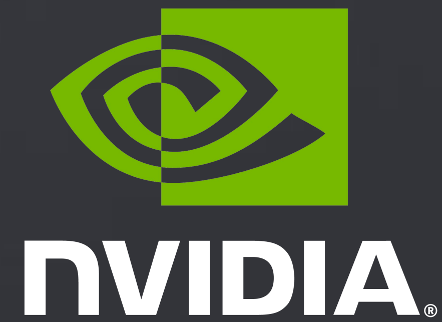



# Standard Fortran on GPUs and its utility in quantum chemistry codes

Jeff Hammond
NVIDIA HPC Group

# Outline

- What is NWChem?
- What is CCSD(T) and how is it computed?
- NWChem CCSD(T) kernels on A100 with Fortran DO CONCURRENT.
- Other Fortran standard parallelism features in the NVHPC SDK.

# What is NWChem?

TL;DR Widely used open-source quantum chemistry code.

- Project started in 1990s with goal of running well on all major HPC platforms.
  - Developed Global Arrays for parallelism before MPI 1.0 existed.
  - All platform features (OS, CPU, I/O, etc) hidden from application code.
- Millions of lines of code, primarily pre-modern Fortran and C.
  - Fortran: 1MLOC by humans, 3MLOC generated.
  - ~100KLOC of C for system code, including memory allocator and I/O (incl. network)
- Primary functional portability bottleneck **was** ARMCI until MPI was embraced.
  - MPI-PR (progress rank) and ARMCI-MPI are the primary communication layers now.
- Primary performance bottlenecks are BLAS, MPI (bandwidth and async progress).
  - Fortran code generation and OpenMP runtime are secondary performance effects.

https://www.nersc.gov/assets/Uploads/Hammond-NERSC-OpenMP-August-2019-1.pdf


NVIDIA.

# What is CCSD(T)?

TL;DR the “gold standard” quantum chemistry method

CCSD(T) = Coupled Cluster Singles and Doubles with (perturbative) Triples

Why do we use CCSD(T)?

- CCSD is $O(n_{iter}N^6)$ compute and $O(N^4)$ storage, but isn’t accurate enough
- CCSDT is $O(n_{iter}N^8)$ compute and $O(N^6)$ storage, which is intractable for large systems
- CCSD(T) is $O(N^7)$ compute and $O(N^4)$ storage, and is quite accurate most of the time


“Why CCSD(T) works: a different perspective” by John Stanton
https://doi.org/10.1016/S0009-2614(97)01144-5

“Molecular Electronic-Structure Theory” by Trygve Helgaker, Poul Jørgensen, Jeppe Olsen
https://onlinelibrary.wiley.com/doi/book/10.1002/9781119019572

# How do we compute CCSD(T)?

TL;DR lots and lots of tensor contractions

How do we implement CCSD(T)?

- Iterate SCF equations to get orbitals, U
- Generate two-body Hamiltonian matrix elements, V, from U
- Iterate CCSD equations to get Singles and Doubles amplitudes, S and D
- Generate batches of approximate Triples amplitudes and the associated energy:

$$T_{ijk}^{abc(2)} = -P_{ijk}^{abc} \frac{\sum_d D_{ij}^{ad} V_{ckbd} - \sum_l D_{ij}^{ab} V_{cklj}}{\epsilon_a + \epsilon_b + \epsilon_c - \epsilon_i - \epsilon_j - \epsilon_k}$$

https://doi.org/10.1109/ICPP.2015.106
https://dl.acm.org/doi/10.1145/3205289.3205296



NVIDIA.

```fortran
!!! syntax modified for slide purposes !!!
subroutine par_sd_t_d2_9(h3d,h2d,h1d,p6d,p5d,p4d,p7d,
&                        triplesx,t2sub,v2sub)
integer :: h3d,h2d,h1d,p6d,p5d,p4d,p7d
integer :: h3,h2,h1,p6,p5,p4,p7
double precision :: triplesx(h2d,h3d,h1d,p4d,p6d,p5d)
double precision :: t2sub(p7d,p4d,h1d,h2d)
double precision :: t2tmp(p7d,h2d,h1d,p4d)
double precision :: v2sub(p7d,h3d,p6d,p5d)
! transposing inputs improves memory access, hence performance
do concurrent (h2=1:h2d, h1=1:h1d, p4=1:p4d, p7=1:p7d)
  t2tmp(p7,h2,h1,p4) = t2sub(p7,p4,h1,h2)
enddo
do concurrent (p5=1:p5d, p6=1:p6d, p4=1:p4d, h1=1:h1d, h3=1:h3d, h2=1:h2d)
  do p7=1,p7d ! no reduction support...yet
    triplesx(h2,h3,h1,p4,p6,p5) += t2tmp(p7,h2,h1,p4) * v2sub(p7,h3,p6,p5)
  enddo
enddo
end
```

# Experiments

A100 DGX Station

- CPU: AMD EPYC 7742
- GPU: NVIDIA A100 SXM 80GB

Compilers and Math Libraries

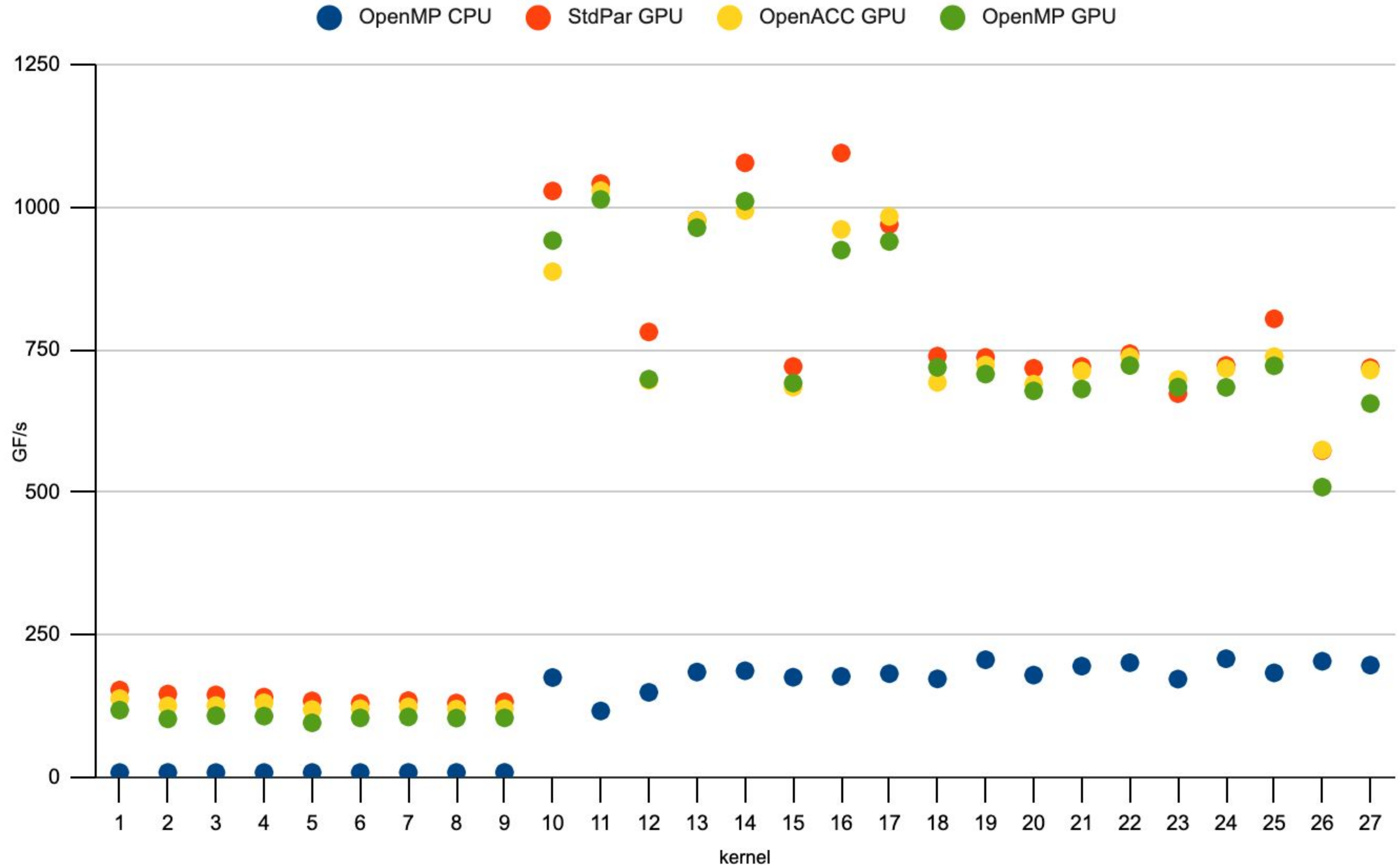
- NVHPC 21.7 compilers and OpenBLAS
- CUTENSOR 1.3.2 (will ship with NVHPC 21.X for X>9, but available now*)

NTTK (standalone driver for NWChem TCE CCSD(T) kernels)

- https://github.com/jeffhammond/nwchem-tce-triples-kernels
- tilesize = 30 keeps the memory footprint under 6 GB

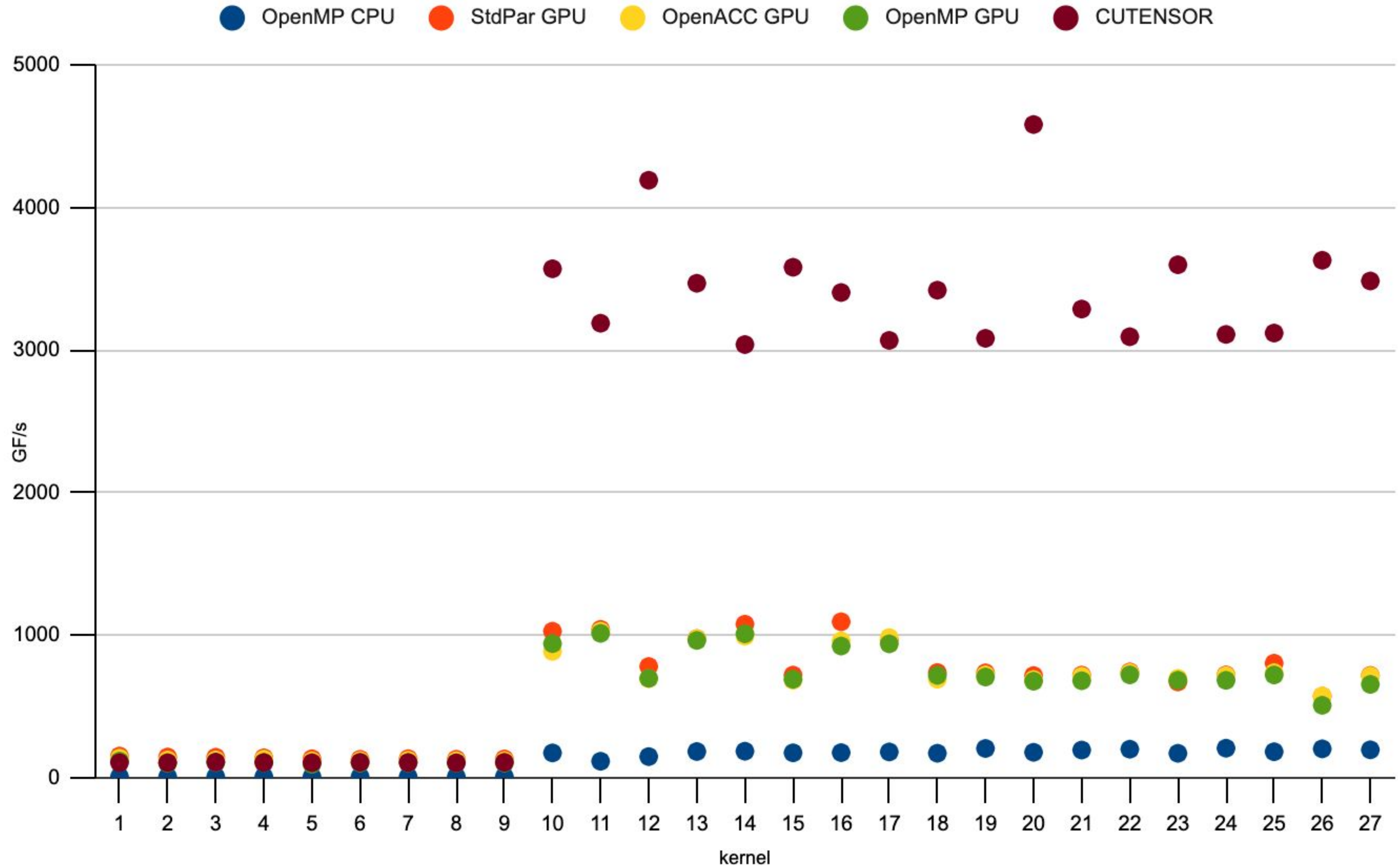
https://developer.nvidia.com/cutensor/1.3.2/downloads

StdPar = DO CONCURRENT




NVIDIA.

StdPar = DO CONCURRENT




NVIDIA.

# Summary so far

DO CONCURRENT, when used in conjunction with unified memory, allows you to write GPU code that has no explicit reference to GPUs or to offload models.

DO CONCURRENT performs on par with both OpenACC and OpenMP target offload on GPUs.

Thus, if you can do it with DO CONCURRENT, there is no need to use directives, and certainly not explicit offload ones.

If your application maps to an optimized library, you should use that.



NVIDIA.

```fortran
!!! syntax modified for slide purposes !!!
subroutine par_sd_t_d2_9(h3d,h2d,h1d,p6d,p5d,p4d,p7d,
&                         triplesx,t2sub,v2sub)
integer :: h3d,h2d,h1d,p6d,p5d,p4d,p7d
integer :: h3,h2,h1,p6,p5,p4,p7
double precision :: triplesx(h2d,h3d,h1d,p4d,p6d,p5d)
double precision :: t2sub(p7d,p4d,h1d,h2d)
double precision :: t2tmp(p7d,h2d,h1d,p4d)
double precision :: v2sub(p7d,h3d,p6d,p5d)
! transposing inputs improves memory access, hence performance
do concurrent (h2=1:h2d, h1=1:h1d, p4=1:p4d, p7=1:p7d)
  t2tmp(p7,h2,h1,p4) = t2sub(p7,p4,h1,h2)
enddo
do concurrent (p5=1:p5d, p6=1:p6d, p4=1:p4d, h1=1:h1d, h3=1:h3d, h2=1:h2d)
  do p7=1,p7d
    triplesx(h2,h3,h1,p4,p6,p5) += t2tmp(p7,h2,h1,p4) * v2sub(p7,h3,p6,p5)
  enddo
enddo
end
```

```fortran
! OpenMP host is CPU-optimized
! primarily using Intel Fortran
subroutine omp_sd_t_d2_9(..)
...
!$omp  parallel do collapse(3)
do p5=1,p5d
do p6=1,p6d
do h1=1,h1d
do h2=1,h2d
do p4=1,p4d
do h3=1,h3d
!$omp simd
do p7=1,p7d
 triplesx(h2,h3,h1,p4,p6,p5) ...
enddo...
!$omp end parallel do
end
```

```fortran
! CPU-optimized DO CONCURRENT
! for Intel Fortran
subroutine par_sd_t_d2_9(..)
...

do concurrent (p5=1:p5d)
do concurrent (p6=1:p6d)
do concurrent (p4=1:p4d)
do h1=1,h1d
do h3=1,h3d
do h2=1,h2d

do p7=1,p7d
 triplesx(h2,h3,h1,p4,p6,p5) ...
enddo...

end
```

# Tuning principles

If the compiler interprets DO CONCURRENT prescriptively (like OpenMP), then you should only use it when you know that parallelism is appropriate. This will make your code less performance portable.

If your compiler interprets DO CONCURRENT descriptively (like OpenACC), then you can use it everywhere that it is correct to do so, and the compiler will (likely) make the right choices for you.

Optimizing for data access is different on CPU and GPU. DO CONCURRENT offers no magic for such problems.

Whenever possible, use a performance library rather than loop code 🙂



NVIDIA.

# ACCELERATED PROGRAMMING IN ISO FORTRAN

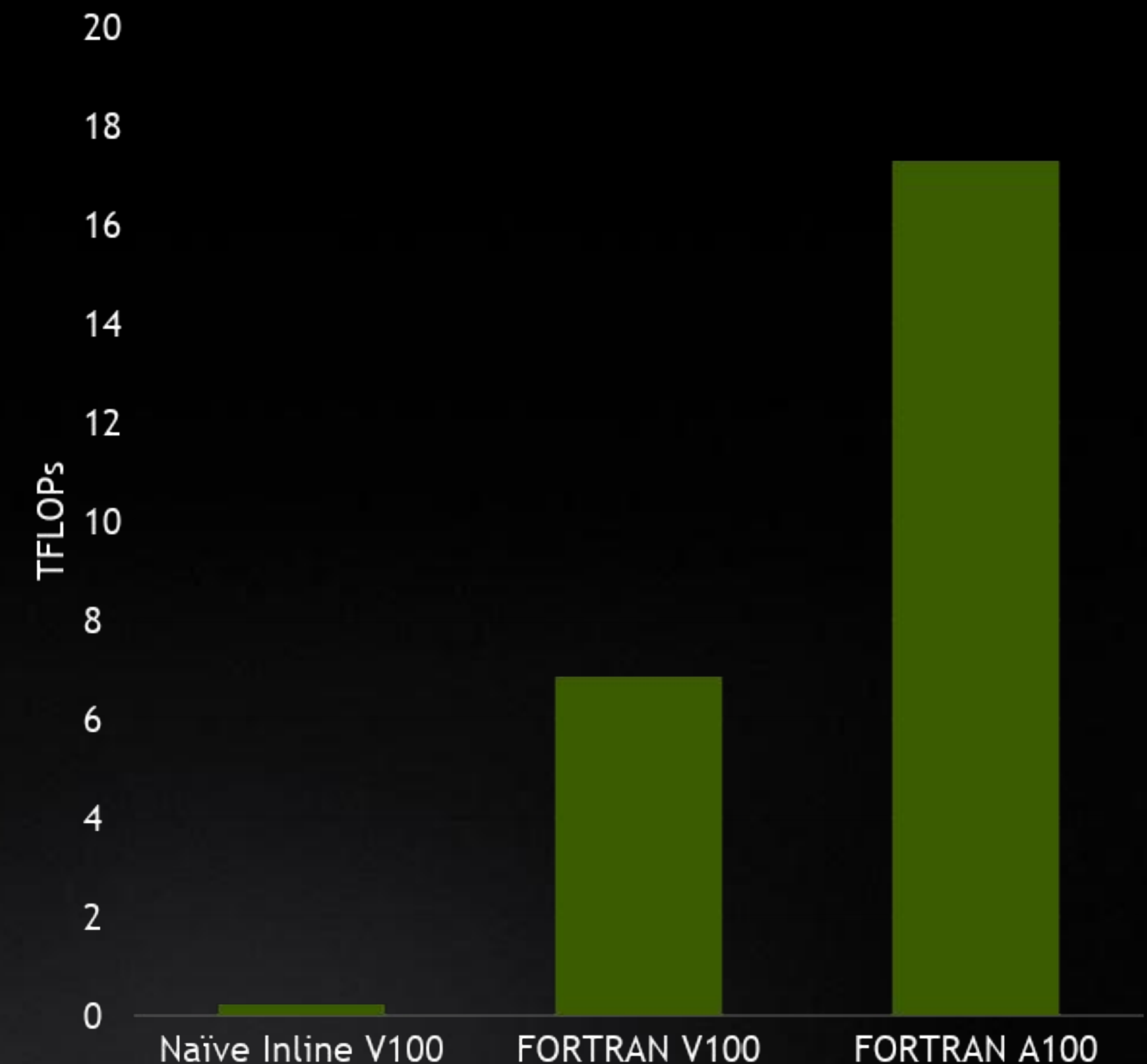
## NVFORTRAN Accelerates Fortran Intrinsics with cuTENSOR Backend

```
real(8), dimension(ni,nk) :: a
real(8), dimension(nk,nj) :: b
real(8), dimension(ni,nj) :: c
...
!$acc enter data copyin(a,b,c) create(d)

do nt = 1, ntimes
  !$acc kernels
  do j = 1, nj
    do i = 1, ni
      d(i,j) = c(i,j)
      do k = 1, nk
        d(i,j) = d(i,j) + a(i,k) * b(k,j)
      end do
    end do
  end do
    !$acc end kernels
end do

!$acc exit data copyout(d)
```

Inline FP64 matrix multiply

```
real(8), dimension(ni,nk) :: a
real(8), dimension(nk,nj) :: b
real(8), dimension(ni,nj) :: c

...

do nt = 1, ntimes
  d = c + matmul(a,b)
end do
```

MATMUL FP64 matrix multiply





NVIDIA.

# HPC PROGRAMMING IN ISO FORTRAN

## Examples of Patterns Accelerated in NVFORTRAN

```
d = 2.5 * ceil(transpose(a)) + 3.0 * abs(transpose(b))
d = 2.5 * ceil(transpose(a)) + 3.0 * abs(b)
d = reshape(a,shape=[ni,nj,nk])
d = reshape(a,shape=[ni,nk,nj])
d = 2.5 * sqrt(reshape(a,shape=[ni,nk,nj],order=[1,3,2]))
d = alpha * conjg(reshape(a,shape=[ni,nk,nj],order=[1,3,2]))
d = reshape(a,shape=[ni,nk,nj],order=[1,3,2])
d = reshape(a,shape=[nk,ni,nj],order=[2,3,1])
d = reshape(a,shape=[ni*nj,nk])
d = reshape(a,shape=[nk,ni*nj],order=[2,1])
d = reshape(a,shape=[64,2,16,16,64],order=[5,2,3,4,1])
d = abs(reshape(a,shape=[64,2,16,16,64],order=[5,2,3,4,1]))
c = matmul(a,b)
c = matmul(transpose(a),b)
c = matmul(reshape(a,shape=[m,k],order=[2,1]),b)
c = matmul(a,transpose(b))
c = matmul(a,reshape(b,shape=[k,n],order=[2,1]))
```

```
c = matmul(transpose(a),transpose(b))
c = matmul(transpose(a),reshape(b,shape=[k,n],order=[2,1]))
d = spread(a,dim=3,ncopies=nk)
d = spread(a,dim=1,ncopies=ni)
d = spread(a,dim=2,ncopies=nx)
d = alpha * abs(spread(a,dim=2,ncopies=nx))
d = alpha * spread(a,dim=2,ncopies=nx)
d = abs(spread(a,dim=2,ncopies=nx))
d = transpose(a)
d = alpha * transpose(a)
d = alpha * ceil(transpose(a))
d = alpha * conjg(transpose(a))
c = c + matmul(a,b)
c = c - matmul(a,b)
c = c + alpha * matmul(a,b)
d = alpha * matmul(a,b) + c
d = alpha * matmul(a,b) + beta * c
```



NVIDIA.

# Summary

- NWChem CCSD(T) kernels can achieve ~1 TF/s with straightforward DO CONCURRENT code and ~4 TF/s with cuTENSOR.

- The NVHPC SDK supports both DO CONCURRENT and array intrinsics running on GPUs.
  - DO CONCURRENT behaves approximately like OpenACC.
  - Array intrinsics map onto cuTENSOR when recognized.
  - The compiler provides diagnostics regarding what it is doing with both of these.

- StdPar offers a reasonable compromise on performance portability, particularly relative to OpenMP, which offers different programming models for CPU and GPU.

- Unified memory is a critical hardware feature for supporting these features.

# Questions/Comments

Twitter: https://twitter.com/science_dot

Email: jeff_hammond@acm.org

LinkedIn: https://www.linkedin.com/in/jeffhammond/

NVIDIA

# PROGRAMMING THE NVIDIA PLATFORM

CPU, GPU, and Network

### Accelerated Standard Languages

```
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
```

```
import legate.numpy as np
…
def saxpy(a, x, y):
    y[:] += a*x
```

### Incremental Portable Optimization

```
#pragma acc data copy(x,y) {
...
std::transform(par, x, x+n, y, y,
    [=](float x, float y){
        return y + a*x;
});
...
}
```

```
#pragma omp target data map(x,y) {
...
std::transform(par, x, x+n, y, y,
    [=](float x, float y){
        return y + a*x;
});
...
}
```

### Platform Specialization

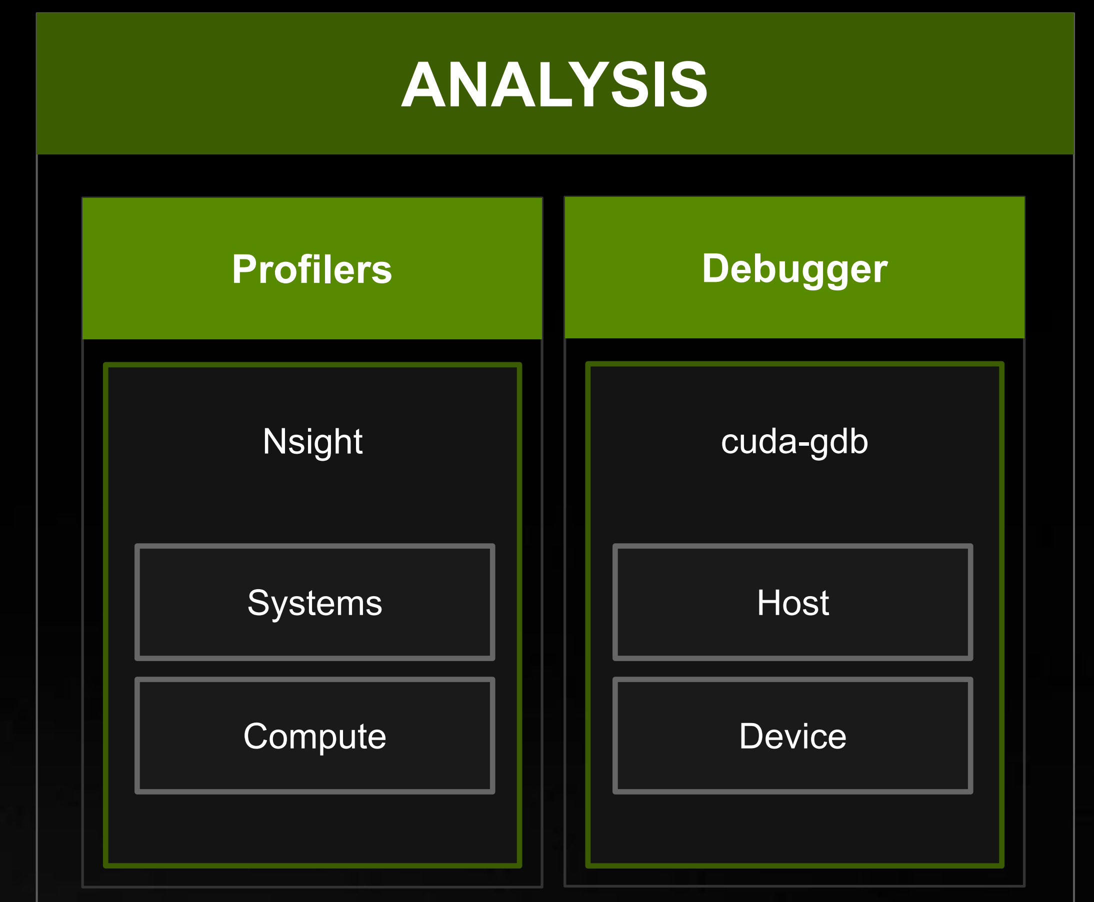
```
__global__
void saxpy(int n, float a,
           float *x, float *y) {
  int i = blockIdx.x*blockDim.x +
          threadIdx.x;
  if (i < n) y[i] += a*x[i];
}

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

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

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

| Core | Math | Communication | Data Analytics | AI | Quantum |
|---|---|---|---|---|---|

Acceleration Libraries

# NVIDIA HPC SDK

Available at developer.nvidia.com/hpc-sdk, on NGC, via Spack, and in the Cloud




Develop for the NVIDIA Platform: GPU, CPU and Interconnect
Libraries | Accelerated C++ and Fortran | Directives | CUDA
7-8 Releases Per Year | Freely Available

# PROGRAMMING THE NVIDIA PLATFORM

## CPU, GPU, and Network

### Accelerated Standard Languages

```
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
```

```
import legate.numpy as np
…
def saxpy(a, x, y):
    y[:] += a*x
```

### Incremental Portable Optimization

```
#pragma acc data copy(x,y) {
...
std::transform(par, x, x+n, y, y,
    [=](float x, float y){
        return y + a*x;
});
...
}
```

```
#pragma omp target data map(x,y) {
...
std::transform(par, x, x+n, y, y,
    [=](float x, float y){
        return y + a*x;
});
...
}
```

### Platform Specialization

```
__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, ...);
```

| Core | Math | Communication | Data Analytics | AI | Quantum |
|---|---|---|---|---|---|

Acceleration Libraries

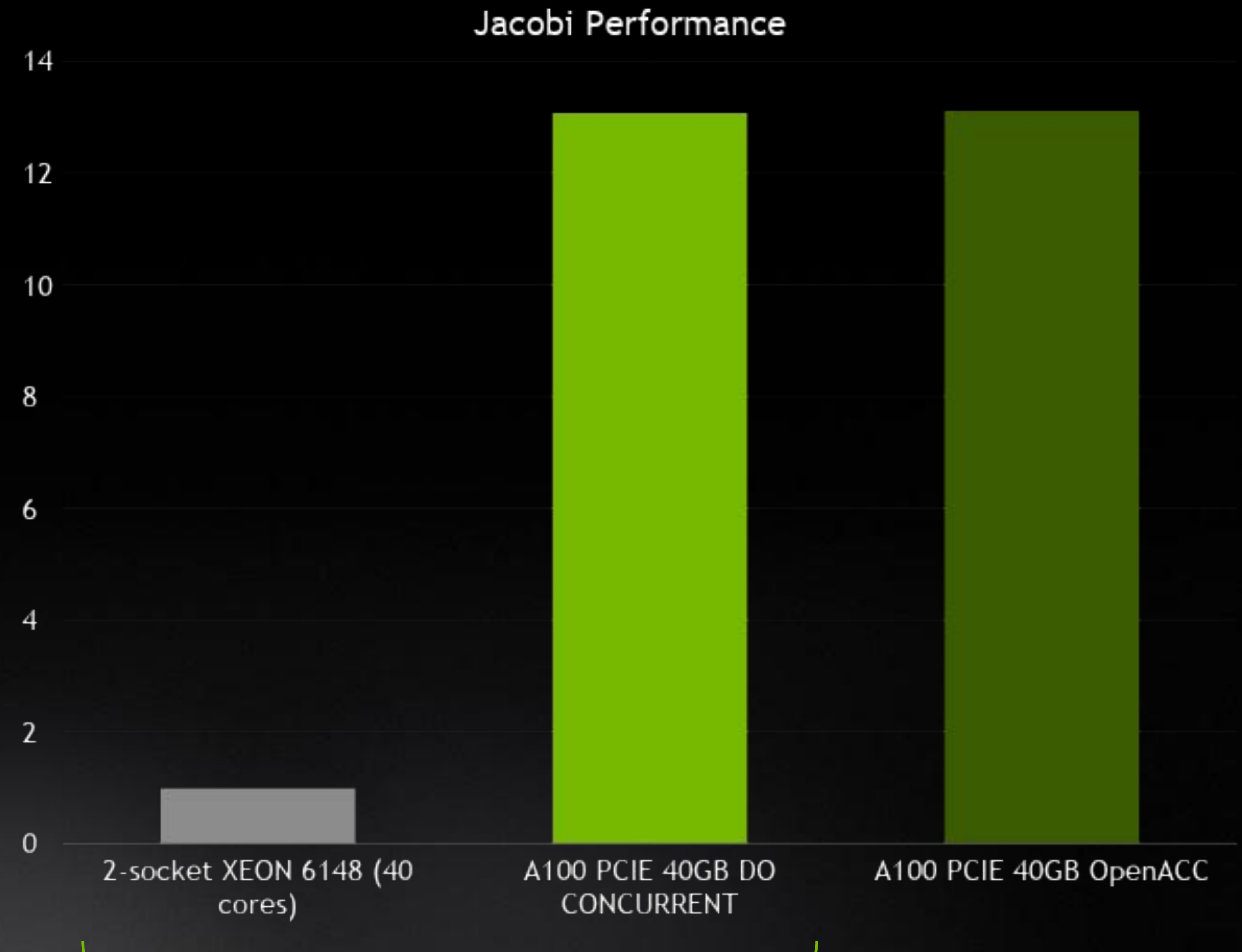
# ACCELERATED PROGRAMMING IN ISO FORTRAN

## DO CONCURRENT

### DO CONCURRENT in NVFORTRAN

- Available since NVFORTRAN 20.11
- Automatic GPU acceleration & multi-core support
- Syntax for nested parallelism / loop collapse; expose more parallelism to the compiler

```
subroutine smooth( a, b, w0, w1, w2, n, m, niters )
 real, dimension(:,:) :: a,b
 real :: w0, w1, w2
 integer :: n, m, niters
 integer :: i, j, iter
  do iter = 1,niters
   do concurrent(i=2 : n-1, j=2 : m-1)
     a(i,j) = w0 * b(i,j) + &
              w1 * (b(i-1,j) + b(i,j-1) + b(i+1,j) + b(i,j+1)) + &
              w2 * (b(i-1,j-1) + b(i-1,j+1) + b(i+1,j-1) + b(i+1,j+1))
   enddo
   do concurrent(i=2 : n-1, j=2 : m-1)
     b(i,j) = w0 * a(i,j) + &
              w1 * (a(i-1,j) + a(i,j-1) + a(i+1,j) + a(i,j+1)) + &
              w2 * (a(i-1,j-1) + a(i-1,j+1) + a(i+1,j-1) + a(i+1,j+1))
   enddo
  enddo
```






NVIDIA.