INTRODUCTION TO OPENACC

BRENT LEBACK, MEMBER OF THE NVIDIA HPC SDK TEAM

OPENACC RESOURCES

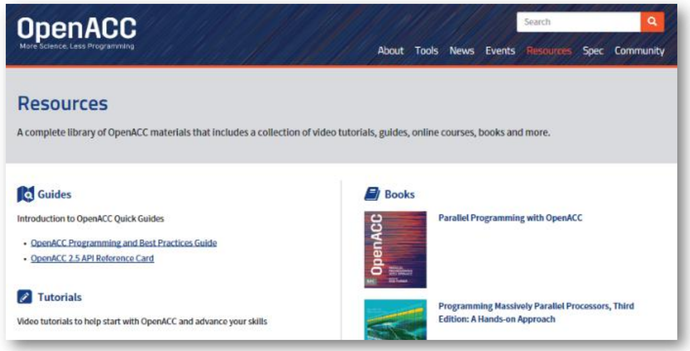
Guides ● Talks ● Tutorials ● Videos ● Books ● Spec ● Code Samples ● Teaching Materials ● Events ● Success Stories ● Courses ● Slack ● Stack Overflow

FREE Compilers

NVIDIA HPC SDK. Free download.

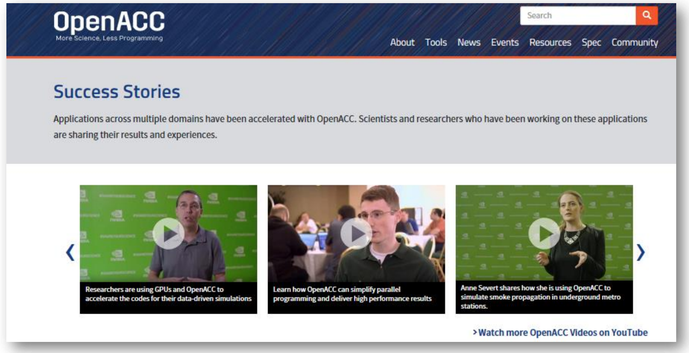
Resources

https://www.openacc.org/resources



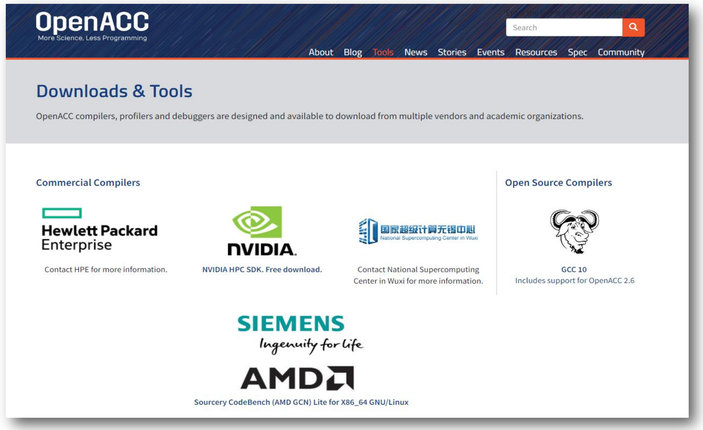
Success Stories

https://www.openacc.org/success-stories



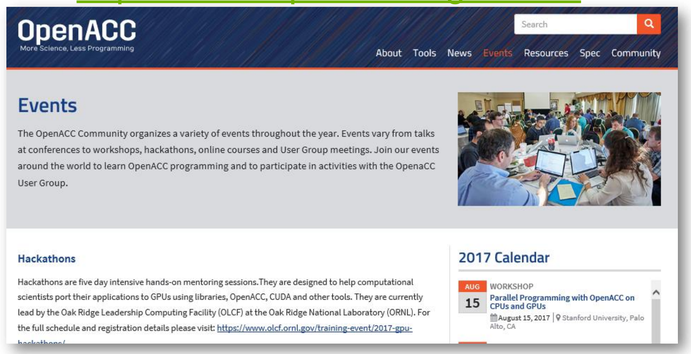
Compilers and Tools

https://www.openacc.org/tools



Events

https://www.openacc.org/events



NVIDIA

BASIC SYNTACTIC CONCEPTS

Directive-Based Programming Designed for Accelerated Computing

C/C++ OpenACC pragma syntax

`#pragma acc` *`directive [clause]... eol`*

continue to next line with backslash

Fortran OpenACC directive syntax

`!$acc` *`directive [clause]...`*

`&` continuation

Fortran-77 syntax rules

`!$acc` or `C$acc` or `*$acc` in columns 1-5

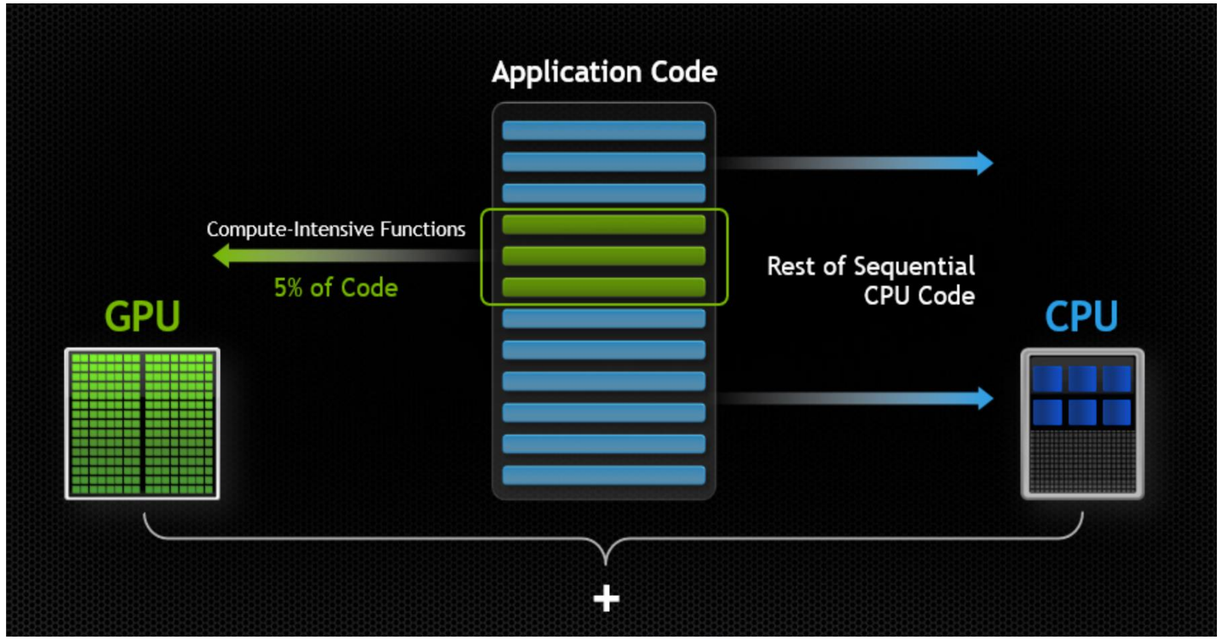
continuation with nonblank in column 6

Directives are roughly divided into 2 groups

Compute Directives

Data Management Directives

There is also a runtime API, typically used for device selection, other less-used functionality



OPENACC KERNELS CONSTRUCT

The kernels construct expresses that a region may contain parallelism and the compiler determines what can safely be parallelized.

```
#pragma acc kernels
{
  for (int i=0; i<N; i++)
  {
    a[i] = 0.0;
    b[i] = 1.0;
    c[i] = 2.0;
  }

  for (int i=0; i<N; i++)
  {
    a[i] = b[i] + c[i]
  }
}
```

kernel 1

kernel 2

The compiler identifies 2 parallel loops and is free to generate 1 or 2 kernels for offload

OPENACC PARALLEL LOOP DIRECTIVE

Parallel: a parallel region of code. The compiler generates a parallel kernel for that region.

Loop: identifies a loop that should be distributed across threads. Parallel and loop are often placed together.

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

<alternatively>

#pragma acc parallel
{
  #pragma acc loop
  for (int i=0; i<N; i++) {
    y[i] = a*x[i] + y[i];
  }
}  //end parallel in Fortran
```

1 parallel kernel

Similar to OpenMP, compiler translates the parallel region into a kernel that runs in parallel on the GPU

GANG, WORKER, AND VECTOR CLAUSES

- The developer can instruct the compiler which levels of parallelism to use on given loops by adding clauses:
- gang – Mark this loop for gang parallelism
- worker – Mark this loop for worker parallelism
- vector – Mark this loop for vector parallelism

These can be combined on the same loop.

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

```
#pragma acc parallel loop \
        collapse(3) gang vector
for( i = 0; i < size; i++ )
  for( j = 0; j < size; j++ )
    for( k = 0; k < size; k++ )
      c[i][j] += a[i][k] * b[k][j];
```

ADJUSTING GANGS, WORKERS, AND VECTORS

Useful when you know more about the loop bounds than the compiler

The compiler will choose a number of gangs, workers, and a vector length for you, but you can change it with clauses.

num_gangs(N) – Generate N gangs for this parallel region

num_workers(M) – Generate M workers for this parallel region

vector_length(Q) – Use a vector length of Q for this parallel region

```
#pragma acc parallel num_gangs(N) \
        num_workers(2) vector_length(32)
{
    #pragma acc loop gang worker
    for(int x = 0; x < N*2; x++) {
        #pragma acc loop vector
        for(int y = 0; y < 32; y++) {
            array[x][y]++;
        }
    }
}
```

SEQ CLAUSE

The **seq** clause (short for sequential) will tell the compiler to run the loop sequentially

In the sample code, 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, if it knows the count is short, or contains a loop-carried dependency

```
#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];
```

OPENACC COLLAPSE CLAUSE

Collapse(n): Applies the associated directive to the following *n* tightly nested loops

Useful when loop extents are short, or there are more loops than levels (gang, worker, vector) available

```
#pragma acc parallel
#pragma acc loop collapse(2)
for (int i=0; i<N; i++)
  for (int j=0; j<N; j++)
    ...
```

```
#pragma acc parallel
#pragma acc loop
for (int ij=0; ij<N*N; ij++)
    i = ij / N;
    j = ij % N;
    ...
```

CALLING USER ROUTINES IN DEVICE CODE

Use the same set of gang, worker, vector, seq to specify the parallelization level

```
! OpenACC
 real function fs(a)
  !$acc routine seq
  fs = a + 1.0
  end function

 subroutine fv(a,j,n)
  !$acc routine vector
  real :: a(n,n)
  !$acc loop vector
  do i = 1, n
   a(i,j) = fs(a(i,j))
  enddo
 end subroutine
```

```
! OpenACC
 subroutine fg(a,n)
  !$acc routine gang
  real :: a(n,n)
  !$acc loop gang
  do j = 1, n
   call fv(a,j,n)
  enddo
 end subroutine

 program main
 !$acc parallel num_gangs(100) vector_length(32)
   call fg(a,n)
 !$acc end parallel
 end program
```

REDUCTION CLAUSE

The **reduction** clause takes many values and “reduces” them to a single value, such as in a sum or maximum

Each thread calculates its part

Reductions can be over all gangs in the kernel, or within a gang

The compiler will perform a final reduction to produce a **single result** using the specified operation

```
for( i = 0; i < size; i++ )
  for( j = 0; j < size; j++ )
    for( k = 0; k < size; k++ )
      c[i][j] += a[i][k] * b[k][j];
```

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

REDUCTION CLAUSE 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)

DEFINING DATA REGIONS

The **data** construct defines a region of code in which GPU arrays remain on the GPU and are shared among all kernels in that region.

```
#pragma acc data
{
  #pragma acc parallel loop
  ...

  #pragma acc parallel loop
  ...
}
```

Arrays used within the data region will remain on the GPU until the end of the data region.

DATA CLAUSES

All copy and create clauses behave as their “present_or” variants

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.
copyin (*list*)	Allocates memory on GPU and copies data from host to GPU when entering region.
copyout (*list*)	Allocates memory on GPU and copies data to the host when exiting region.
create (*list*)	Allocates memory on GPU but does not copy.
present (*list*)	Data is already present on GPU from another containing data region.

UNSTRUCTURED DATA DIRECTIVES

Basic Example

enter data: Defines the start of an unstructured data region

clauses: copyin(list), create(list)

exit data: Defines the end of an unstructured data region

clauses: copyout(list), delete(list)

```
#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)
```

DYNAMIC DATA LIFETIMES

C

```
#pragma acc enter data copyin( list )
#pragma acc enter data create( list )
#pragma acc declare create( list )
```

Fortran

```
!$acc enter data copyin( list )
!$acc enter data create( list )
!$acc declare create( list )
```

Starts a data lifetime (not a data region)

Data appears in present table, just as structured data does

C

```
#pragma acc exit data delete( list ) copyout( list )
```

Fortran

```
!$acc exit data delete( list ) copyout( list )
```

OPENACC UPDATE DIRECTIVE

Update: Explicitly transfers data between the host and the device

Always updates, not a “present_or” operation

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

Clauses:

device: copies from the host to the device

host: copies data from the device to the host

```
#pragma acc update host(x[0:count])
MPI_Send(x,count,datatype,dest,tag,comm);
```

UPDATE DIRECTIVE

C

```
#pragma acc update host( list )
#pragma acc update device( list )
```

Fortran

```
!$acc update host( list )
!$acc update device( list )
```

Data must be in a data clause for an enclosing data region

- may be noncontiguous
- implies present(*list*)
- both may be on a single line

```
update host( list ) device( list )
```

ARRAY SHAPING

When the compiler fails to properly determine the size of arrays

Sometimes the compiler cannot determine the size of array

Examine the –Minfo output!

Developers must specify sizes explicitly using data clauses and array “shape”

C

#pragma acc data copyin(a[0:size]), copyout(b[s/4:3*s/4])

Numbers in brackets are starting-element : number-of-elements

Fortran

!$acc data copyin(a(1:end)), copyout(b(s/4+1:3*s/4))

Numbers in parenthesis are starting-element : ending-element

- Note: data clauses can be used on **data**, **parallel**, or **kernels directives**

SUMMARY: BASIC USE OF DATA DIRECTIVES IN OPENACC AND OPENMP

more similar than different

```
! OpenACC
!$acc data <clause>  ! Starts a structured data region

  copy(list) Allocates memory on the GPU and copies data from
host to GPU when entering region and copies data to the host
when exiting region.

  copyin(list) Allocates memory on the GPU and copies data from
host to GPU when entering region

  copyout(list) Allocates memory on GPU and copies data to the
host when exiting region.

  create(list)  Allocates memory on GPU but does not copy.

!$acc enter data <clause>  ! Starts unstructured data region.
  clause can be copyin or create

!$acc exit data <clause>  ! Ends unstructured data region.
  clause can be copyout or delete

!$acc update [host|self|device](list)
```

```
! OpenMP
!$omp target data<clause>  ! Starts a structured data region

  map(tofrom:list) Allocates memory on the GPU and copies data
from host to GPU when entering region and copies data to the host
when exiting region.

  map(to:list) Allocates memory on the GPU and copies data from
host to GPU when entering region

  map(from:list) Allocates memory on GPU and copies data to the
host when exiting region.

  map(alloc:list)  Allocates memory on GPU but does not copy.

!$omp target enter data <clause>  ! Starts unstructured data
region.
  clause can be map(to:) or map(alloc:)

!$omp target exit data <clause>  ! Ends unstructured data region.
  clause can be map(from:) or map(delete:)

!$omp target update [to|from](list)
```

PASSING DEVICE POINTERS TO CUDA LIBRARIES IN OPENACC

Getting the compiler to pass the device pointer within a data region

```
! OpenACC
 use curand
 integer, parameter :: N=10000000, HN=10000
 integer            :: a(N), h(HN), i
 integer(8)         :: nbits
 type(curandGenerator) :: g

 istat = curandCreateGenerator(g,CURAND_RNG_PSEUDO_XORWOW)

 !$acc data copy(a)

 !$acc host_data use_device(a)
 istat = curandGenerate(g, a, N)
 !$acc end host_data

 istat = curandDestroyGenerator(g)
 !$acc end data

 nbits = 0
 do i = 1, n
     nbits = nbits + popcnt(a(i))
 end do
 nbits = nbits / n
 print *,"Should be roughly half the bits set“, nbits
```

```
/* OpenACC */
#include “curand.h”
. . .
curandGenerator_t g;
double rmean, sumd;
a = (float *) malloc(n*4);

istat = curandCreateGenerator(&g, CURAND_RNG_PSEUDO_DEFAULT);
istat = curandSetStream(g,(cudaStream_t)acc_get_cuda_stream(acc_async_sync));
if (istat != CURAND_STATUS_SUCCESS) printf("Error from set stream\n");

/* Uniform */
printf("Should be uniform around 0.5\n");
#pragma acc data copy(a[0:n], b[0:n])
{
    #pragma acc host_data use_device(a)
    {
      istat = curandGenerateUniform(g, a, n);
      if (istat != CURAND_STATUS_SUCCESS) printf("Error %d\n",istat);
    }
    #pragma acc update host(a[0:n])
    sumd = 0.0;
    for (i = 0; i < n; i++) {
        if ((a[i] < 0.0f) || (a[i] > 1.0f)) passing = 0;
        sumd += (double) a[i];
    }
    rmean = sumd / (double) n;
```

ASYNCHRONOUS BEHAVIOR, QUEUES, STREAMS

1-1 correspondence between OpenACC async numbers and streams

```
! OpenACC
 !$acc data create(a, b, c)

  ierr = cufftPlan2D(iplan1,n,m,CUFFT_C2C)
  ierr = cufftSetStream(iplan1,acc_get_cuda_stream(10))

  !$acc update device(a) async(10)

  !$acc host_data use_device(a,b,c)
  ierr = ierr + cufftExecC2C(iplan1,a,b,CUFFT_FORWARD)
  ierr = ierr + cufftExecC2C(iplan1,b,c,CUFFT_INVERSE)
  !$acc end host_data

  ! scale c
  !$acc kernels async(10)
  c = c / (m*n)
  !$acc end kernels

  !$acc update host(c) async(10)
  !$acc wait(10)

  ! Check inverse answer
  write(*,*) 'Max error C2C INV: ', maxval(abs(a-c))

  !$acc end data
```

```
/* OpenACC */
 ierr = 0;
 ierr += cufftPlan2d(&plan2, m, n, CUFFT_R2C);
 ierr += cufftPlan2d(&plan3, m, n, CUFFT_C2R);
 ierr += cufftSetStream(plan2, (cudaStream_t) acc_get_cuda_stream(12));
 ierr += cufftSetStream(plan3, (cudaStream_t) acc_get_cuda_stream(12));

 float rmaxval = 0.0f;
 #pragma acc enter data copyin(r[0:m*n]) create(b[0:m*n],q[0:m*n]) async(12)
 #pragma acc host_data use_device(r, b, q)
 {
     ierr += cufftExecR2C(plan2, r, (cufftComplex *) b);
     ierr += cufftExecC2R(plan3, (cufftComplex *) b, q);
 }
 #pragma acc kernels async(12)
 {
     for (int i = 0; i < n; ++i) {
         for (int j = 0; j < m; ++j) {
             float x = fabs(r[i*m+j] - q[i*m+j] / (m*n));
             if (x > rmaxval) rmaxval = x;
          }
     }
 }
 #pragma acc exit data delete(r, b, q) async(12)  /* no copyout */
 #pragma acc wait(12)
 printf("Max error R2C/C2R: %f\n",rmaxval);
```

USING SHARED MEMORY FOR PERFORMANCE

Using the OpenACC Cache Directive on Gang-Private Data

```
! CUDA Fortran
real(kind=8), shared :: tile(blockDim%y,blockDim%x)

do jstart=(blockIdx%y-1)*blockDim%y, n, blockDim%y*gridDim%y
  do istart=(blockIdx%x-1)*blockDim%x, n, blockDim%x*gridDim%x
    i = threadIdx%x+istart
    j = threadIdx%y+jstart
    if (i<n .AND. j<n) then
      tile(threadIdx%y,threadIdx%x) = A(i,j)
    endif

    call syncthreads()

    i = threadIdx%y+istart
    j = threadIdx%x+jstart
    if (i<n .AND. j<n) then
       B(j,i)=tile(threadIdx%x,threadIdx%y)
    endif
  enddo
enddo
```

```
! OpenACC
 !$acc parallel loop gang collapse(2) vector_length(16*16) private(tile)
    do jstart=1, n, ythreads
       do istart=1, n, xthreads
          !$acc cache(tile(:,:))
          !$acc loop vector collapse(2)
          do jj = 1, ythreads  ! 1:16
             do ii = 1, xthreads  ! 1:16
                i = ii+istart-1
                j = jj+jstart-1
                if(i<n .AND. j<n) then
                   tile(ii,jj) = A(i,j)
                endif
             enddo
          enddo
          !$acc loop vector collapse(2)
          do ii = 1, xthreads
             do jj = 1, ythreads
                i = ii+istart-1
                j = jj+jstart-1
                if(i<n .AND. j<n) then
                   B(j,i) = tile(ii,jj)
                endif
             enddo
          enddo
       enddo
    enddo
 !$acc end parallel
```

NVIDIA HPC GPU COMPILERS COMMONLY USED OPTIONS

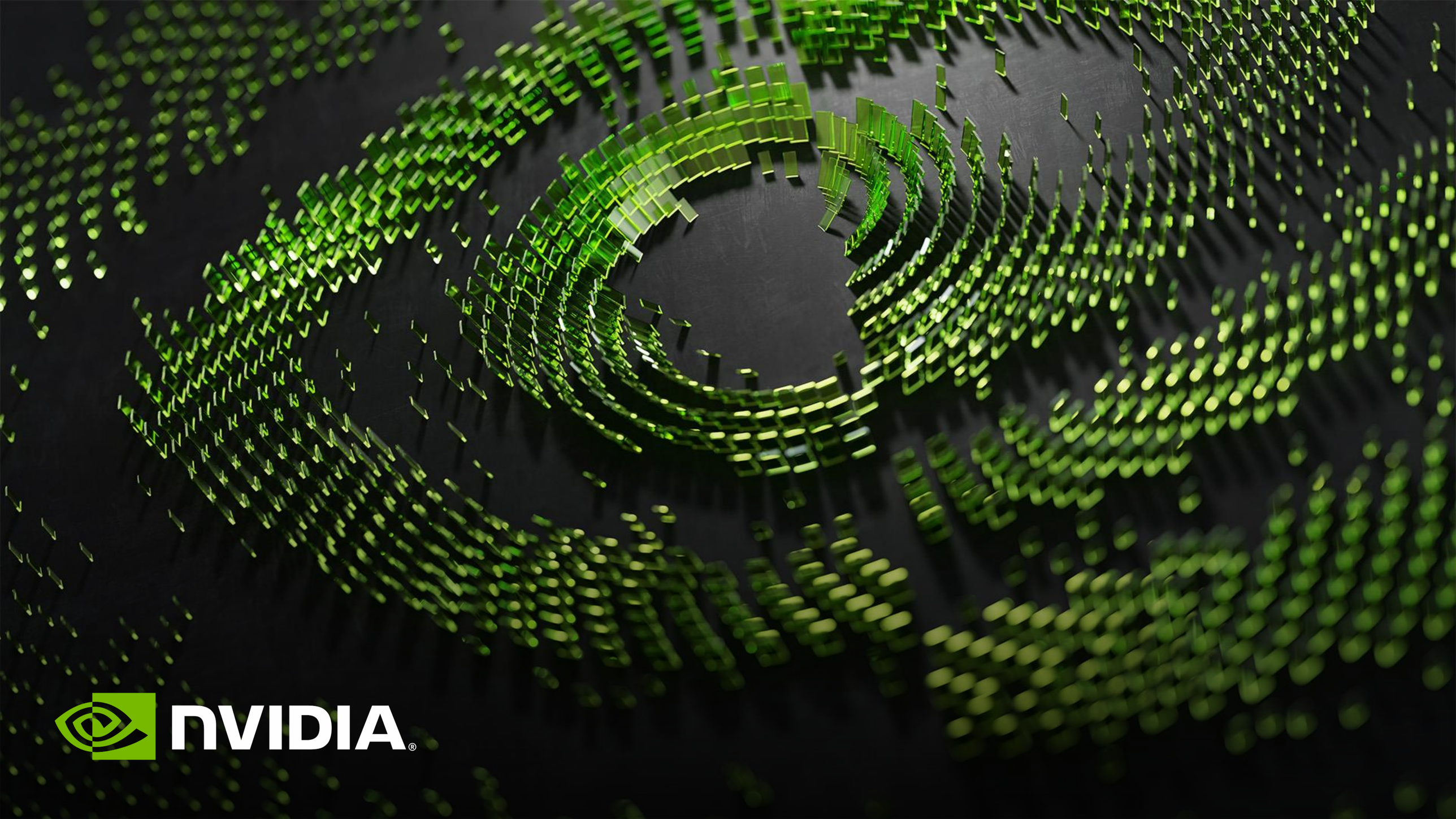
-gpu=[no]managed: use CUDA managed memory

-gpu=maxregcount:<n>: Set the maximum number of registers to use on the GPU

-gpu=lineinfo: Generate GPU code line information, useful for debugging, diagnostics

-gpu=ccXY: Only generate code for compute capability XY

-gpu=autocompare: Automatically compare CPU and GPU results (PCAST)
-gpu=redundant: Run redundantly on CPU and GPU (PCAST w/manually instrumented code)


NVIDIA