfo!

Gang, worker, vector

```
#pragma acc parallel num_workers(2)
#pragma acc loop gang worker
for( i = 0; i < N; i++ )
   #pragma acc loop vector
   for( j = 0; j < M; j++ )
      < loop code >
```

Sometimes having more workers in a gang helps to better map the data

Especially if the size for the vector length is small

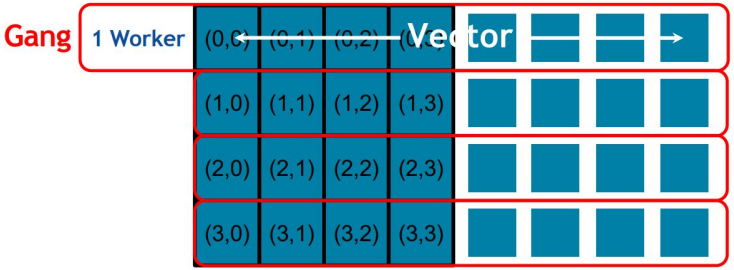


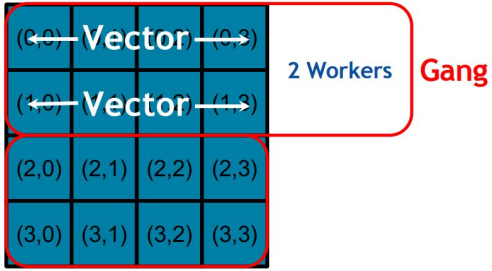
Gang, worker, vector

```
#pragma acc parallel num_workers(2)
#pragma acc loop gang worker
for( i = 0; i < N; i++ )
   #pragma acc loop vector
   for( j = 0; j < M; j++ )
      < loop code >
```

Sometimes having more workers in a gang helps to better map the data

Especially if the size for the vector length is small





Gang, worker, vector

```
parallel:

 ✳   num_gangs(n)
 ✳   num_workers(n)
 ✳   vector_length(n)

kernels:

 ✳   gang(n)
 ✳   worker(n)
 ✳   vector(n)
```

The size of a gang is

num_workers * vector_length

the maximum vector_length is 1024

the minimum vector_length (NVIDA) is 32

the maximum size of a gang is 1024

Routine directive

`routine` Specifies that the compiler should generate a device copy of the function/subroutine

CLAUSES

- `gang/worker/vector/seq:`
  parallelism for loops contained in the routine
- `bind()`
  optional name of the routine at call-site
- `no_host`
  the routine will only be used on the device

```
void square_array(float *arr, int length) {
    for(int i = 0; i < length; ++i) {
        arr[i] = arr[i] * arr[i];
    }
}
```

```
int main() {
    const int size = 100; float data[size];

    #pragma acc parallel loop
    for(int i = 0; i < size; ++i) {
        data[i] = i;
    }
    for(int i = 0; i < size; ++i) {
            square_array(&data[i], 1);
    }
}
```

Routine directive

`routine` Specifies that the compiler should generate a device copy of the function/subroutine

CLAUSES

* `gang/worker/vector/seq:`

  parallelism for loops contained in the routine

* `bind()`

  optional name of the routine at call-site

* `no_host`

  the routine will only be used on the device

```
#pragma acc routine worker
void square_array(float *arr, int length) {
    #pragma acc parallel loop worker
    for(int i = 0; i < length; ++i) {
        arr[i] = arr[i] * arr[i];
    }
}


int main() {
    const int size = 100; float data[size];

    #pragma acc parallel loop
    for(int i = 0; i < size; ++i) {
        data[i] = i;
    }
    #pragma acc parallel
    {
        #pragma acc loop gang
        for(int i = 0; i < size; ++i) {
            square_array(&data[i], 1);
        }
    }
}
```

Routine directive

The **seq** clause in the **routine** directive for OpenACC is used to indicate that the specified routine should be executed **sequentially in one device thread** (GPU).

At call the compiler needs to know

- Function will be available on the GPU (!$acc routine)
- It is a sequential routine, executed by one device thread (seq)

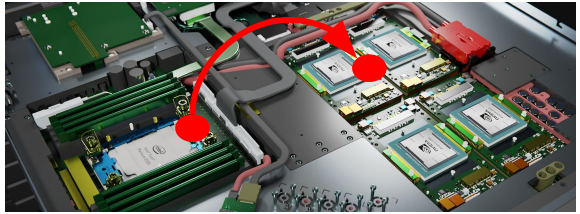
At call site

- Function is called in a parallel loop region (parallel loop)
- Each thread in the loop will call it and execute its own instance

```
module b1
 contains
  real function sqab(a)
     !$acc routine seq
     real :: a
     sqab = sqrt(abs(a))
  end function
 end module
```

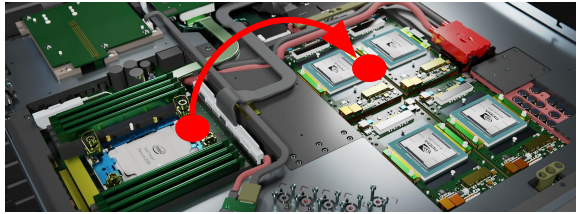
```
subroutine test( x, n )
     use b1
     real, dimension(*) :: x
     integer :: n
     integer :: i
     !$acc parallel loop copy(x(1:n))
     do i = 1, n
        x(i) = sqab(x(i))
     enddo
 end subroutine
```

Asynchronous programming

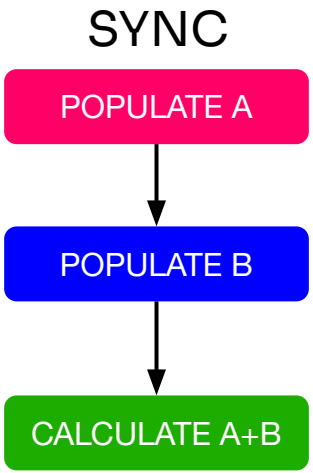


```
for (i = 0; i<n: i++)
 a[i] = 1

for (i = 0; i<n: i++)
 b[i] = 1

for (i = 0; i<n: i++)
c[i] = a[i] + b[i]
```

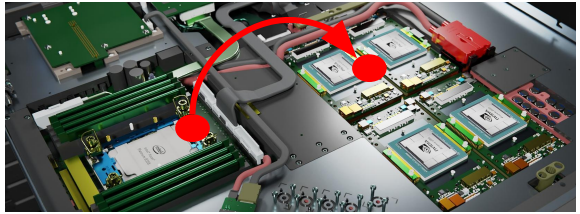
Asynchronous programming



```
for (i = 0; i<n: i++)
 a[i] = 1

for (i = 0; i<n: i++)
 b[i] = 1

for (i = 0; i<n: i++)
c[i] = a[i] + b[i]
```

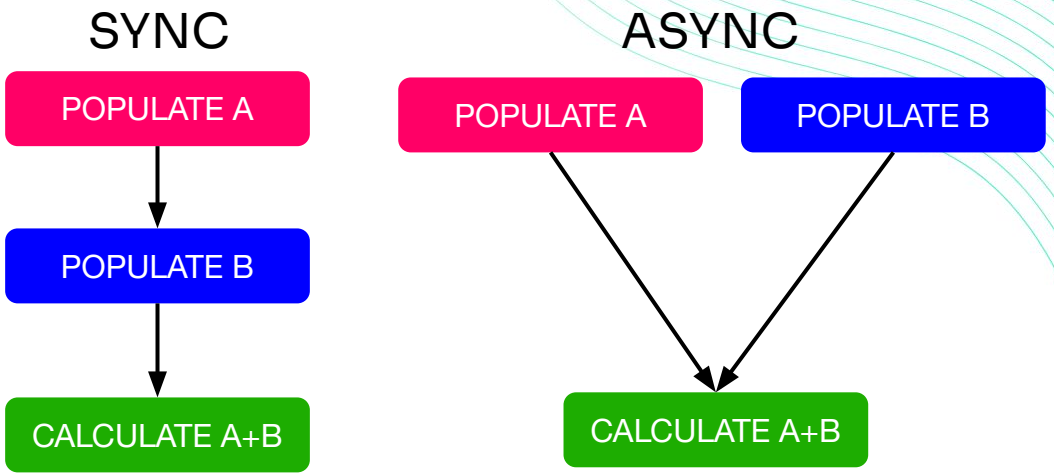


Asynchronous programming



```
for (i = 0; i<n: i++)
 a[i] = 1

for (i = 0; i<n: i++)
 b[i] = 1

for (i = 0; i<n: i++)
c[i] = a[i] + b[i]
```

Enables concurrent operations on a GPU



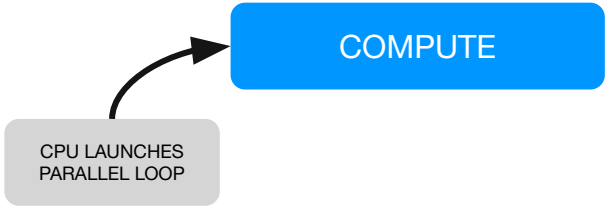
Asynchronous programming

So far all of the OpenACC directives operates synchronously with the host, i.e host will wait for device to complete its execution.

async

It can be used on parallel, kernel and update directives

```
#pragma acc loop async
for (i = 0; i<n: i++)
c[i] = a[i] + b[i]
```



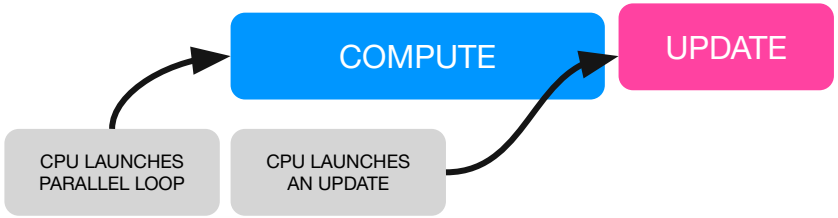
Asynchronous programming

So far all of the OpenACC directives operates synchronously with the host, i.e host will wait for device to complete its execution.

async

It can be used on parallel, kernel and update directives

```
#pragma acc loop async
for (i = 0; i<n: i++)
c[i] = a[i] + b[i]
#pragma acc update self[c[0:N]] async
```



Asynchronous programming

So far all of the OpenACC directives operates synchronously with the host, i.e host will wait for device to complete its execution.

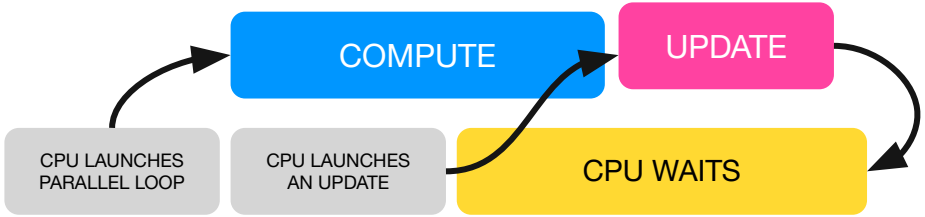
`async`

It can be used on parallel, kernel and update directives

`wait`

Instructs the runtime to wait for the past asynchronous operation to complete before proceeding

```
#pragma acc loop async
for (i = 0; i<n: i++)
c[i] = a[i] + b[i]
#pragma acc update self[c[0:N]] async
#pragma acc wait
```



Asynchronous programming

So far all of the OpenACC directives operates synchronously with the host, i.e host will wait for device to complete its execution.

`async`

It can be used on parallel, kernel and update directives

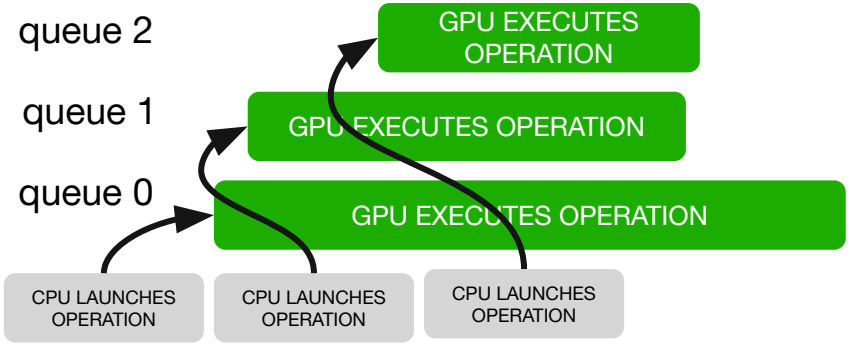
`wait`

Instructs the runtime to wait for the past asynchronous operation to complete before proceeding

`async(N)`

a number can be added to `async` and `wait`, in order to identify the "queue" for the async operation

```
#pragma acc loop async(N)
for (i = 0; i<n: i++)
c[i] = a[i] + b[i]
#pragma acc update self[c[0:N]] async(N)
#pragma acc wait
```



Asynchronous programming

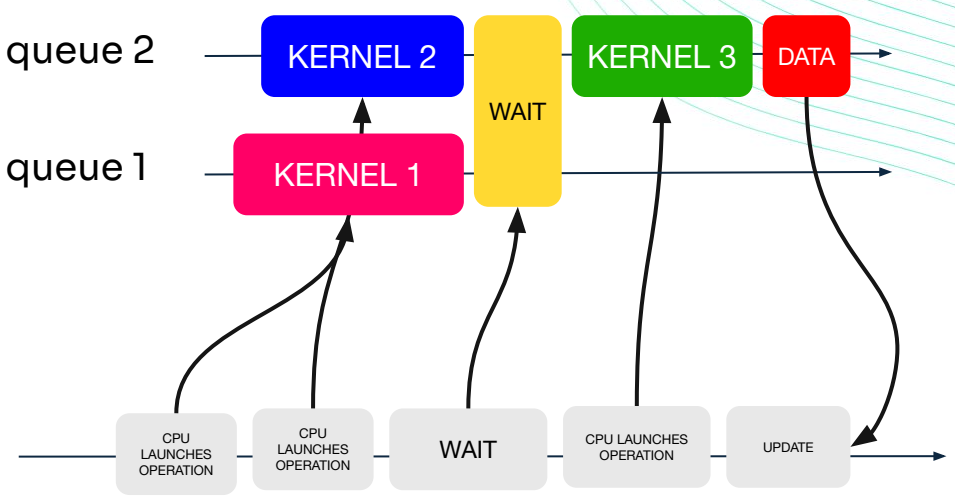
```
#pragma acc parallel loop async(1)
for (i = 0; i<n: i++)
 a[i] = 1
#pragma acc parallel loop async(2)
for (i = 0; i<n: i++)
 b[i] = 1
#pragma acc wait(1) async(2)
#pragma acc loop async(2)
for (i = 0; i<n: i++)
c[i] = a[i] + b[i]
#pragma acc update self[c[0:N]] async(2)
#pragma acc wait
```

KERNEL 1 IS COMPUTED ON “STREAM” 1

KERNEL 2 IS COMPUTED ON “STREAM” 2

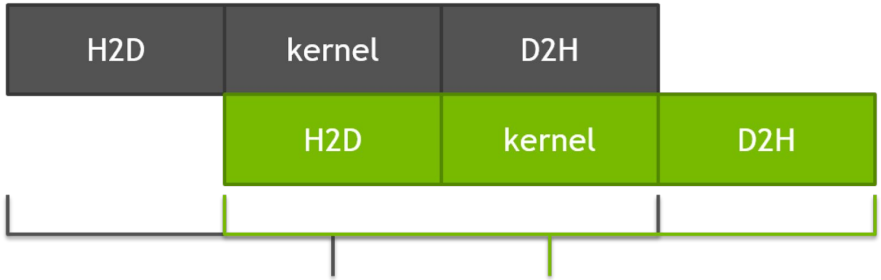
THIS KERNEL WAITS FOR OPERATION IN 1 TO BE COMPLETED AND THEN IS PUT IN THE QUEUE OF STREAM 2

WHEN STREAM2 HAS COMPLETED THE OPERATION, THE HOST IS UPDATED



Asynchronous programming

Two Independent Operations Serialized



Overlapping Copying and Computation

NOTE: In real applications, your boxes will not be so evenly sized.

CUDA - OpenACC interoperability

How to use a CUDA APIs in OpenACC code?

CUDA : use device buffer as input/output

```
ierr = ierr + cufftExecC2C(iplan1,a_d,b_d,CUFFT_FORWARD)
```

CUDA - OpenACC interoperability

How to use a CUDA APIs in OpenACC code?

CUDA : use device buffer as input/output

```
ierr = ierr + cufftExecC2C(iplan1,a_d,b_d,CUFFT_FORWARD)
```

OpenACC : the name of the variable is the the same for device and host buffer. `host_data use_device` directives clarifies which buffer should be passed.

CUDA - OpenACC interoperability

How to use a CUDA APIs in OpenACC code?

CUDA : use device buffer as input/output

```
ierr = ierr + cufftExecC2C(iplan1,a_d,b_d,CUFFT_FORWARD)
```

OpenACC : the name of the variable is the the same for device and host buffer. `host_data use_device` directives clarifies which buffer should be passed.

```
#pragma acc host_data use_device(a,b)
ierr = ierr + cufftExecC2C(iplan1,a,b,CUFFT_FORWARD)
#pragma acc end host_data
```

CUDA libraries

CUDA Math Libraries

GPU-accelerated math libraries lay the foundation for compute-intensive applications in areas such as molecular dynamics, computational fluid dynamics, computational chemistry, medical imaging, and seismic exploration.

cuBLAS

GPU-accelerated basic linear algebra (BLAS) library.

Learn More >

cuFFT

GPU-accelerated library for Fast Fourier Transform implementations.

Learn More >

cuRAND

GPU-accelerated random number generation.

Learn More >

cuSOLVER

GPU-accelerated dense and sparse direct solvers.

Learn More >

cuSPARSE

GPU-accelerated BLAS for sparse matrices.

Learn More >

cuTENSOR

GPU-accelerated tensor linear algebra library.

Learn More >

cuDSS

GPU-accelerated direct sparse solver library.

Learn More >

CUDA Math API

GPU-accelerated standard mathematical function APIs.

Learn More >

AmgX

GPU-accelerated linear solvers for simulations and implicit unstructured methods.

Learn More >

- Offload math to GPU
- Support data moved both with CUDA and OpenACC

Available in the CUDA toolkit or HPCSDK suite, can be linked by compilation flag

Multiple versions available

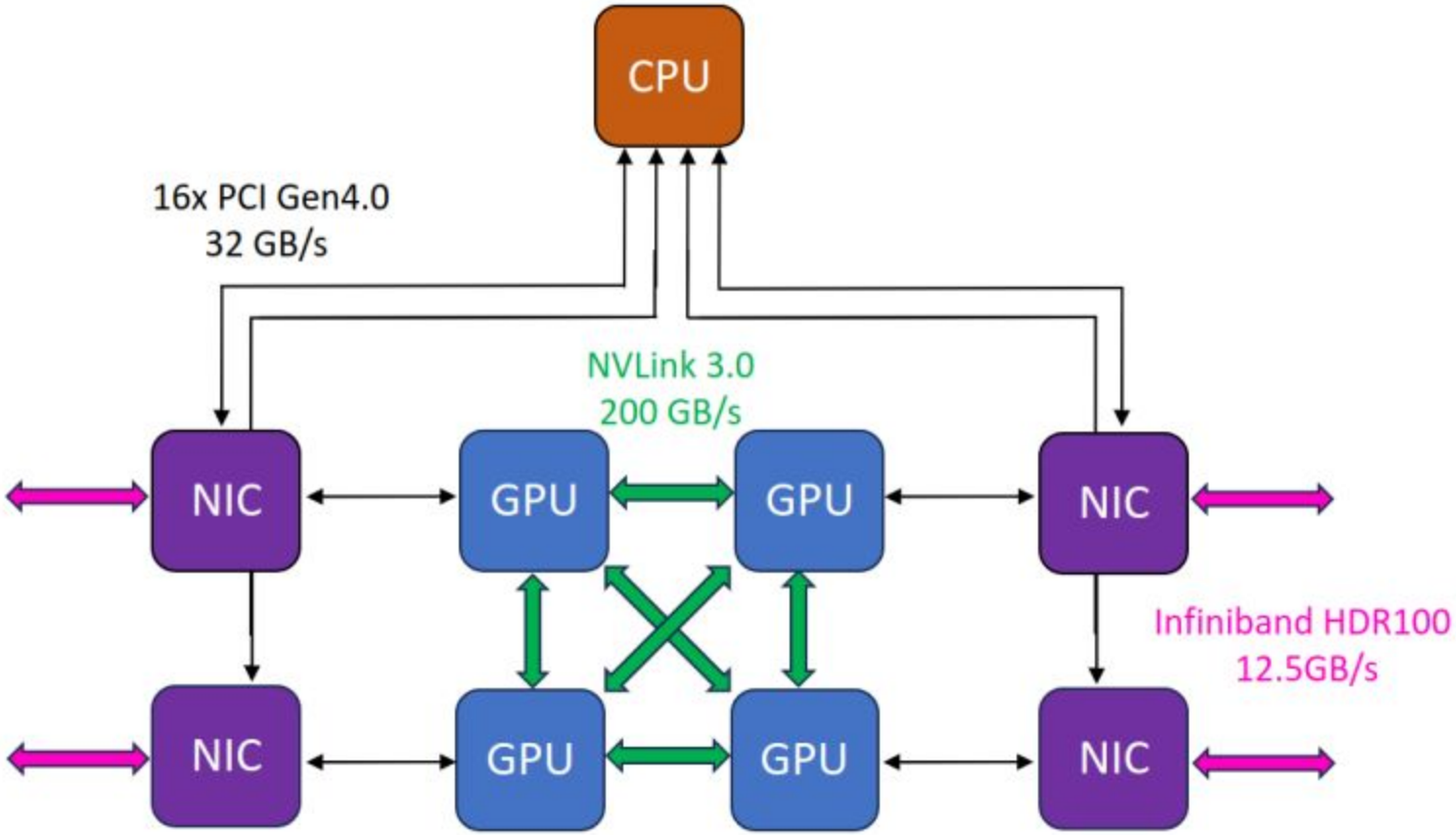
- single gpu (e.g. cuBLAS)
  - batched
  - multi-stream
- single-process multi GPU (cuBLASXt)
- multi-process multi-GPU (cuBLASMp)

CUDA libraries

CUDAFortran provides

- interfaces to CUDA C library APIs (cuBLAS, cuFFT, cuRANDS, cuSPARSE,...)
- interfaces to CPU or GPU API depending on the input type

```
use cufft
use openacc
. . .
!$acc data copyin(a), copyout(b,c)
ierr = cufftPlan2D(iplan1,m,n,CUFFT_C2C)
ierr = ierr +
cufftSetStream(iplan1,acc_get_cuda_stream(acc_async_sync))
!$acc host_data use_device(a,b,c)
ierr = ierr + cufftExecC2C(iplan1,a,b,CUFFT_FORWARD)
ierr = ierr + cufftExecC2C(iplan1,b,c,CUFFT_INVERSE)
!$acc end host_data

!$acc kernels
c = c / (m*n)
!$acc end kernels
!$acc end data
```


CPU
16x PCI Gen4.0
32 GB/s
NVLink 3.0
200 GB/s
NIC
GPU
GPU
NIC
Infiniband HDR100
12.5GB/s
NIC
GPU
GPU
NIC

EPICURE

Profiling with NSight Systems

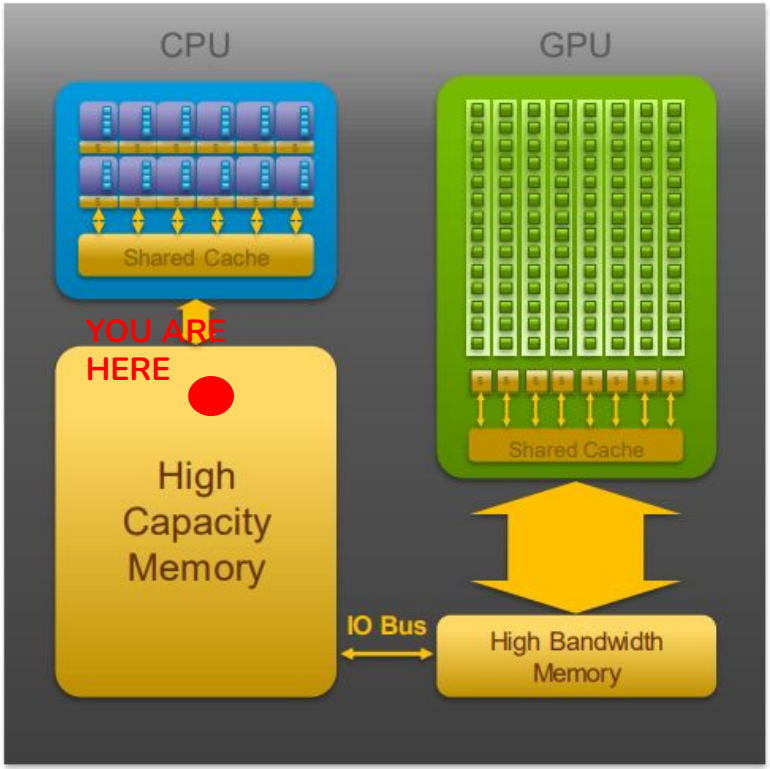
Epicure hackathon @ CINECA, Casalecchio di Reno (BO) Italy
28-31 October 2024

Co-funded by the European Union

EuroHPC Joint Undertaking

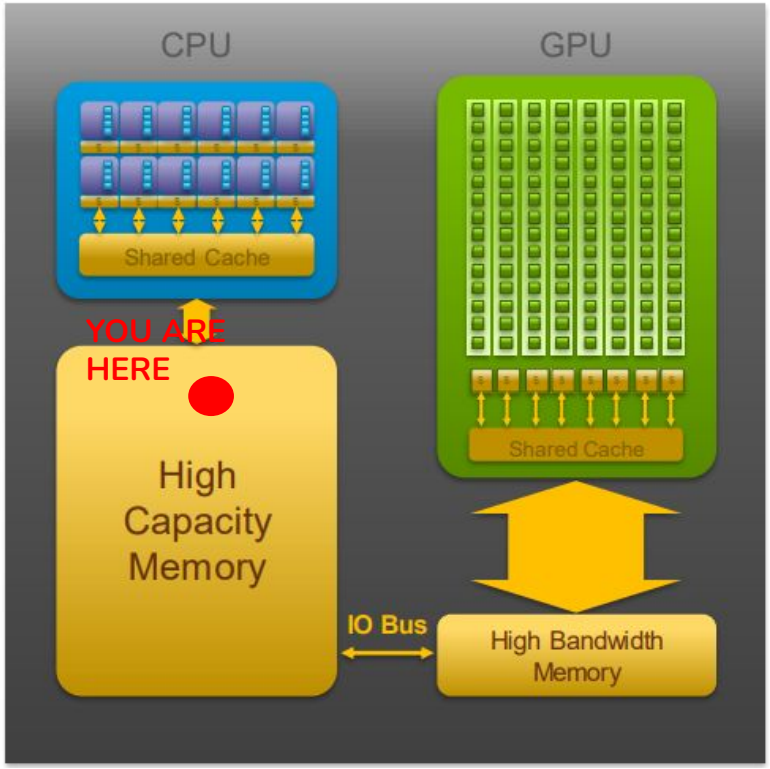
This project has received funding from the High Performance Computing Joint Undertaking under grant agreement No 101139786

Heterogeneous programming



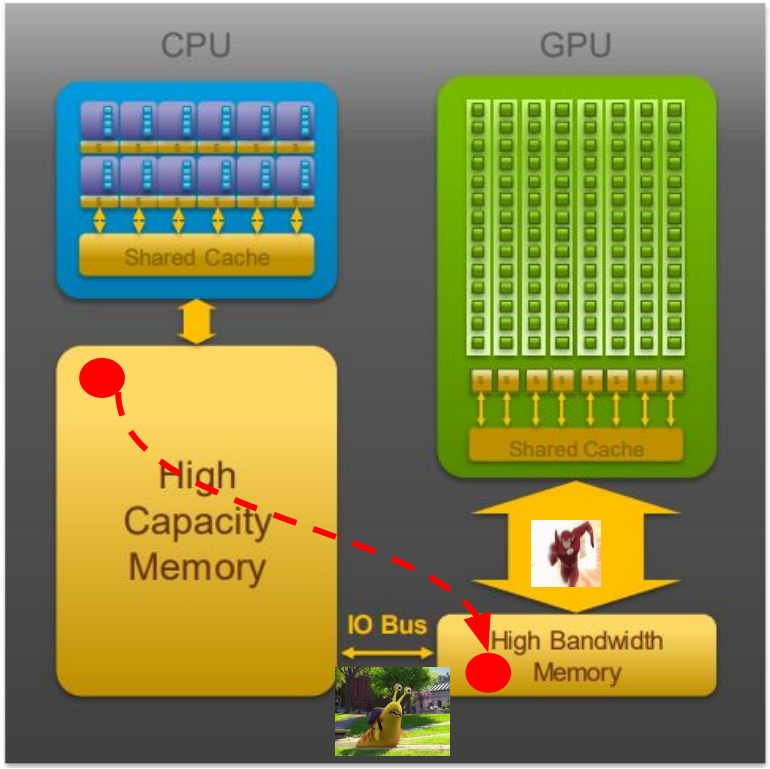
- The program starts on the CPU

Heterogeneous programming



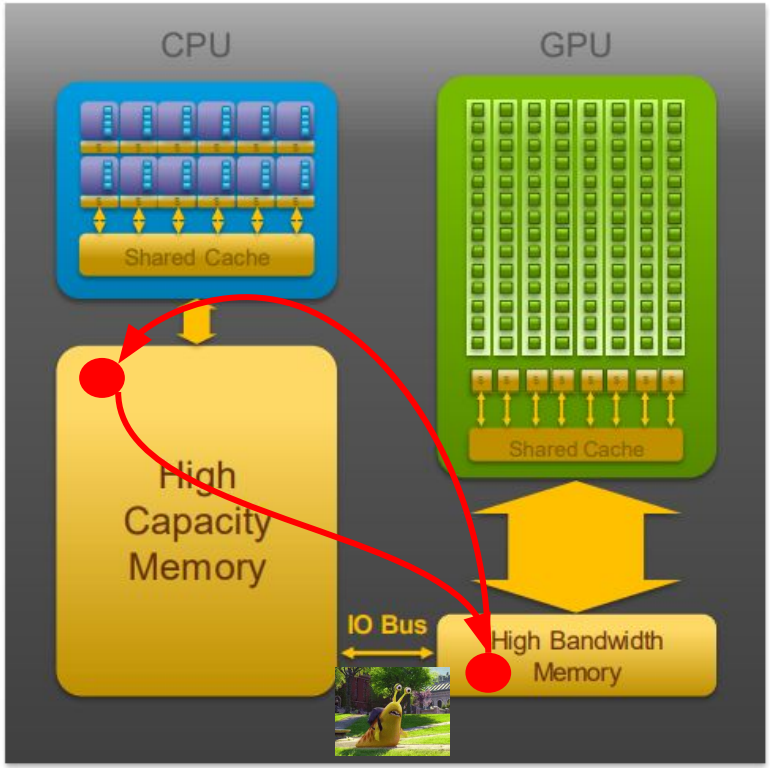
- The program starts on the CPU
- GPUs and CPUs have separated memories

Heterogeneous programming



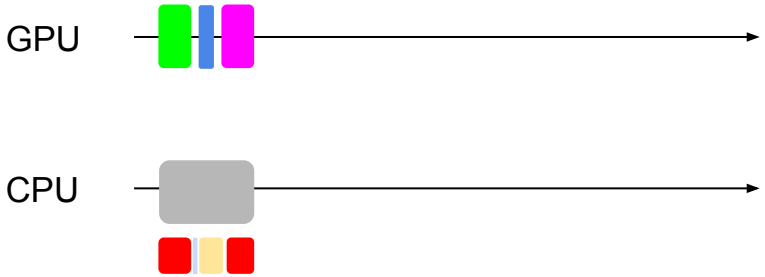
- The program starts on the CPU
- GPUs and CPUs have separated memories
- GPUs need data in their own memory

Heterogeneous programming

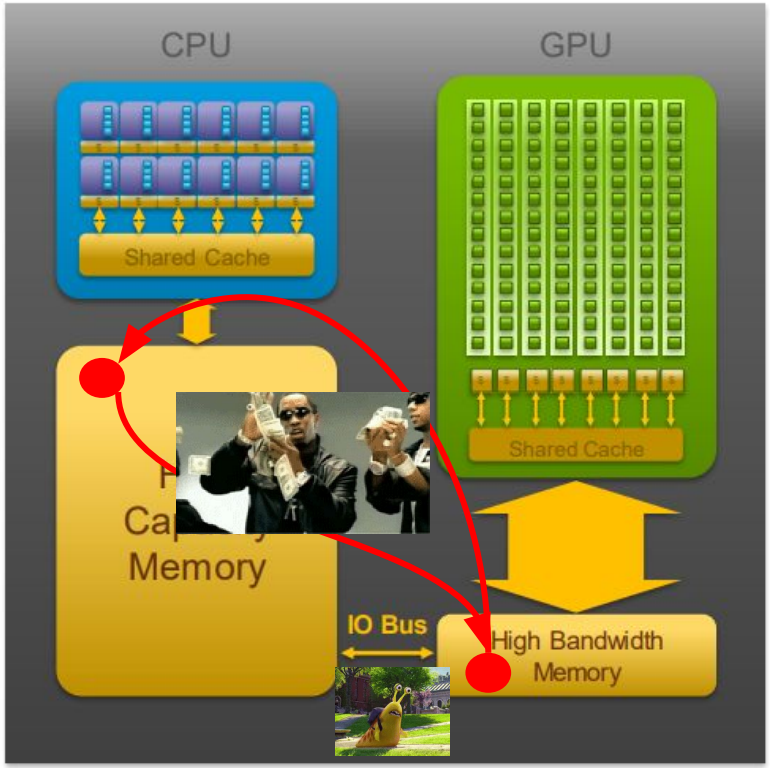


- The program starts on the CPU
- GPUs and CPUs have separated memories
- GPUs need data in their own memory
- IO bus is slow compared to GPU BW

MOVING DATA IS EXPENSIVE

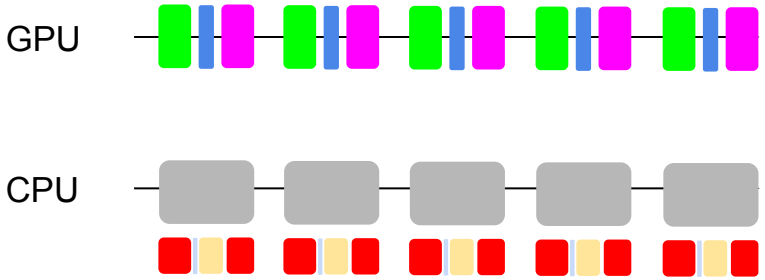


Heterogeneous programming

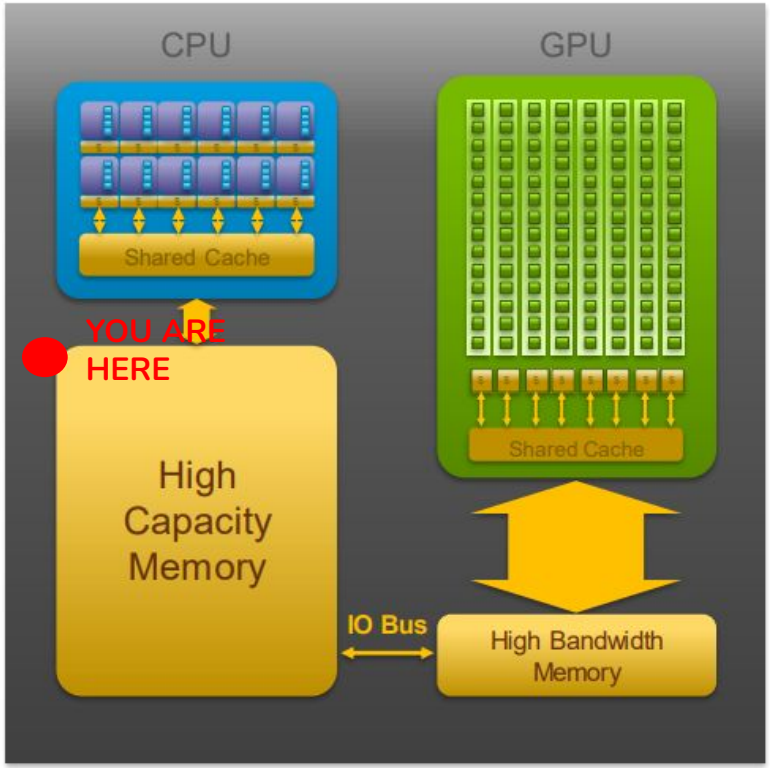


- The program starts on the CPU
- GPUs and CPUs have separated memories
- GPUs need data in their own memory
- IO bus is slow compared to GPU BW

MOVING DATA IS EXPENSIVE



Heterogeneous programming

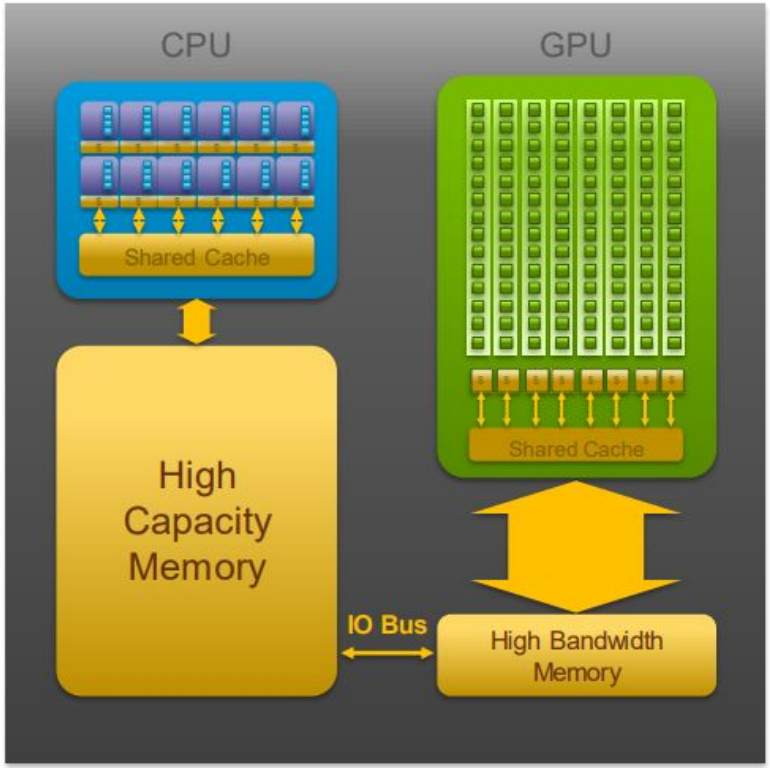


- The program starts on the CPU
- GPUs and CPUs have separated memories
- GPUs need data in their own memory
- IO bus is slow compared to GPU BW

MOVING DATA IS EXPENSIVE

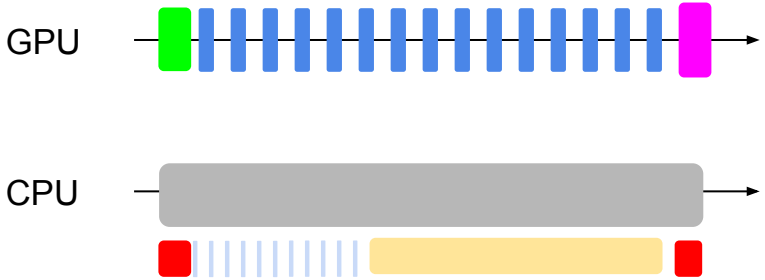
IMPROVE DATA LOCALITY

Heterogeneous programming

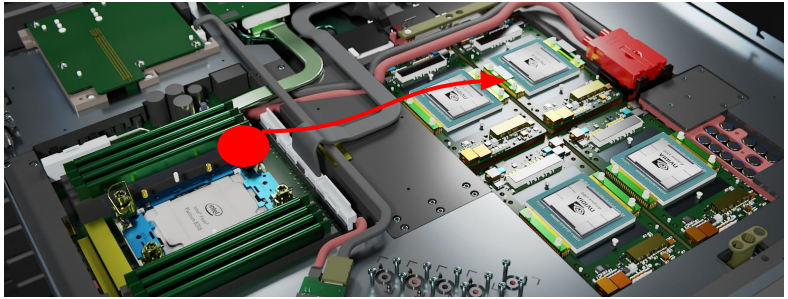


- The program starts on the CPU
- GPUs and CPUs have separated memories
- GPUs need data in their own memory
- IO bus is slow compared to GPU BW

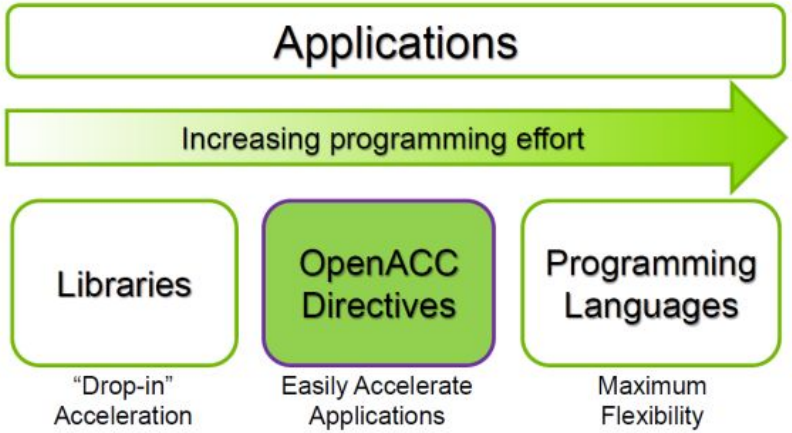
MOVING DATA IS EXPENSIVE

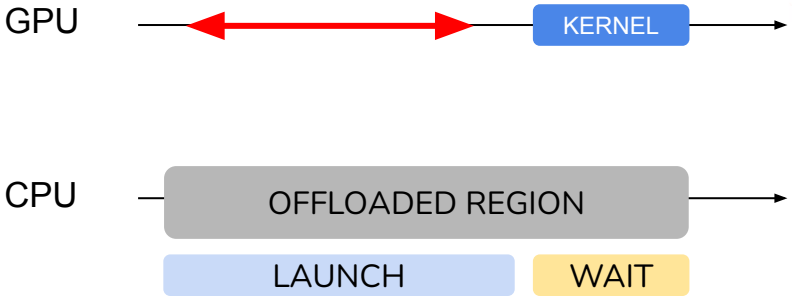


Heterogeneous programming

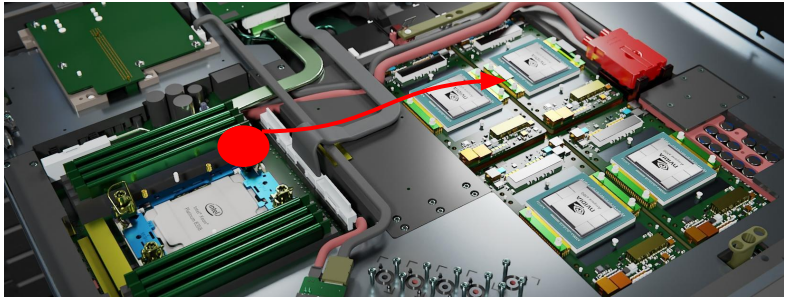


- The program starts on the CPU
- Many ways to offload kernels to GPUs
- There is a time needed to launch the kernels ("latency")

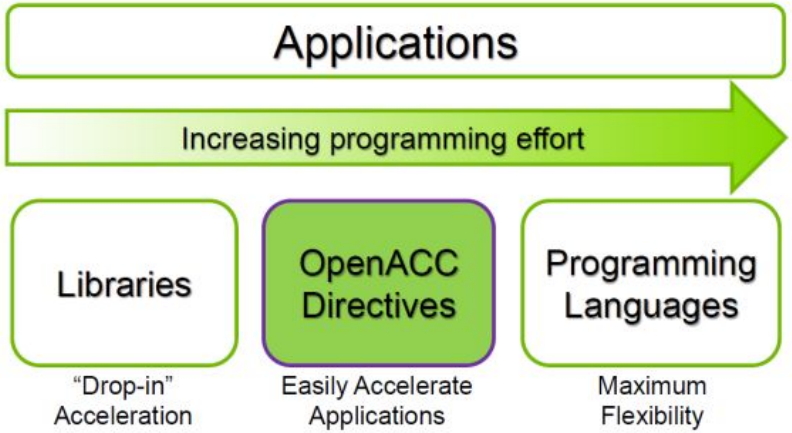


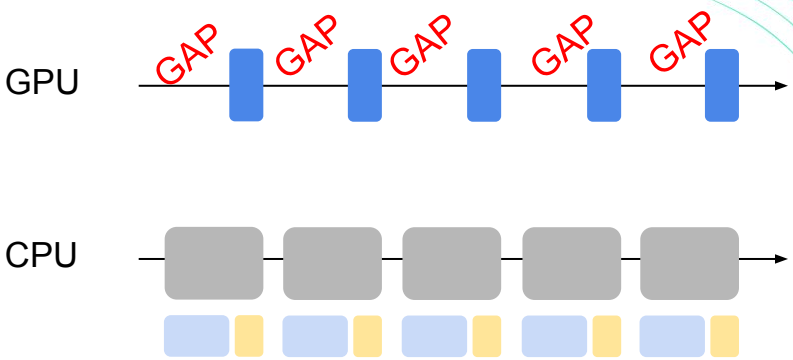


Heterogeneous programming

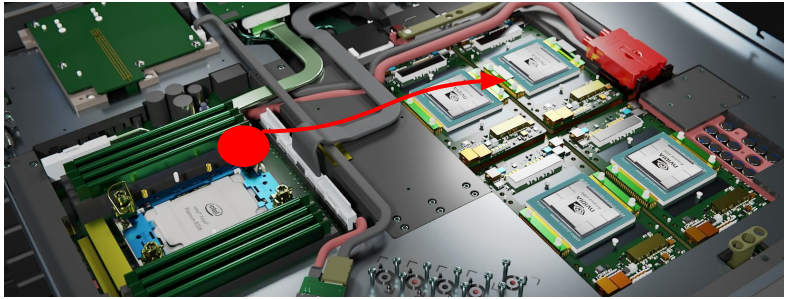


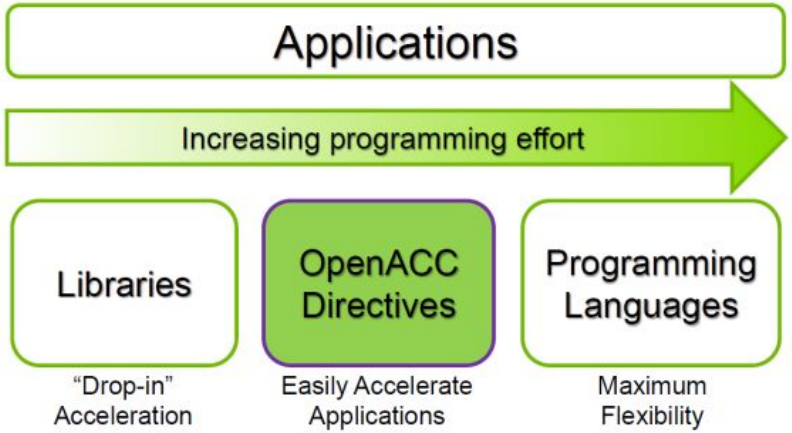
- The program starts on the CPU
- Many ways to offload kernels to GPUs
- There is a time needed to launch the kernels ("latency")





Heterogeneous programming





- The program starts on the CPU
- Many ways to offload kernels to GPUs
- There is a time needed to launch the kernels ("latency")

AVOID SMALL KERNELS

EXPOSE AS MUCH PARALLELISM AS POSSIBLE

COLLAPSE LOOPS, REFACTOR

NSight Systems

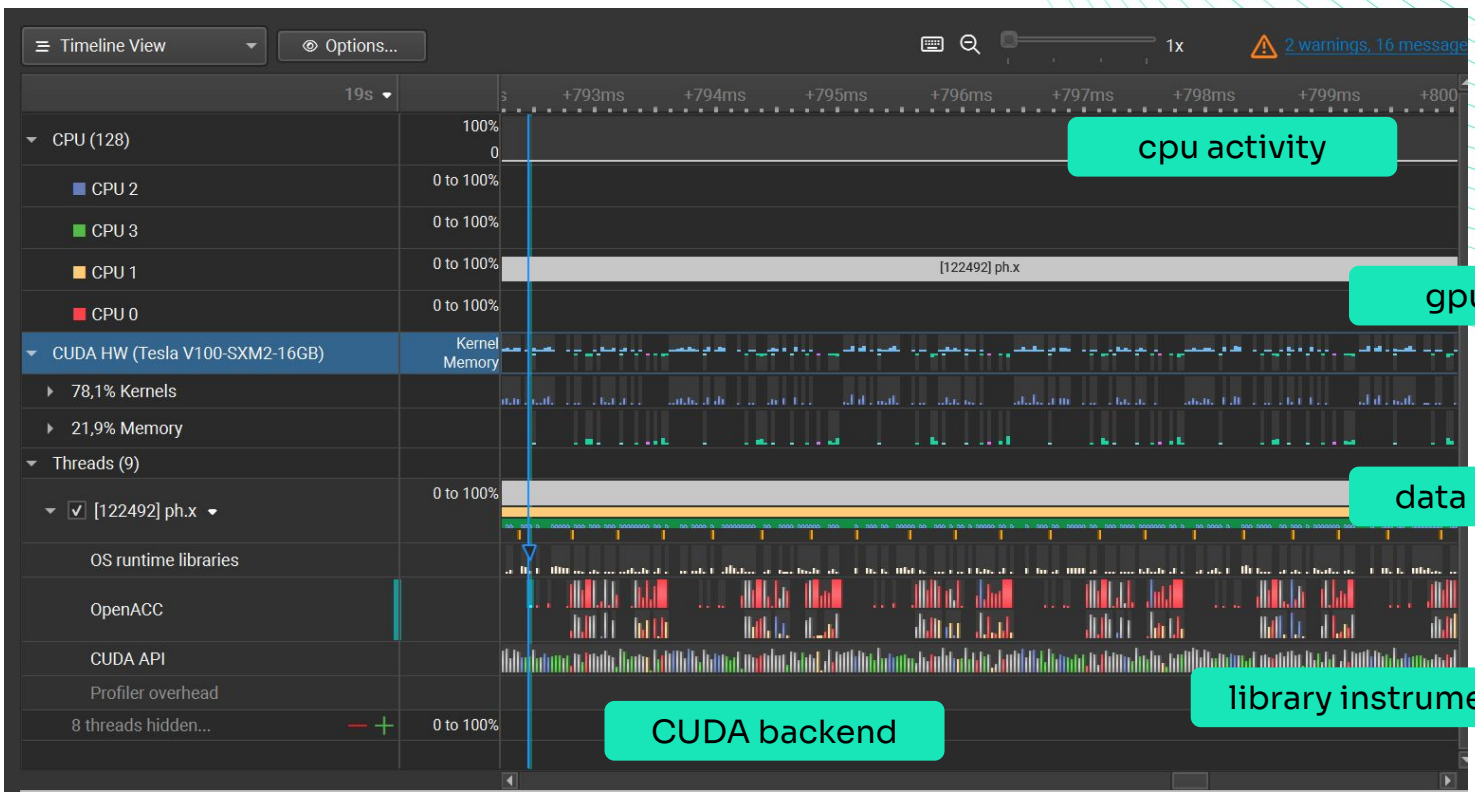
MEASUREMENT

SAMPLING	GPU	TRACES
INSTRUMENTATION	CPU	SUMMARIES

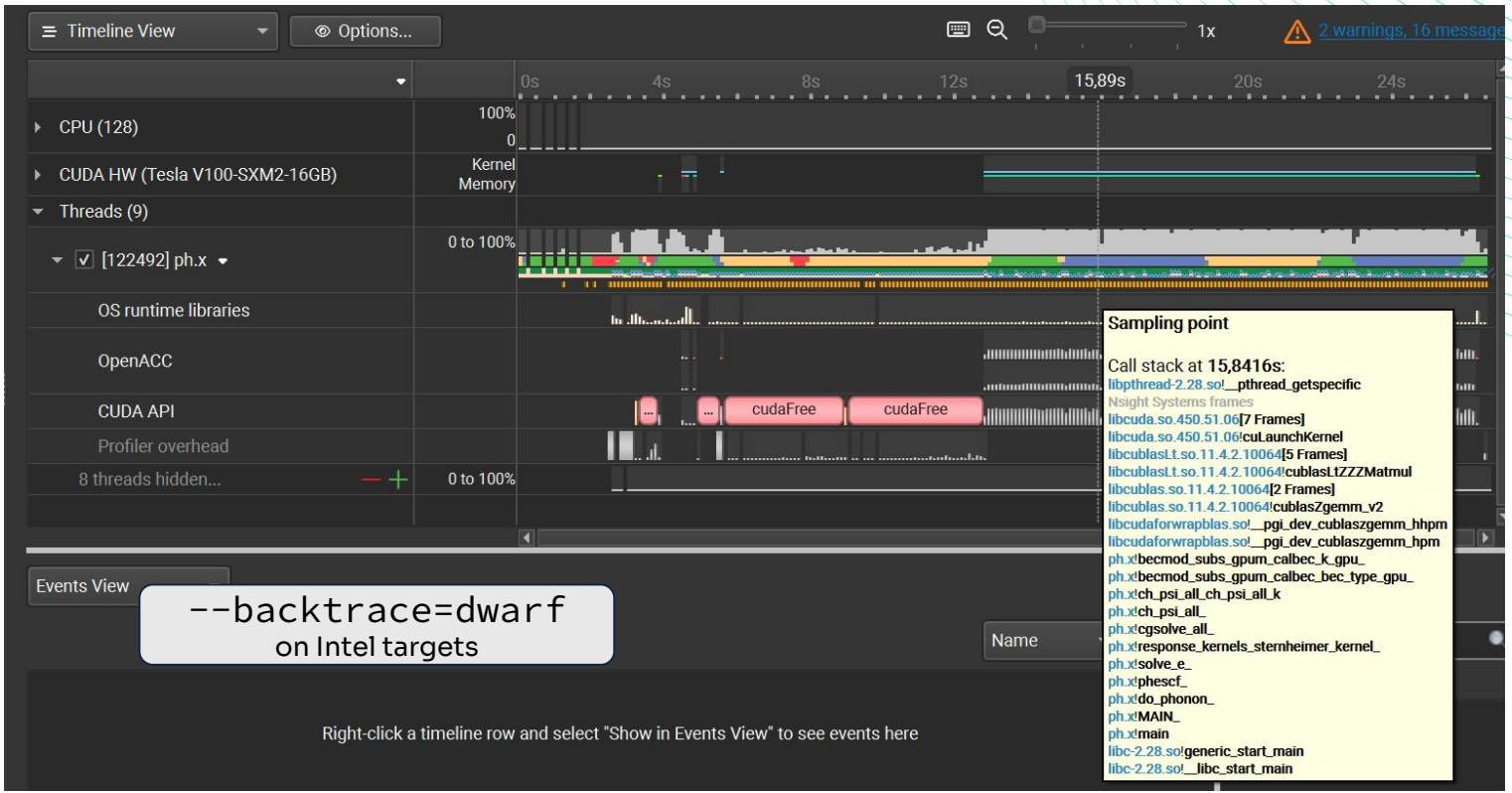
- Instrument CUDA APIs
- Instrument OpenACC regions
- Instrument MPI, OpenMP for the CPU
- Instrument GPU kernels, GPU metrics
- Sample the CPU (usr routines, libraries)
- Network metrics (H2D, D2D, NIC metrics)

Generate the report with CLI (nsys) → `*.nsys-rep` → open the trace with GUI (nsys-ui)

Timeline view



CPU sampling



`--backtrace=dwarf`
on Intel targets

Summaries

Flat View | Native | Process [122492] ph.x (9 of 9 threads)

Filter... | 49.703 samples are used.

Symbol Name	Self, %	Stack, %	Module Name
__libc_start_...	·	74,08	/usr/lib64/power9/libc-2.28.so
generic_star...	·	74,08	/usr/lib64/power9/libc-2.28.so
main	·	74,03	/m100_scratch/userinternal/lbellen1/qe_src/prototipo0/gpuv/build_nonvtx/bin/ph.x
MAIN_	·	74,03	/m100_scratch/userinternal/lbellen1/qe_src/prototipo0/gpuv/build_nonvtx/bin/ph.x
do_phonon_	·	65,88	/m100_scratch/userinternal/lbellen1/qe_src/prototipo0/gpuv/build_nonvtx/bin/ph.x
cgsolve_all_	0,01	50,25	/m100_scratch/userinternal/lbellen1/qe_src/prototipo0/gpuv/build_nonvtx/bin/ph.x
ch_psi_all_	0,03	49,41	/m100_scratch/userinternal/lbellen1/qe_src/prototipo0/gpuv/build_nonvtx/bin/ph.x
response_ke...	0,01	45,22	/m100_scratch/userinternal/lbellen1/qe_src/prototipo0/gpuv/build_nonvtx/bin/ph.x
phqscf_	·	36,51	/m100_scratch/userinternal/lbellen1/qe_src/prototipo0/gpuv/build_nonvtx/bin/ph.x
solve_linter_	·	36,37	/m100_scratch/userinternal/lbellen1/qe_src/prototipo0/gpuv/build_nonvtx/bin/ph.x
phescf_	·	29,01	/m100_scratch/userinternal/lbellen1/qe_src/prototipo0/gpuv/build_nonvtx/bin/ph.x
solve_e_	·	28,42	/m100_scratch/userinternal/lbellen1/qe_src/prototipo0/gpuv/build_nonvtx/bin/ph.x
__clone	·	25,44	/usr/lib64/power9/libc-2.28.so
start_thread	·	25,44	/usr/lib64/power9/libpthread-2.28.so
h_psi_gpu_	·	23,88	/m100_scratch/userinternal/lbellen1/qe_src/prototipo0/gpuv/build_nonvtx/bin/ph.x
h_psi__gpu_	0,01	23,05	/m100_scratch/userinternal/lbellen1/qe_src/prototipo0/gpuv/build_nonvtx/bin/ph.x
0x2000585f	0,10	18,58	/usr/lib64/libcuda.so.450.51.06

Statistics

Stats System View

Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Name
19.0%	137,204 ms	42020	3,265 µs	3,355 µs	2,843 µs	6,621 µs	329 ns	void composite_2way_fft<20u, 4u, 32u, (padding_t)0, (twidd
18.0%	130,783 ms	22556	5,798 µs	5,756 µs	5,499 µs	14,618 µs	292 ns	volta_zgemm_32x32_nn
17.0%	123,229 ms	16917	7,284 µs	7,356 µs	6,045 µs	12,861 µs	374 ns	void zgemm_largek_warp<true, false, true, false, 3, 3, 4, 3, 2,
10.0%	71,216 ms	21010	3,389 µs	3,453 µs	2,810 µs	6,972 µs	457 ns	void composite_2way_fft<20u, 4u, 16u, (padding_t)0, (twidd
2.0%	19,550 ms	8285	2,359 µs	2,748 µs	1,435 µs	3,357 µs	559 ns	fft_scalar_cufft_cfft3d_gpu_586_gpu
2.0%	18,334 ms	12010	1,526 µs	1,531 µs	1,499 µs	2,109 µs	17 ns	add_vuspsi_k_gpu_248_gpu
2.0%	17,736 ms	16917	1,048 µs	1,052 µs	986 ns	1,468 µs	23 ns	void scal_kernel<double2, double2, 1, true, 5, 4, 4, 4>(cublas
2.0%	17,436 ms	12010	1,451 µs	1,438 µs	1,275 µs	1,980 µs	52 ns	dp_dev_memcpy_c2d_770_gpu
2.0%	15,034 ms	6005	2,503 µs	2,590 µs	1,466 µs	3,037 µs	285 ns	vloc_psi_k_gpu_464_gpu
2.0%	14,024 ms	6005	2,335 µs	2,363 µs	2,042 µs	3,644 µs	91 ns	vloc_psi_k_gpu_456_gpu
1.0%	13,388 ms	6005	2,229 µs	2,236 µs	2,013 µs	3,003 µs	57 ns	vloc_psi_k_gpu_477_gpu
1.0%	10,685 ms	6005	1,779 µs	1,787 µs	1,627 µs	3,004 µs	49 ns	h_psi__gpu_158_gpu
1.0%	9,865 ms	6005	1,642 µs	1,658 µs	1,531 µs	2,237 µs	33 ns	ch_psi_all_132_gpu
1.0%	9,763 ms	6005	1,625 µs	1,628 µs	1,531 µs	2,172 µs	30 ns	ch_psi_all_k_266_gpu

CLI : nsys command

`nsys [command_switch] [optional command_switch_options][application] [optional application_options`

Command switches

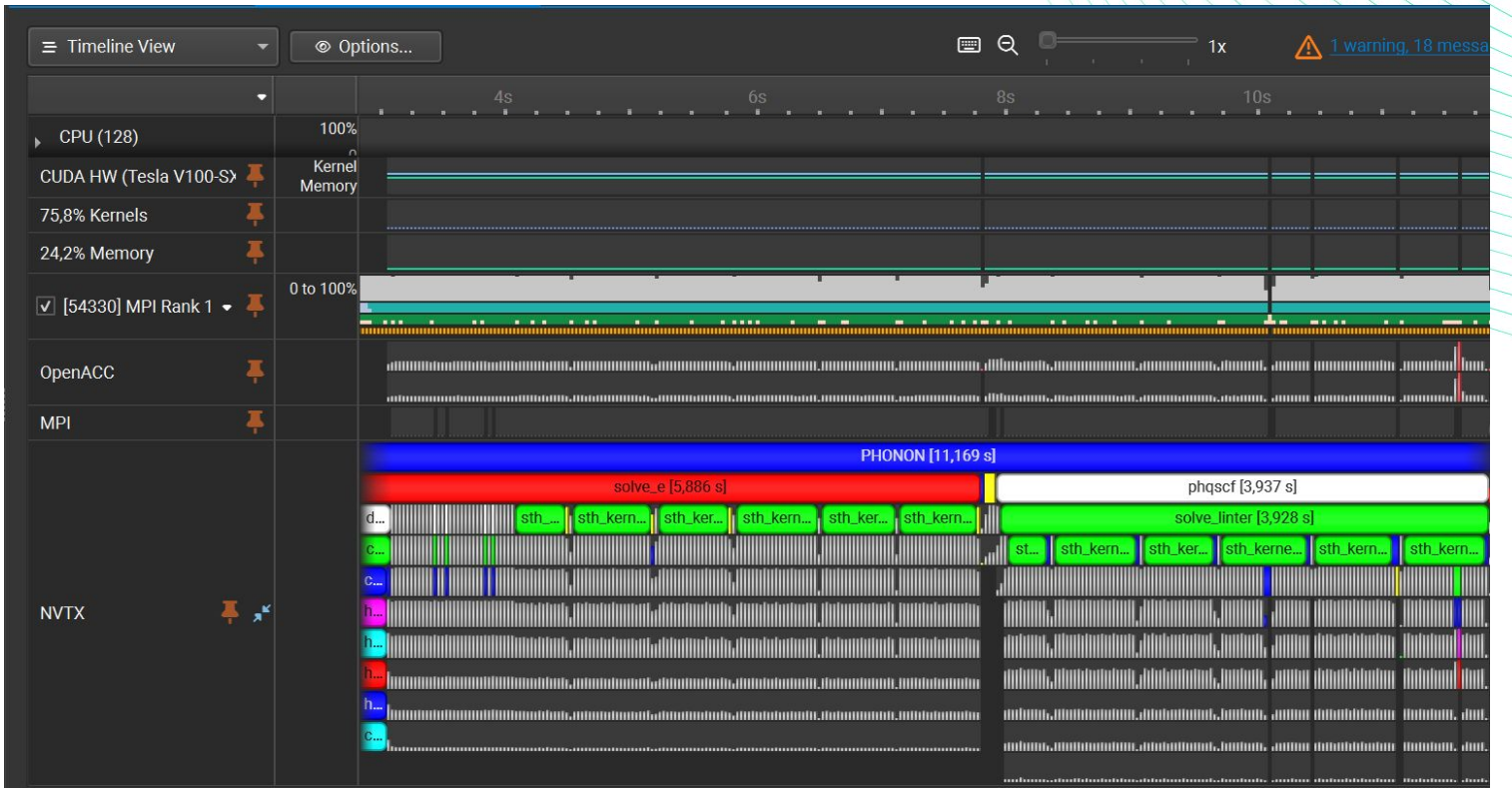
- **profile** — All-in-one , needed for concurrent profiling sessions
- start - launch - cancel - shutdown - stop — Interactive mode
- export - stats - analyze — Postprocessing to export / textual summaries

Command switch options

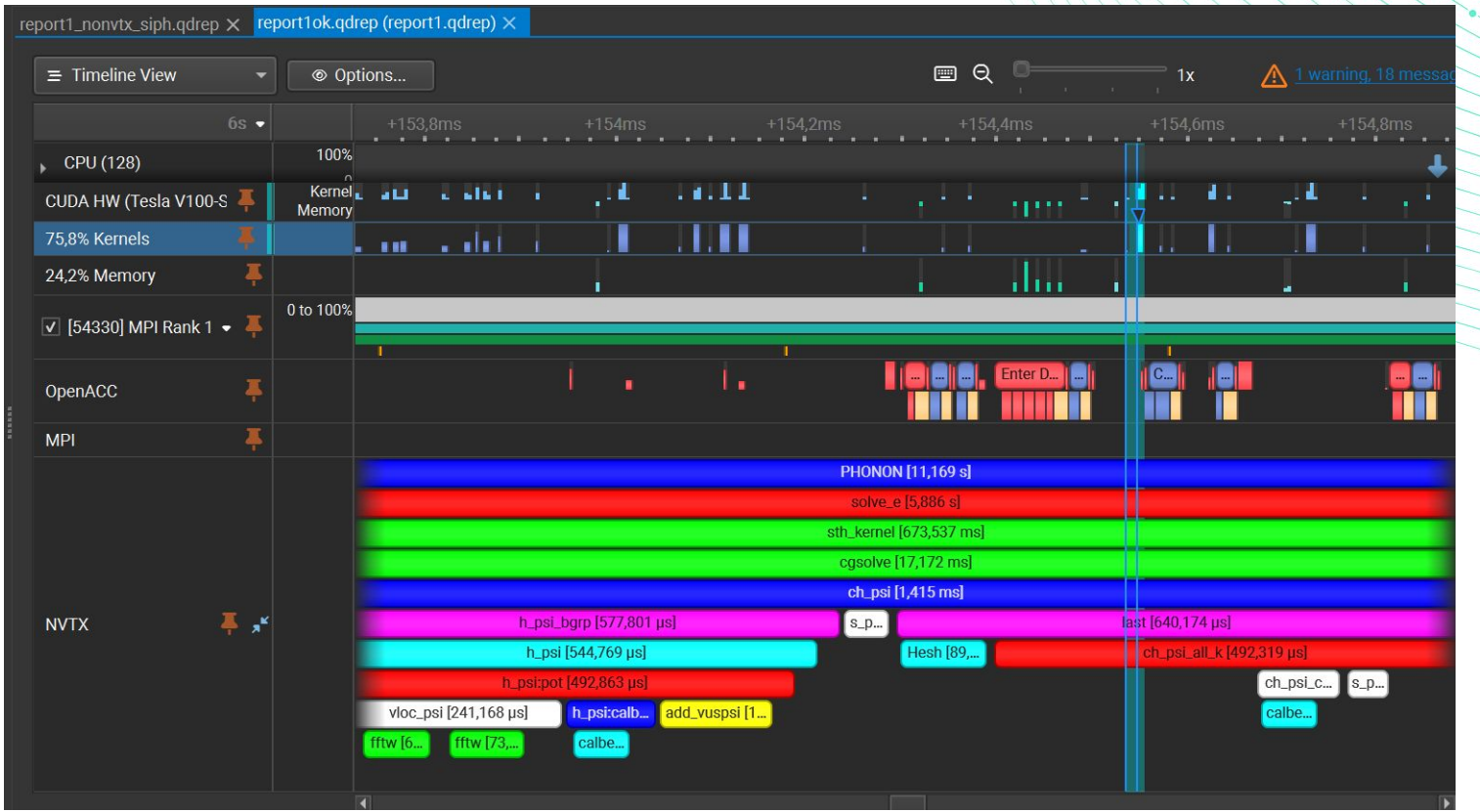
- **--trace=**[nvtx,cuda,osrt],mpi,openmp,openacc,cublas,cusolver,... — Events to trace
- --cuda-memory-usage — Collect GPU memory usage
- --nic-metrics — Network bandwidth

Example: `nsys profile --trace=openacc,cuda,nvtx cfd.exe → report1.nsys-rep`

NVTX ranges


Timeline View
Options...
1x
1 warning, 18 messa
4s
6s
8s
10s
CPU (128)
100%
0
CUDA HW (Tesla V100-S
Kernel
Memory
75,8% Kernels
24,2% Memory
[54330] MPI Rank 1
0 to 100%
OpenACC
MPI
NVTX
PHONON [11,169 s]
solve_e [5,886 s]
phqscf [3,937 s]
solve_linter [3,928 s]
sth_kern...
sth_ker...
sth_kerne...

NVTX ranges


report1_nonvtx_siph.qdrep
report1ok.qdrep (report1.qdrep)
Timeline View
Options...
1x
1 warning, 18 messag
6s
+153,8ms
+154ms
+154,2ms
+154,4ms
+154,6ms
+154,8ms
CPU (128)
100%
0
CUDA HW (Tesla V100-S
Kernel
Memory
75,8% Kernels
24,2% Memory
[54330] MPI Rank 1
0 to 100%
OpenACC
Enter D...
C...
MPI
NVTX
PHONON [11,169 s]
solve_e [5,886 s]
sth_kernel [673,537 ms]
cgsolve [17,172 ms]
ch_psi [1,415 ms]
h_psi_bgrp [577,801 µs]
s_p...
last [640,174 µs]
h_psi [544,769 µs]
Hesh [89,...
ch_psi_all_k [492,319 µs]
h_psi:pot [492,863 µs]
ch_psi_c...
s_p...
vloc_psi [241,168 µs]
h_psi:calb...
add_vuspsi [1...
calbe...
fftw [6...
fftw [73,...
calbe...

NVTX ranges

NVTX library (nvToolsExt.h) can be used to enhance trace readability:

- C-based API to annotate events and code ranges, to be visualized in the Nsight System timeline
- Limited overhead when the tool is not attached to the application

Functionalities

- **NVTX Markers** → annotate events occurring at a specific time
- **NVTX Ranges** → annotate timespan of code regions

Events

- associated to message (ASCII, Unicode) → **A, W** variant of function calls
- associated to structure with attributes → **Ex** variant of function calls

```
//Set to default

nvtxEventAttributes_t eventAttrib = {0};

//Declare version and size

eventAttrib.version = NVTX_VERSION;
eventAttrib.size = NVTX_EVENT_ATTRIB_STRUCT_SIZE;

// Message type and message
// // ASCII

eventAttrib.messageType = NVTX_MESSAGE_TYPE_ASCII;
eventAttrib.message.ascii = __FUNCTION__ ":ascii";
// //UNICODE

eventAttrib2.messageType = NVTX_MESSAGE_TYPE_UNICODE;
eventAttrib2.message.unicode = __FUNCTIONW__ L
":unicode \u2603 snowman";

// Color type and color

eventAttrib.colorType = NVTX_COLOR_ARGB;
eventAttrib.color = COLOR_YELLOW;
```

NVTX ranges

NVTX library (nvToolsExt.h) can be used to enhance trace readability:

- C-based API to annotate events and code ranges, to be visualized in the Nsight System timeline
- Limited overhead when the tool is not attached to the application

Functionalities

- **NVTX Markers** → annotate events occurring at a specific time
- **NVTX Ranges** → annotate timespan of code regions

Events

- associated to message (ASCII, Unicode) → **A, W** variant of function calls
- associated to structure with attributes → **Ex** variant of function calls

```
//Set to default

nvtxEventAttributes_t eventAttrib = {0};

//Declare version and size

eventAttrib.version = NVTX_VERSION;
eventAttrib.size = NVTX_EVENT_ATTRIB_STRUCT_SIZE;

// Message type and message
// // ASCII

eventAttrib.messageType = NVTX_MESSAGE_TYPE_ASCII;
eventAttrib.message.ascii = __FUNCTION__ ":ascii";
// //UNICODE

eventAttrib2.messageType = NVTX_MESSAGE_TYPE_UNICODE;
eventAttrib2.message.unicode = __FUNCTIONW__ L
":unicode \u2603 snowman";

// Color type and color

eventAttrib.colorType = NVTX_COLOR_ARGB;
eventAttrib.color = COLOR_YELLOW;
```

NVTX ranges

NVTX library (nvToolsExt.h) can be used to enhance trace readability:

- C-based API to annotate events and code ranges, to be visualized in the Nsight System timeline
- Limited overhead when the tool is not attached to the application

Functionalities

- **NVTX Markers** → annotate events occurring at a specific time
- **NVTX Ranges** → annotate timespan of code regions

Events

- associated to message (ASCII, Unicode) → **A, W** variant of function calls
- associated to structure with attributes → **Ex** variant of function calls

```
//Set to default

nvtxEventAttributes_t eventAttrib = {0};

//Declare version and size

eventAttrib.version = NVTX_VERSION;
eventAttrib.size = NVTX_EVENT_ATTRIB_STRUCT_SIZE;

// Message type and message
// // ASCII

eventAttrib.messageType = NVTX_MESSAGE_TYPE_ASCII;
eventAttrib.message.ascii = __FUNCTION__ ":ascii";
// //UNICODE

eventAttrib2.messageType = NVTX_MESSAGE_TYPE_UNICODE;
eventAttrib2.message.unicode = __FUNCTIONW__ L
":unicode \u2603 snowman";

// Color type and color

eventAttrib.colorType = NVTX_COLOR_ARGB;
eventAttrib.color = COLOR_YELLOW;
```

NVTX ranges

Markers (events at a specific time)

```
nvtxMarkA(__FUNCTION__ ":nvtxMarkA");
nvtxMarkW(__FUNCTIONW__ L":nvtxMarkW");
nvtxMarkEx(&eventAttrib);
```

Ranges (nested time ranges occurring on a CPU thread)

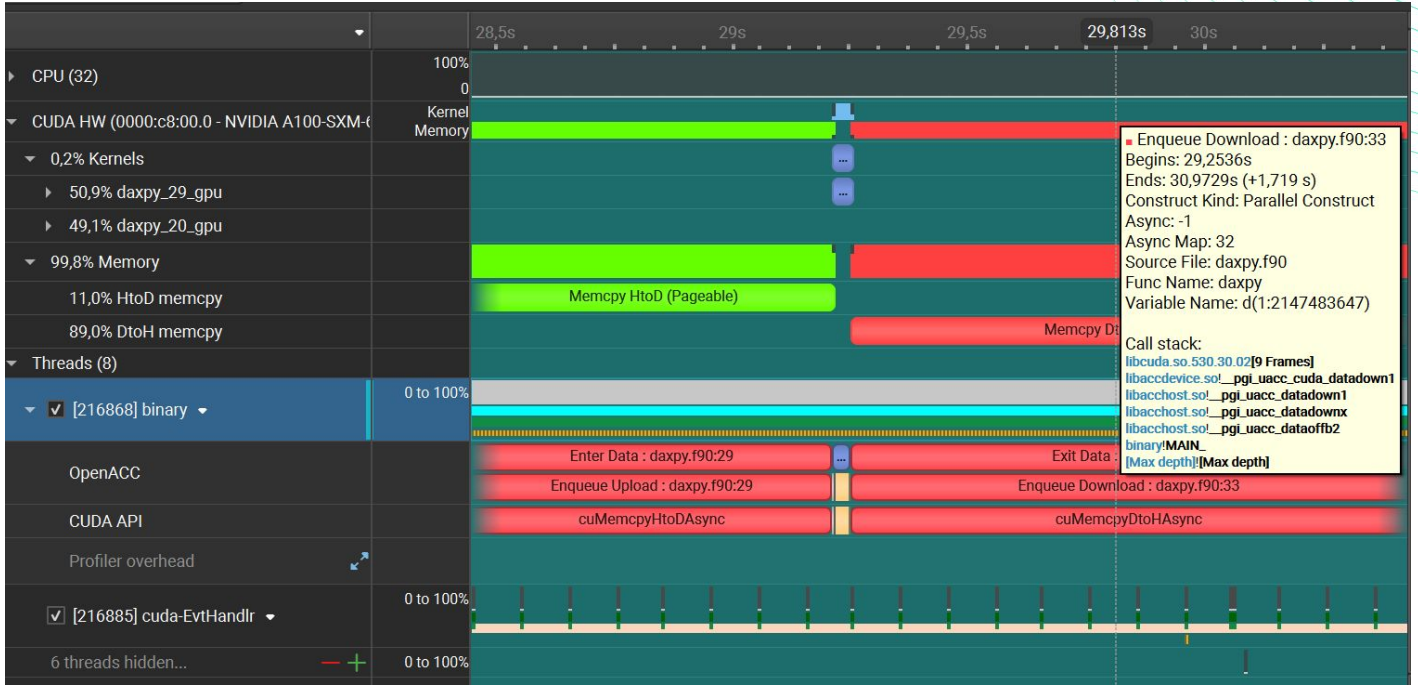
```
start:  nvtxRangePushEx
        nvtxRangePushA
        nvtxRangePushW
end :  nvtxRangePop
```

```
// for message-only events

nvtxRangePushA(__FUNCTION__ ":nvtxRangePushA");
[... code here ...]
nvtxRangePop();

nvtxRangePushW(__FUNCTIONW__
L":nvtxRangePushW");
[... code here ...]
nvtxRangePop();

// for structured events

nvtxRangePushEx(&eventAttrib);
[... code here ...]
nvtxRangePop();
```

CUDAFortran: interfaces provided by nvtx module (subroutines nvtxStartRange, nvtxEndRange)

DAXPY profiled

```
nvfortran -fast -acc -gpu=cc80,cuda12.3 -Minfo=accel -lnvhpcwrapnvtx -o binary daxpy.f90
```



DAXPY profiled

```
nvfortran -fast -acc -gpu=cc80,cuda12.3 -Minfo=accel -lnvhpcwrapnvtx -o binary daxpy.f90
```

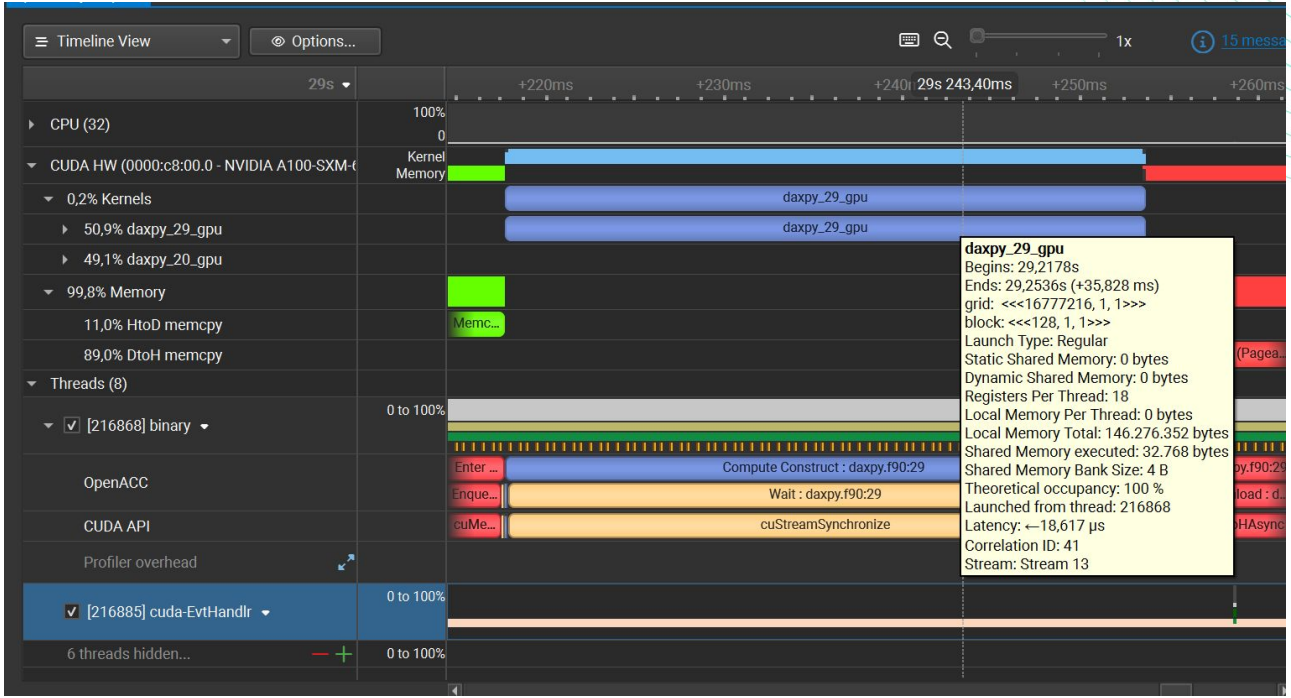
```
call nvtxstartRange("initialize",1)
!$acc parallel loop
do i = 1, N
  D(i) = 0
  X(i) = 1
  Y(i) = 2
end do
call nvtxEndRange()

call nvtxStartRange("daxpy",2)
!$acc parallel loop
do i = 1, N
   D(i) = A* X(i) + Y(i)
end do
call nvtxEndRange()
```

Data movements

```
nvfortran -fast -acc -gpu=cc80,cuda12.3 -Minfo=accel -lnvhpcwrapnvtx -o binary daxpy.f90
```

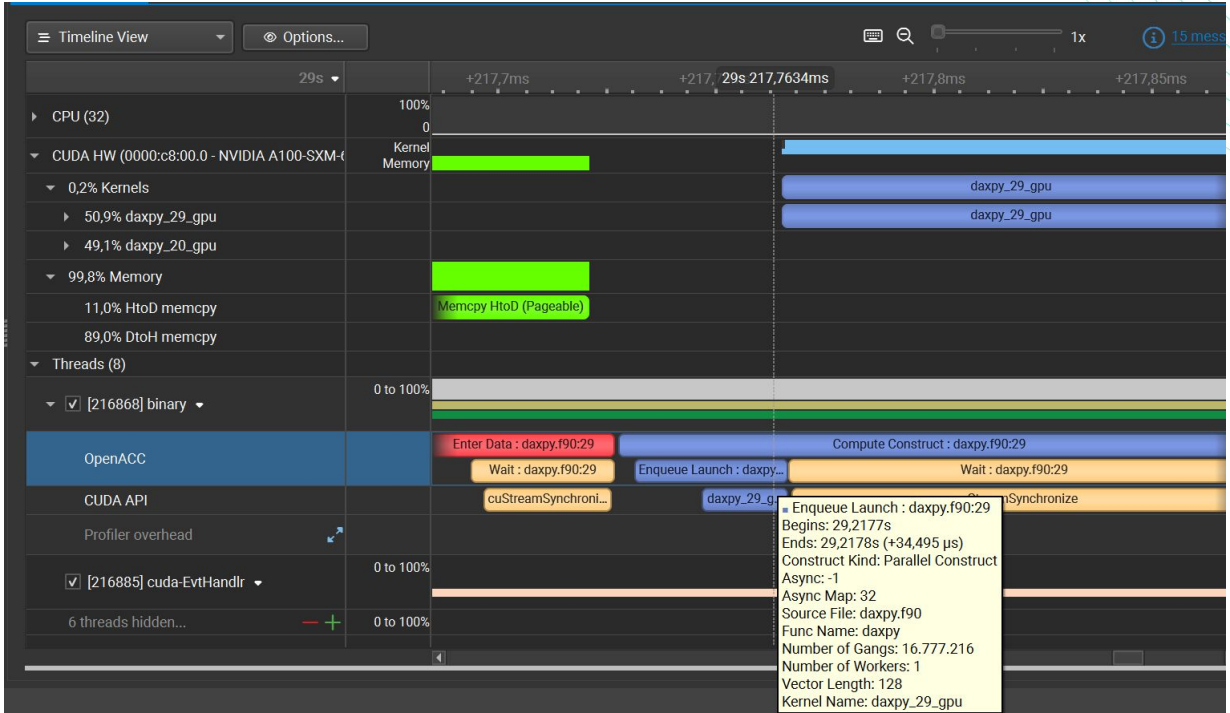
GPU kernels

```
nvfortran -fast -acc -gpu=cc80,cuda12.3 -Minfo=accel -lnvhpcwrapnvtx -o binary daxpy.f90
```



Kernel launches

```
nvfortran -fast -acc -gpu=cc80,cuda12.3 -Minfo=accel -lnvhpcwrapnvtx -o binary daxpy.f90
```



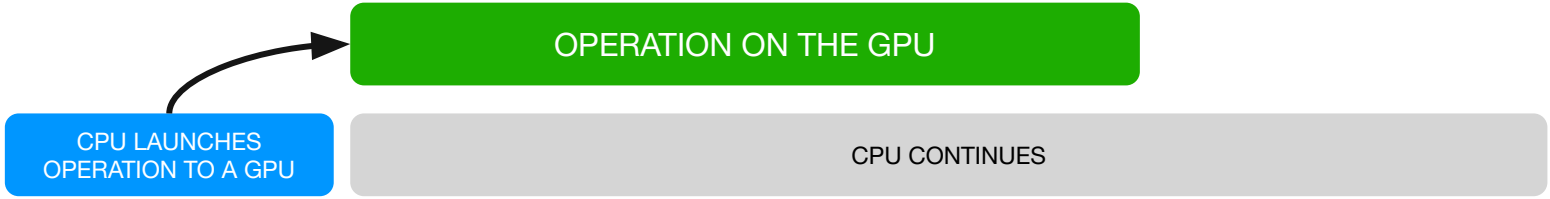
Patterns in OpenACC traces

SYNC : CPU waits for the operation on the GPU to be over

Patterns in OpenACC traces

SYNC : CPU waits for the operation on the GPU to be over

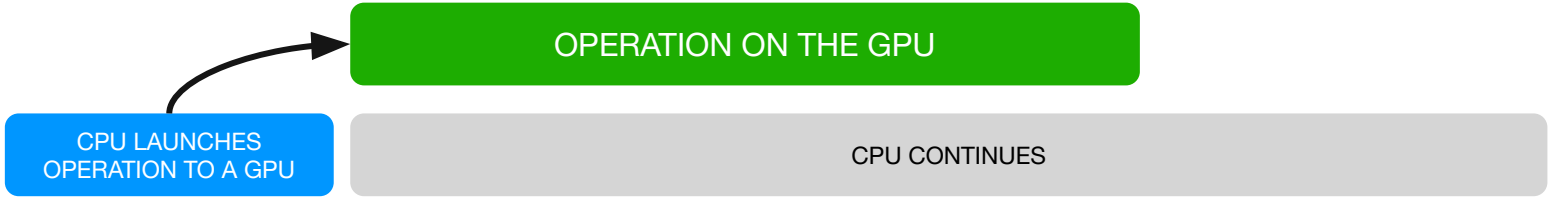
ASYNC : CPU does not wait for the operation on the GPU to be over



Patterns in OpenACC traces

SYNC : CPU waits for the operation on the GPU to be over

ASYNC : CPU does not wait for the operation on the GPU to be over



Patterns for compute directives

By default, OpenACC queues operation on the default stream

Data directives, parallel and kernels impose sync between CPU and GPU

Parallel and kernels embody an implicit data region

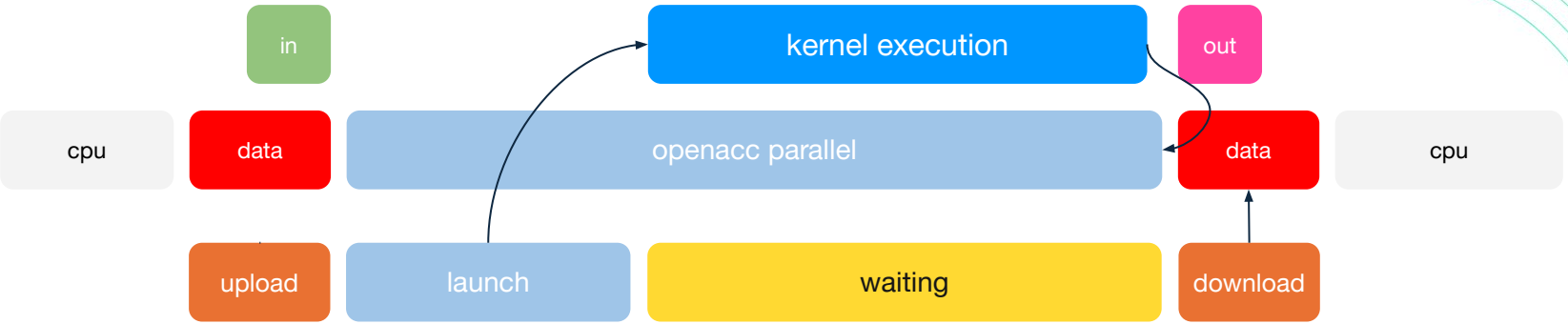
```
!$acc parallel loop [copyout(a)]
do i = 1, N
 a(i) = 0
end do
```

Patterns for compute directives

By default, OpenACC queues operation on the default stream

Data directives, parallel and kernels impose sync between CPU and GPU

Parallel and kernels embody an implicit data region



Patterns for compute and data

What if parallel/kernels is in a data region?

If parallel/kernel is in the same routine of the **structured data region**, the runtime knows that the data is already on the GPU

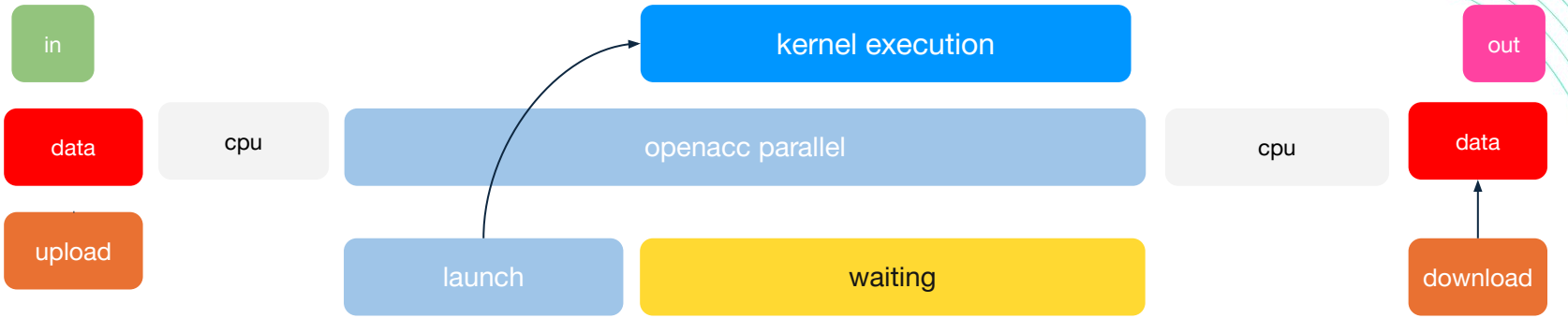
Remind that `enter data` does not open a data region!

```
subroutine myloop()
<...>
!$acc data copyout(a)
!$acc parallel loop
do i = 1, N
 a(i) = 0
end do
!$acc end data
<...>
end subroutine
```

Patterns for compute and data

Unless needed to copy some missing variables, the implicit data region is not opened (because already opened!)

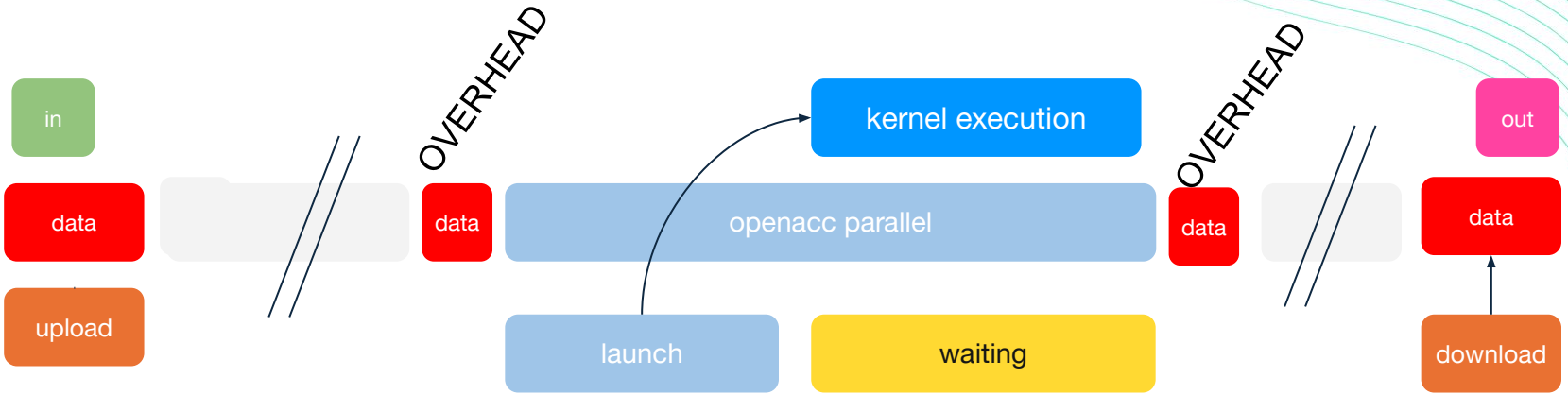
If the compute region is in the same routine of the data region, the runtime will not check for variables on the GPU



Patterns for compute and data

If the compute directive is not in the same routine of the data region, the runtime will check anyways if the data is on the GPU or not

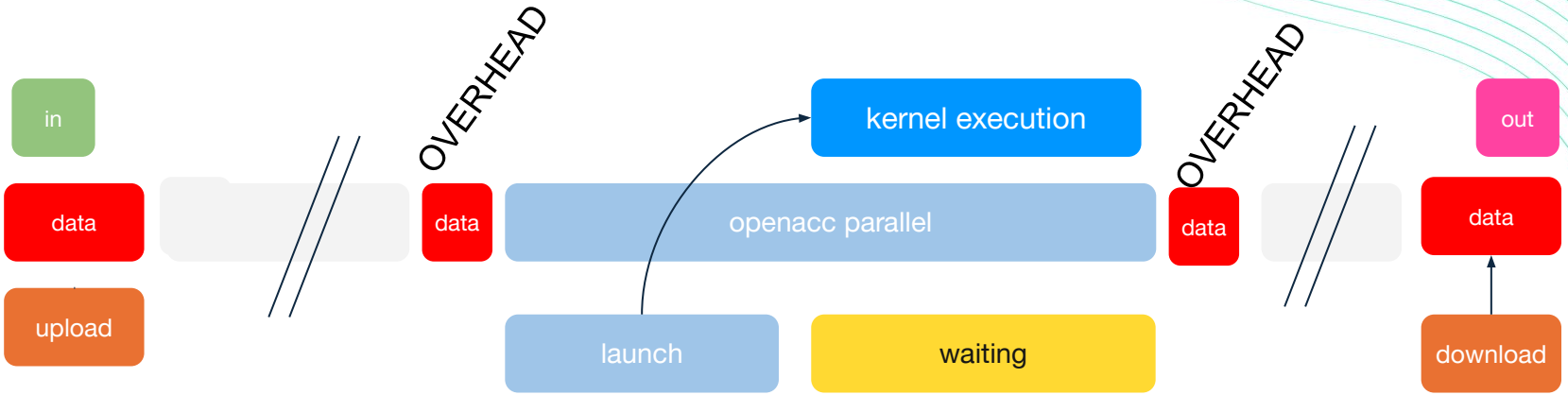
This extra check does not trigger an actual data movement, but it is an overhead



Patterns for compute and data

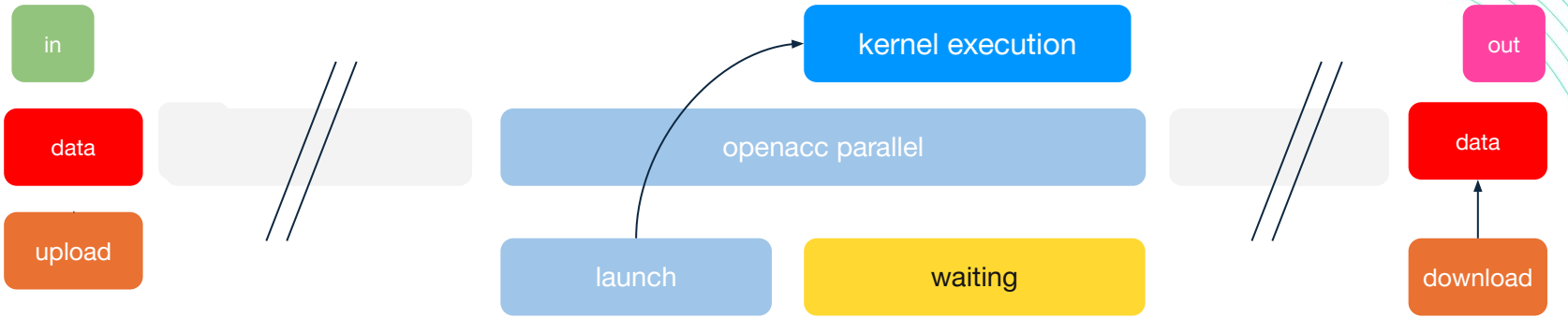
This extra check does not trigger an actual data movement, but it is an overhead

Be careful when using **PRESENT** clause, it forces this extra check, might add an overhead



Patterns for compute and data

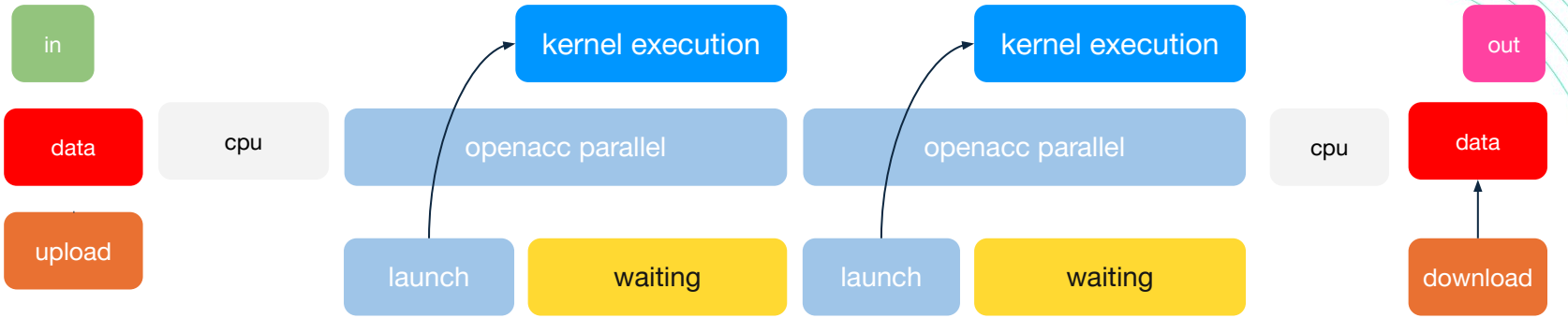
This extra check can be avoided only with the `declare` directive on a global variables



Patterns for compute and data

Unless needed to copy some missing variables, the implicit data region is not opened (because already opened!)

We can open a single data region for multiple kernels



Kernel launches

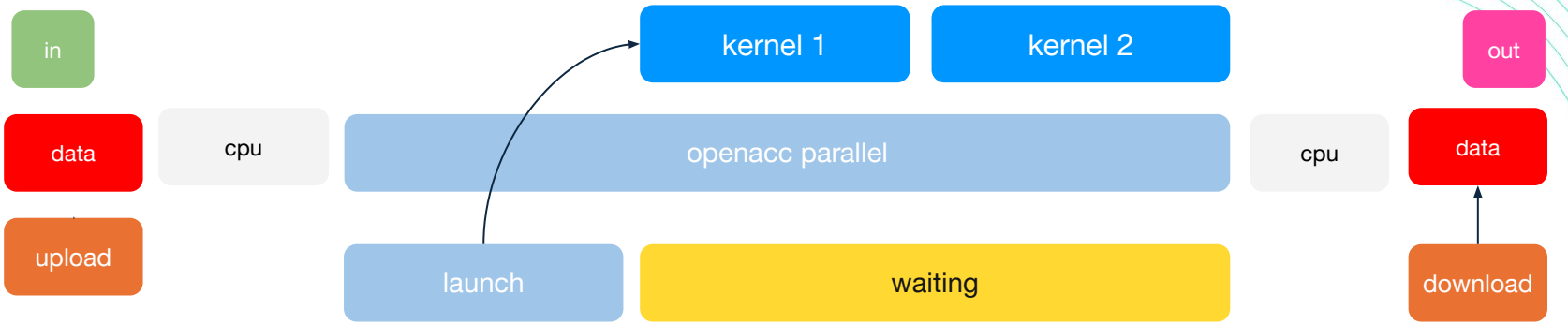
What if a single parallel encloses multiple loops

```
!$acc data copyout(a,b)
!$acc parallel

!$acc loop
do i = 1, N
 a(i) = 0
end do
!$acc loop
do i = 1, N
 b(i) = 0
end do

!$acc end parallel
!$acc end data
```

Kernel launches

There will be one OpenACC compute region, <u>one single launch</u> and multiple kernels

Avoids multiple kernel launches (reduces overhead)



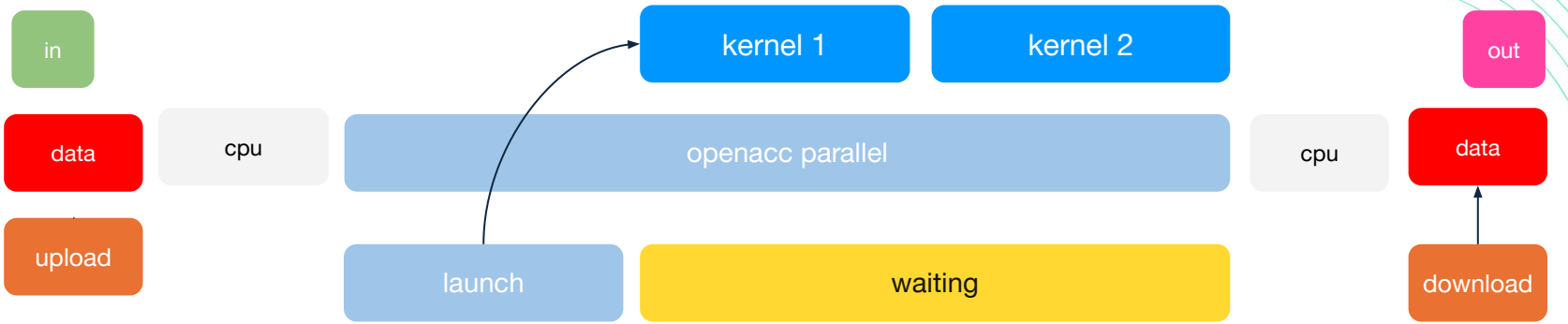
Kernel launches

There will be one OpenACC compute region, <u>one single launch</u> and multiple kernels

Avoids multiple kernel launches (reduces overhead)

! Parallel: Gangs do not sync: if one gang has finished relative job in kernel 1, continues in kernel 2

! Kernels: imposes a sync barrier



Parallel vs kernels

Careful if gangs access different data locations in loops

```
!$acc parallel
!$acc loop
do i = 1, N
    D(i) = 0
    X(i) = 1
    Y(i) = 2
end do

!$acc loop
do i = 1, N
     D(i) = A*X(i)+Y(i)
end do
!$acc end parallel
```

```
!$acc parallel
!$acc loop
do i = 1, N
    D(i) = 0
    X(i) = 1
    Y(i) = 2
end do

!$acc loop
do i = 1, N-1
     D(i) = A*X(i+1)+Y(i+1)
end do
!$acc end parallel
```

```
!$acc kernels
do i = 1, N
    D(i) = 0
    X(i) = 1
    Y(i) = 2
end do

[implicit wait]

do i = 1, N-1
     D(i) = A*X(i+1)+Y(i+1)
end do
!$acc end kernels
```

Kernels or parallel?

Kernels

- Compiler decides what to parallelize with direction from user
- Compiler guarantees correctness
- Can cover multiple loop nests

Parallel

- Programmer decides what to parallelize and communicates that to the compiler
- Programmer guarantees correctness
- Must decorate each loop nest

When fully optimized, both will give similar performance.

Summaries

```
nsys stats report1.nsys-rep
```

Time	Total Time	Count	Avg	Med	Min	Max	StdDev	Operation
89.0%	27,115 s	4	6,779 s	8,122 s	1,719 s	9,152 s	3,410 s	[CUDA memcpy Device-to-Host]
11.0%	3,359 s	2	1,680 s	1,680 s	1,677 s	1,682 s	3,730 ms	[CUDA memcpy Host-to-Device]

Total	Count	Avg	Med	Min	Max	StdDev	Operation
64,00 GiB	4	16,00 GiB	16,00 GiB	16,00 GiB	16,00 GiB	0 B	[CUDA memcpy Device-to-Host]
32,00 GiB	2	16,00 GiB	16,00 GiB	16,00 GiB	16,00 GiB	0 B	[CUDA memcpy Host-to-Device]

Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Category	Operation
88.0%	27,115 s	4	6,779 s	8,122 s	1,719 s	9,152 s	3,410 s	MEMORY_OPER	[CUDA memcpy Device-to-Host]
11.0%	3,359 s	2	1,680 s	1,680 s	1,677 s	1,682 s	3,730 ms	MEMORY_OPER	[CUDA memcpy Host-to-Device]
0.0%	35,828 ms	1	35,828 ms	35,828 ms	35,828 ms	35,828 ms	0 ns	CUDA_KERNEL	daxpy_29_gpu
0.0%	34,609 ms	1	34,609 ms	34,609 ms	34,609 ms	34,609 ms	0 ns	CUDA_KERNEL	daxpy_20_gpu

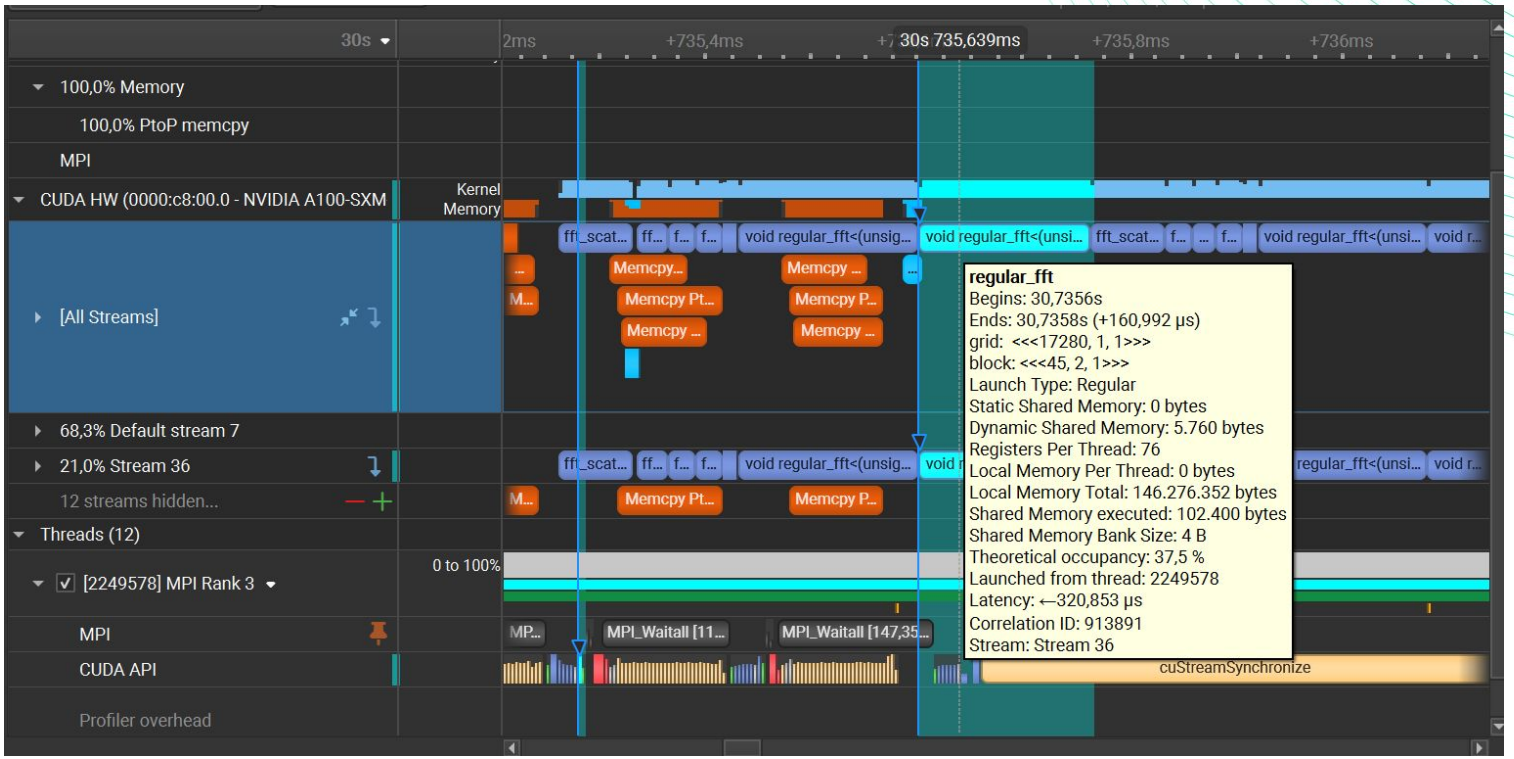
OpenACC runtime

Timeline View | Options... | 1x | 1 warning, 17 message

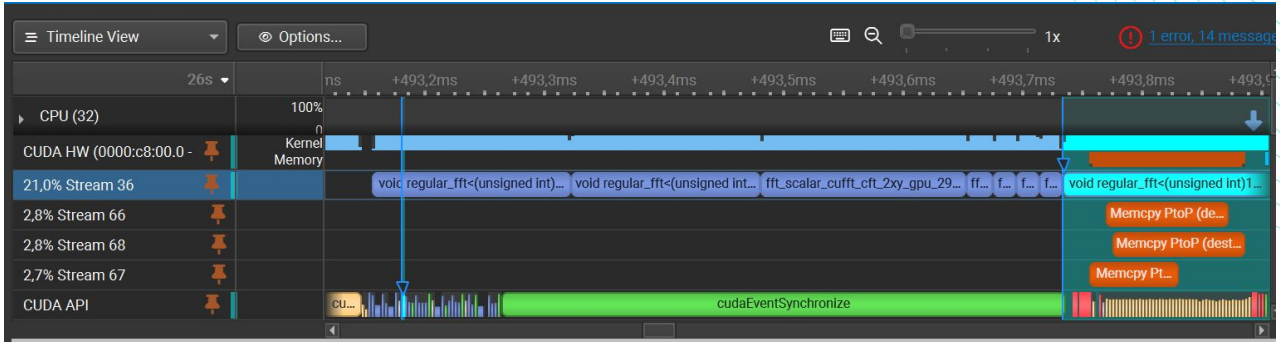
Stats System View

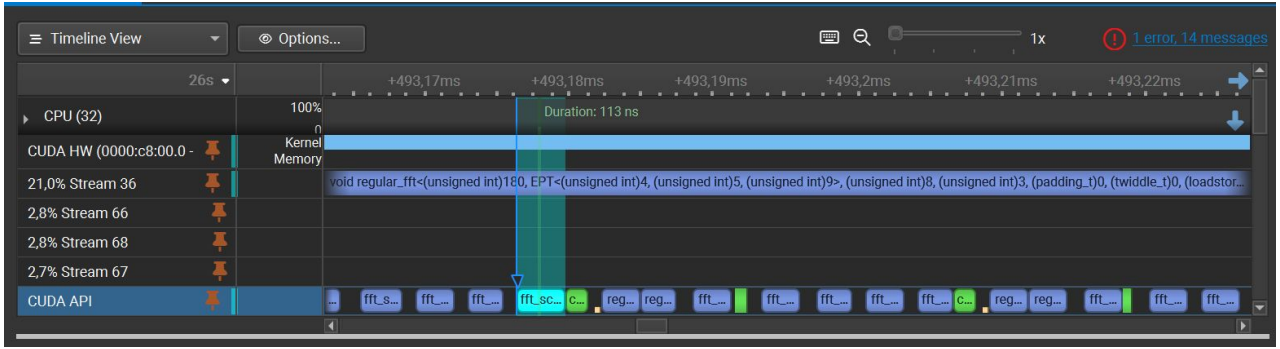
Time	Total Time	Num Calls	Avg	Med	Min	Max	StdDev	Name
41.0%	24,963 s	1	24,963 s	24,963 s	24,963 s	24,963 s	0 ns	Exit Data@daxpy.f90:20
41.0%	24,963 s	3	8,321 s	8,041 s	7,832 s	9,091 s	674,577 ms	Enqueue Download@daxpy.f90:26
5.0%	3,250 s	1	3,250 s	3,250 s	3,250 s	3,250 s	0 ns	Enter Data@daxpy.f90:29
5.0%	3,250 s	2	1,625 s	1,625 s	1,624 s	1,626 s	2,050 ms	Enqueue Upload@daxpy.f90:29
2.0%	1,710 s	1	1,710 s	1,710 s	1,710 s	1,710 s	0 ns	Exit Data@daxpy.f90:29
2.0%	1,710 s	1	1,710 s	1,710 s	1,710 s	1,710 s	0 ns	Enqueue Download@daxpy.f90:33
0.0%	37,751 ms	1	37,751 ms	37,751 ms	37,751 ms	37,751 ms	0 ns	Enter Data@daxpy.f90:20
0.0%	35,878 ms	1	35,878 ms	35,878 ms	35,878 ms	35,878 ms	0 ns	Compute Construct@daxpy.f90:29
0.0%	35,872 ms	2	17,936 ms	17,936 ms	31,261 µs	35,840 ms	25,321 ms	Wait@daxpy.f90:29
0.0%	34,669 ms	1	34,669 ms	34,669 ms	34,669 ms	34,669 ms	0 ns	Compute Construct@daxpy.f90:20
0.0%	34,635 ms	2	17,317 ms	17,317 ms	11,136 µs	34,623 ms	24,475 ms	Wait@daxpy.f90:20
0.0%	103,849 µs	1	103,849 µs	103,849 µs	103,849 µs	103,849 µs	0 ns	Device Init@daxpy.f90:20
0.0%	35,866 µs	1	35,866 µs	35,866 µs	35,866 µs	35,866 µs	0 ns	Enqueue Launch@daxpy.f90:20
0.0%	33,241 µs	1	33,241 µs	33,241 µs	33,241 µs	33,241 µs	0 ns	Enqueue Launch@daxpy.f90:29
0.0%	5,756 µs	1	5,756 µs	5,756 µs	5,756 µs	5,756 µs	0 ns	Wait@daxpy.f90:26
0.0%	4,749 µs	1	4,749 µs	4,749 µs	4,749 µs	4,749 µs	0 ns	Wait@daxpy.f90:33
0.0%	0 ns	3	0 ns	0 ns	0 ns	0 ns	0 ns	Alloc@daxpy.f90:20
0.0%	0 ns	3	0 ns	0 ns	0 ns	0 ns	0 ns	Create@daxpy.f90:20
0.0%	0 ns	3	0 ns	0 ns	0 ns	0 ns	0 ns	Create@daxpy.f90:29
0.0%	0 ns	3	0 ns	0 ns	0 ns	0 ns	0 ns	Delete@daxpy.f90:26
0.0%	0 ns	3	0 ns	0 ns	0 ns	0 ns	0 ns	Delete@daxpy.f90:33

NVIDIA libraries



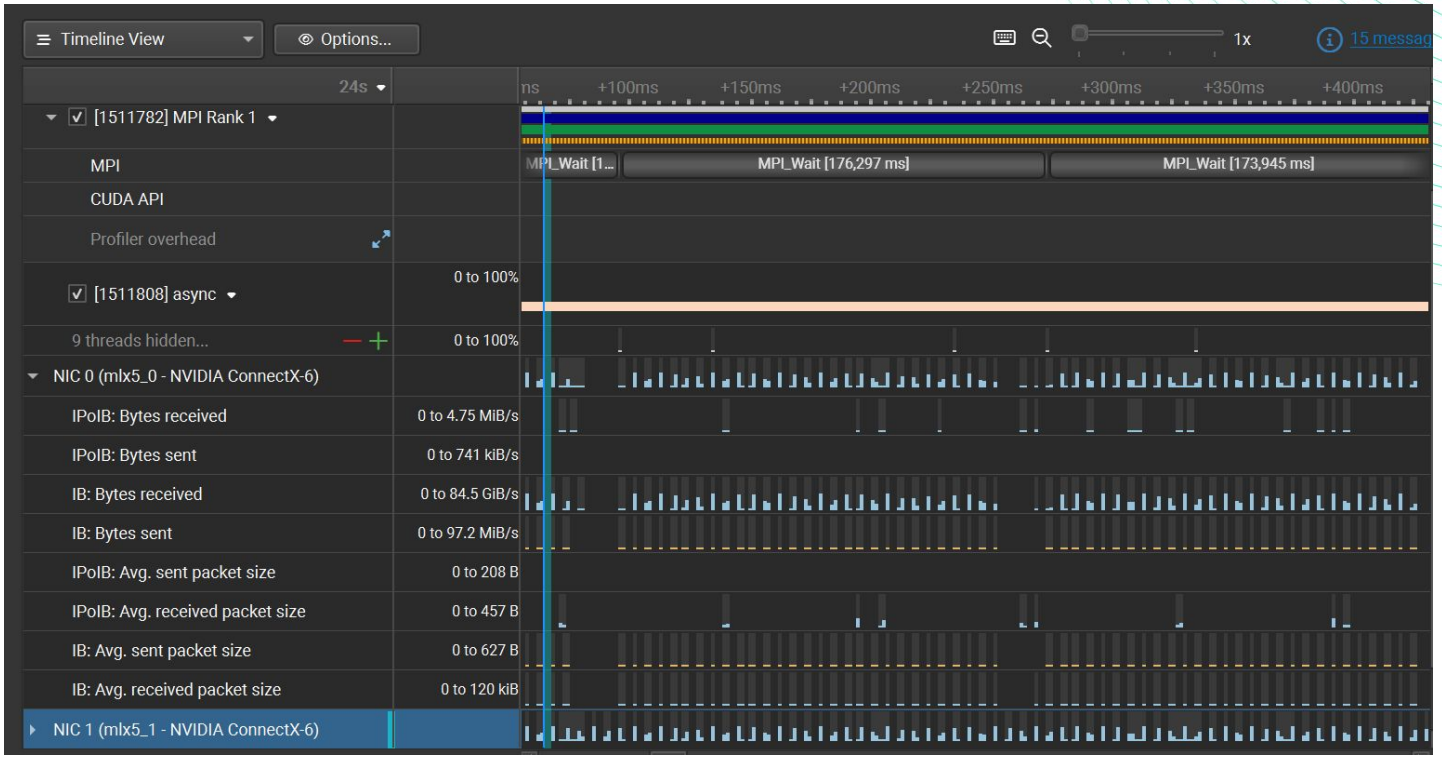
Asynchronous and overlap



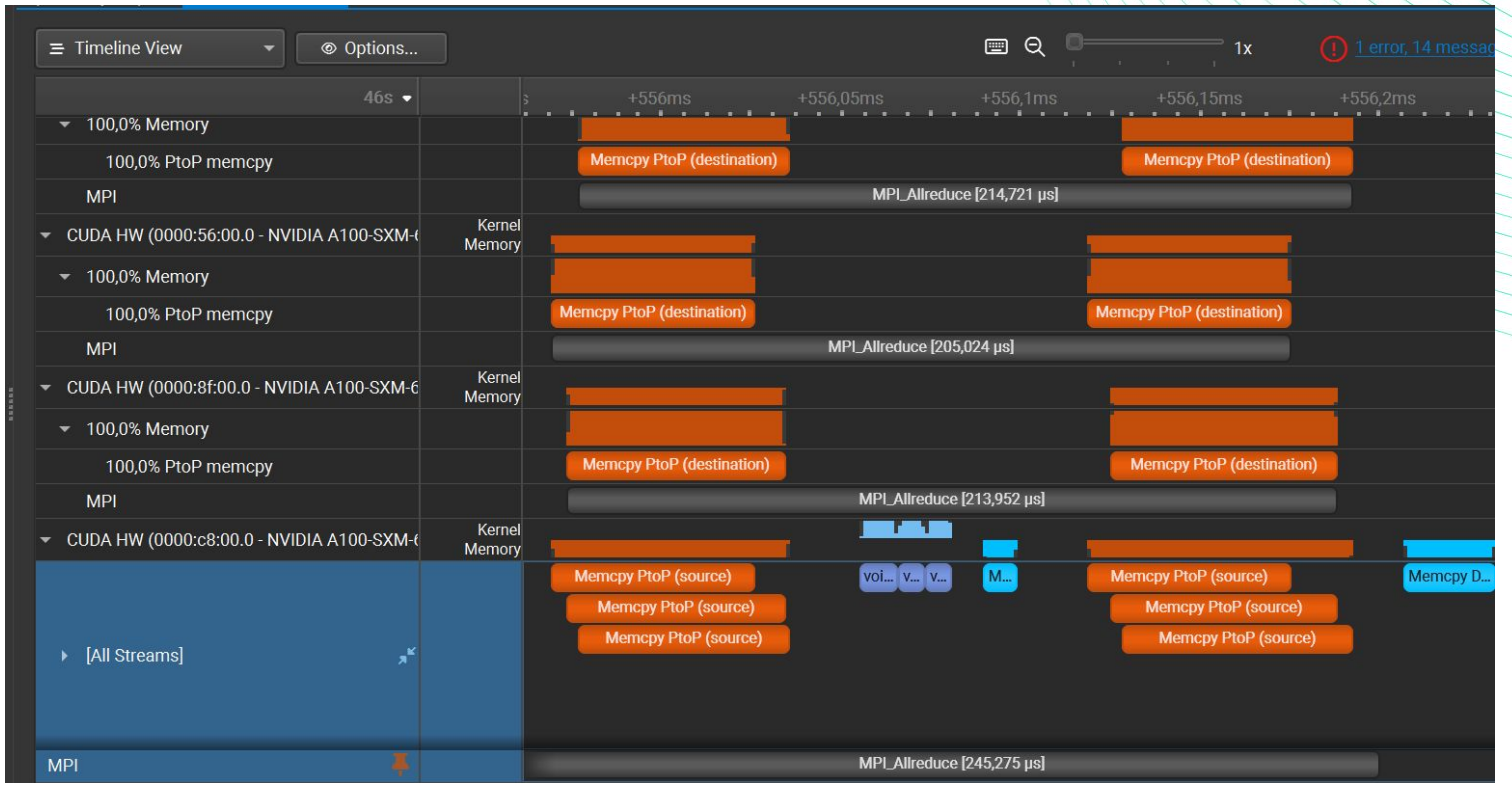


Network usage

```
-trace=mpi --nic-metrics=true
```



MPI awareness



EPICURE

CINECA

We acknowledge **OpenACC.org** and **NVIDIA** for fruitful learning and training material