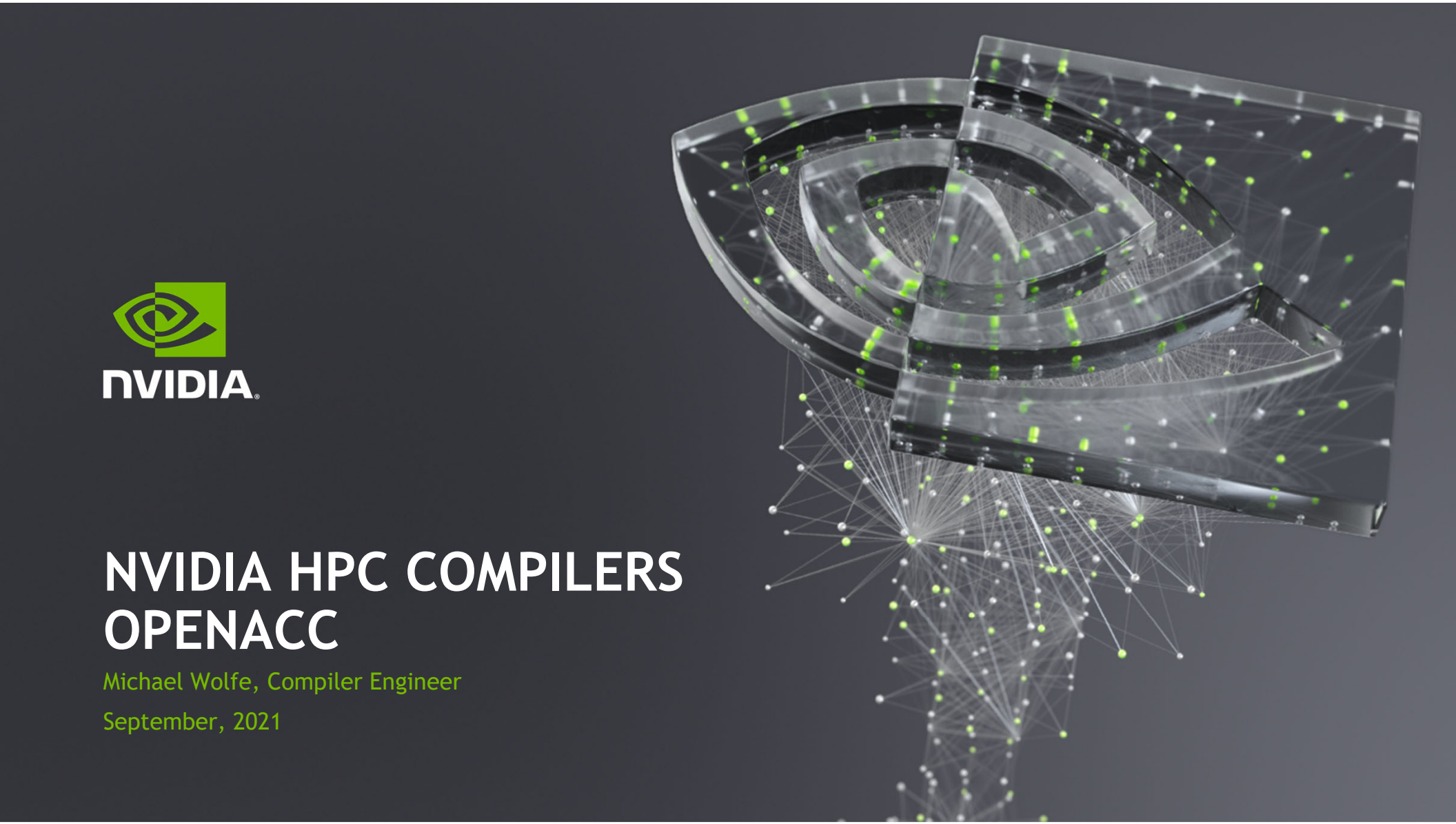

NVIDIA HPC COMPILERS OPENACC

Michael Wolfe, Compiler Engineer

September, 2021

THE FUTURE OF HPC GPU PROGRAMMING

Math Libraries | Standard Languages | Directives | CUDA

```
std::transform(par, x, x+n, y, y,
    [=] (float x, float y) {
        return y + a*x;
});
```

```
do concurrent (i = 1:n)
   y(i) = y(i) + a*x(i)
enddo
```

GPU Accelerated C++17 and Fortran 2018

```
#pragma acc data copy(x,y)
{

...

#pragma acc parallel loop ...
for (int i = 0; i < n; ++i)
  y[i] += a * x[i];

...

}
```

Incremental Performance Optimization with OpenACC

```
__global__
void saxpy(int n, float a,
           float *x, float *y) {
  int i = blockIdx.x*blockDim.x +
          threadIdx.x;
  if (i < n) y[i] += a*x[i];
}

int main(void) {
  ...
  cudaMemcpy(d_x, x, ...);
  cudaMemcpy(d_y, y, ...);

  saxpy<<<(N+255)/256,256>>>(...);

  cudaMemcpy(y, d_y, ...);
```

Maximize GPU Performance with CUDA C++/Fortran

OPENACC NEW FEATURES: ERROR HANDLER

```
% mpirun -np 400 vasp_std
```

What happens when one rank dies, due to an OpenACC programming error?

```
acc_callback_register(acc_ev_error, myhandler, acc_reg);

void myhandler(acc_callback_info* cb, acc_event_info* ev, acc_api_info* api) {
  fprintf(stderr, "error code %d: %s\n", ev->error_code, ev->error_message);
  mpi_abort(...);
}
```

OPENACC PERFORMANCE: SMALL DATA MOVEMENT

```
struct {
  float a, b;
} x, y, z;

#pragma acc enter data copyin(x, y, z)
```

```
cuMemcpyHtoD(xdev, &x, 8);
cuMemcpyHtoD(ydev, &y, 8);
cuMemcpyHtoD(zdev, &z, 8);
```

```
ups.p[0] = xdev;
ups.off[0] = 0;
ups.len[0] = 8;
ups.d[0] = x.a;
ups.d[1] = x.b;
upd.p[1] = ydev;
...
upload<<<...>>>(ups);
```

```
real, allocatable, dimension(:) :: a, b
allocate(a(...),b(...))

!$acc enter data copyin(a,b)
```

```
if(sizeof(a) < magic) {
  defer to small data upload
} else {
  cuMemcpyHtoD(adev, a, sizeof(a));
}
```

OPENACC PERFORMANCE: MULTIPLE VERSION KERNELS

```
#pragma acc parallel loop default(present)
for (i = 0; i < n; ++i)
  x[i] += a*y[i];
```

```
kernel(float* x, float* y, int n){
 int i;
 i = blockIdx.x * blockDim.x + threadIdx.x;
 x[i] += a * y[i];
}




kernel<<<n/128, 128>>>(xdev, ydev, n);
```

```
kernel(float* x, float* y, int n){
 int i;
 float2 xx, yy;
 i = blockIdx.x * blockDim.x + threadIdx.x;
 xx = *(float2*)(&(x[i*2]);
 yy = *(float2*)(&(y[i*2]);
 xx += a*yy;
 *(float2*)(&(x[i*2])) = xx;
}

kernel<<<n/256, 128>>>(xdev, ydev, n);
```

OPENACC DEBUGGING: AUTOCOMPARE

```
float ss = 0;
#pragma acc parallel loop default(present) reduction(+:ss) copy(ss)
for (i = 0; i < n; ++i) {
  x[i] *= a * y[i];
  ss += x[i];
}
```

```
% nvc r.c -acc=gpu -gpu=autocompare
./a.out 2400000

PCAST Float ss in function testit, /proj/scratch/mwolfe/test/mint/a9/r.c:12
idx: 0 FAIL ABS  act: 4.65269890e+01 exp: 4.61936684e+01 dif: 3.33320618e-01
sum = 46.193668, x = 2.335896, 1.339727, 0.939197, 0.723035, 0.587759, 0.495124...
compared 2 blocks, 2400001 elements, 9600004 bytes
1 errors found in 1 blocks
 absolute tolerance = 0.00000000000000000e+00, abs=0
```

OPENACC DEBUGGING: ATTACH

```
w%work => actual1(:,:,k)
!$acc enter data copyin(w%work)
!$acc parallel loop present(w)
do i ...
  do j ...
    ...w%work(i,j)...
  enddo
enddo
...
w%work => actual2(:,k,:)

!$acc parallel loop present(w)
do i = ...
  do j = ...
    ...w%work(i,j)...
  enddo
enddo
...
```

PERFORMANCE, PORTABILITY, PRODUCTIVITY

Good languages and intelligent compilers make writing parallel programs easier

For example, loop vectorization vs. SIMD intrinsics

```
!$omp metadirective &
!$omp when(target_device={kind(gpu)}:target teams distribute) &
!$omp default(parallel for)
do j = 1, n
  x(j) = sin(y(j))
  !$omp metadirective &
  !$omp when(device={kind(gpu)} : parallel for) &
  !$omp default(simd)
  do i = 1, m
    a(i,j) = a(i,j) + x(j) * z(i)
  enddo
enddo
```