Introduction to GPU and Accelerator Architectures

Brent Leback *- bleback@nvidia.com*

NVIDIA HPC SDK Team

March 3rd, 2022

NCAR
UCAR

NVIDIA®

Workshop Etiquette

- Please mute yourself and turn off video during the session.

- Questions may be submitted in the chat and will be answered when appropriate. You may also raise your hand, unmute, and ask questions during Q&A at the end of the presentation.

- By joining today, you are agreeing to UCAR's Code of Conduct

- Recordings & other material will be archived & shared publicly.

- Feel free to follow up with the GPU workshop team via Slack or submit support requests to support.ucar.edu
  - **Office Hours**: Asynchronous support via Slack or schedule a time

Workshop Series and Logistics

- Sequence of sessions thru Aug 2022 detailed on main webpage
  - Full workshop course description/syllabus
  - Useful resources for independent self-directed learning included

- Registrants may use workshop's Project ID & Casper core hours
  - Please only **submit non-production, test/debug scale jobs**
  - Some workshop sessions will feature interactive coding
  - For non-workshop/learning work, request an allocation. Startup allocations may be available for new faculty and graduate students.
  - New NCAR HPC users should review our HPC Tutorials page

- Continue discussion on NCAR GPU Users Slack. May schedule individual office hours with the GPU Workshop organizers here or contact dhoward@ucar.edu.

NVIDIA HPC SDK

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

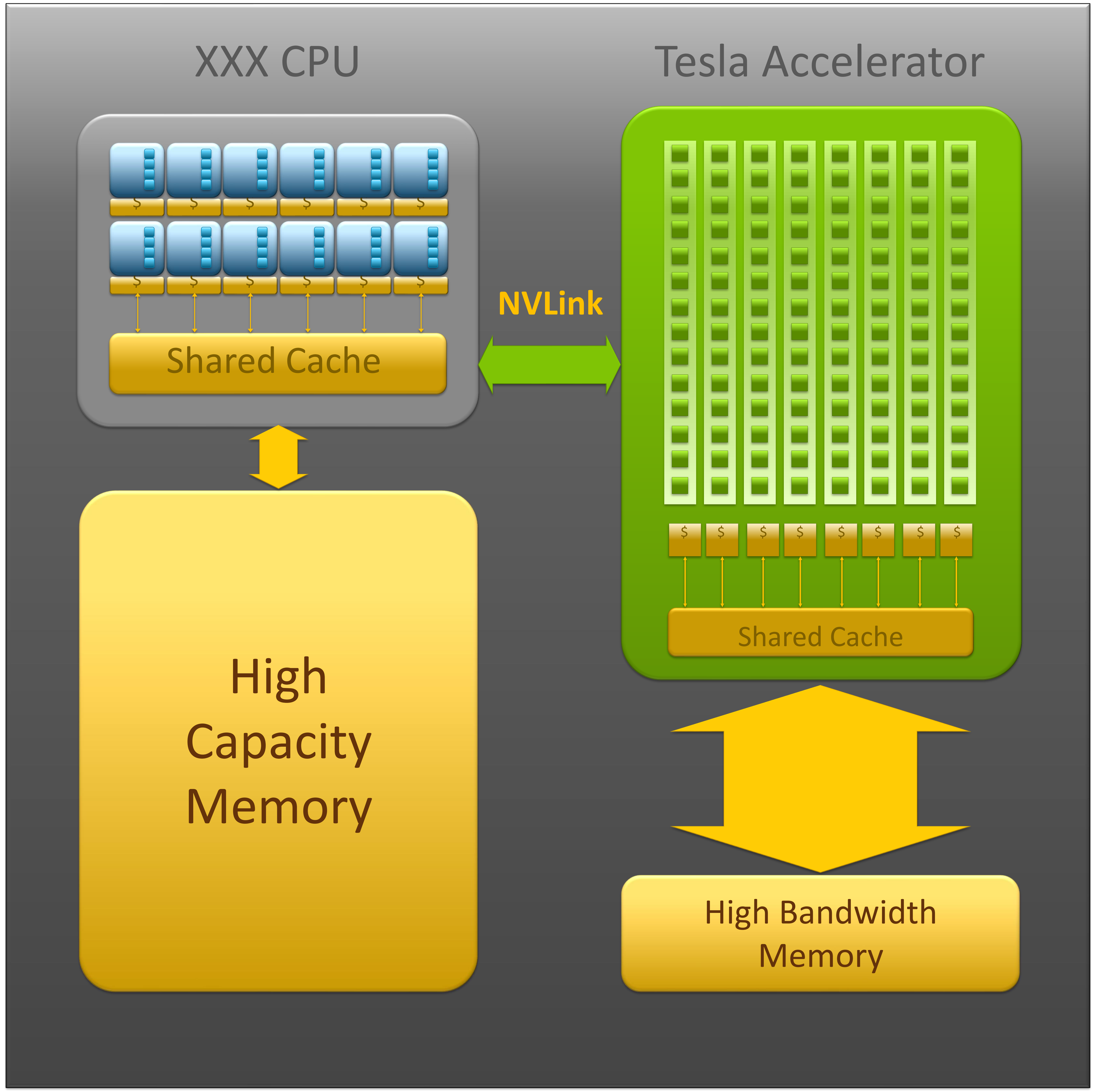
DEVELOPMENT
Programming Models
Standard C++ & Fortran
OpenACC & OpenMP
CUDA
Compilers
nvcc
nvc
nvc++
nvfortran
Core Libraries
libcu++
Thrust
CUB
Math Libraries
cuBLAS
cuTENSOR
cuSPARSE
cuSOLVER
cuFFT
cuRAND
Communication Libraries
HPC-X
MPI
UCX
SHMEM
SHARP
HCOLL
NVSHMEM
NCCL
ANALYSIS
Profilers
Nsight
Systems
Compute
Debugger
cuda-gdb
Host
Device

Develop for the NVIDIA Platform: GPU, CPU and Interconnect

Libraries | Accelerated C++ and Fortran | Directives | CUDA

7-8 Releases Per Year | Freely Available

A SLIDE FROM A PREVIOUS TALK AT NCAR (2016)



PROCESSOR COUNTS THROUGH THE YEARS

o

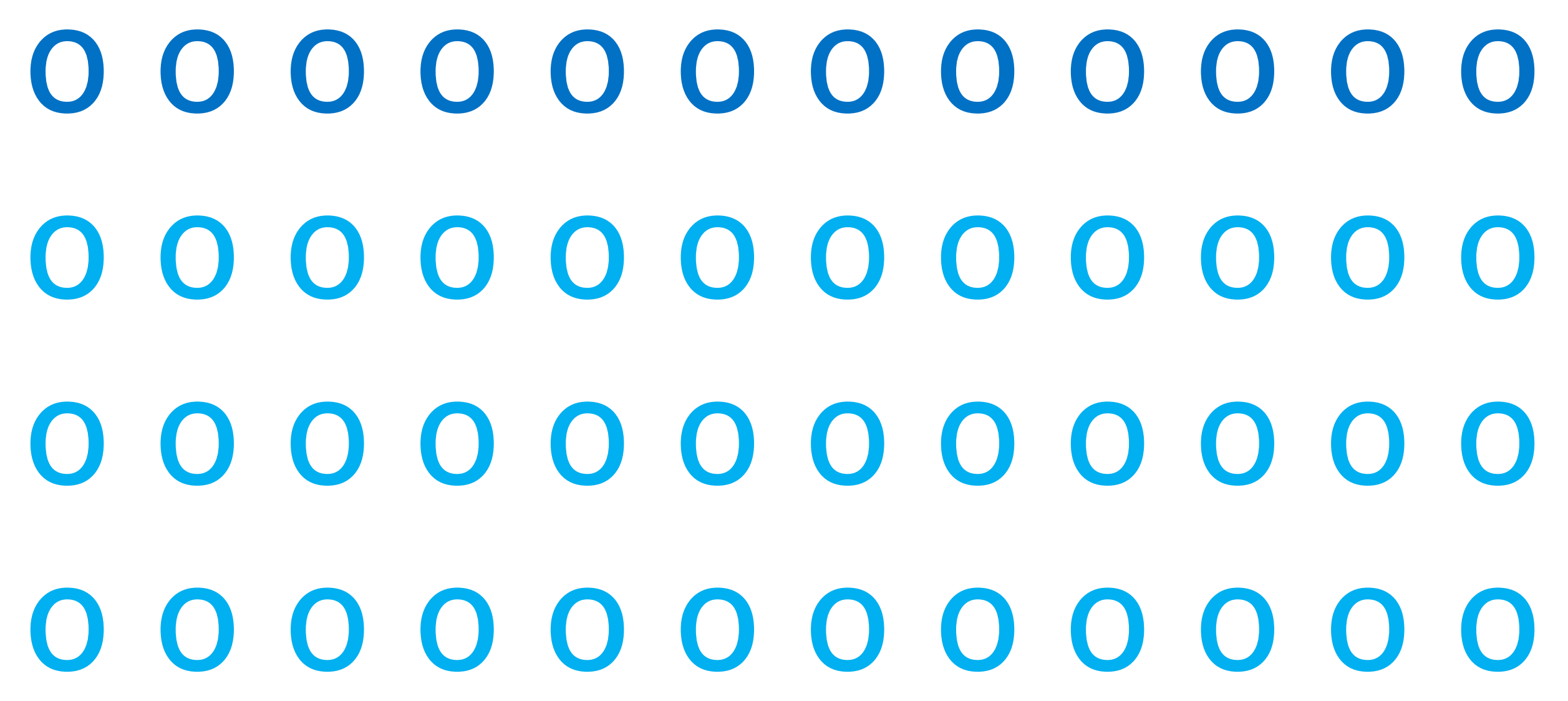
- Parallelism via MPI
- Performance improvements via manufacturing/process improvements, ILP, more registers, faster clocks.
- CPU SW gained features like dynamic memory allocation, large heaps, large stack.

THE ADVENT OF SIMD HARDWARE AND INSTRUCTIONS

- o
- o
- o
- o

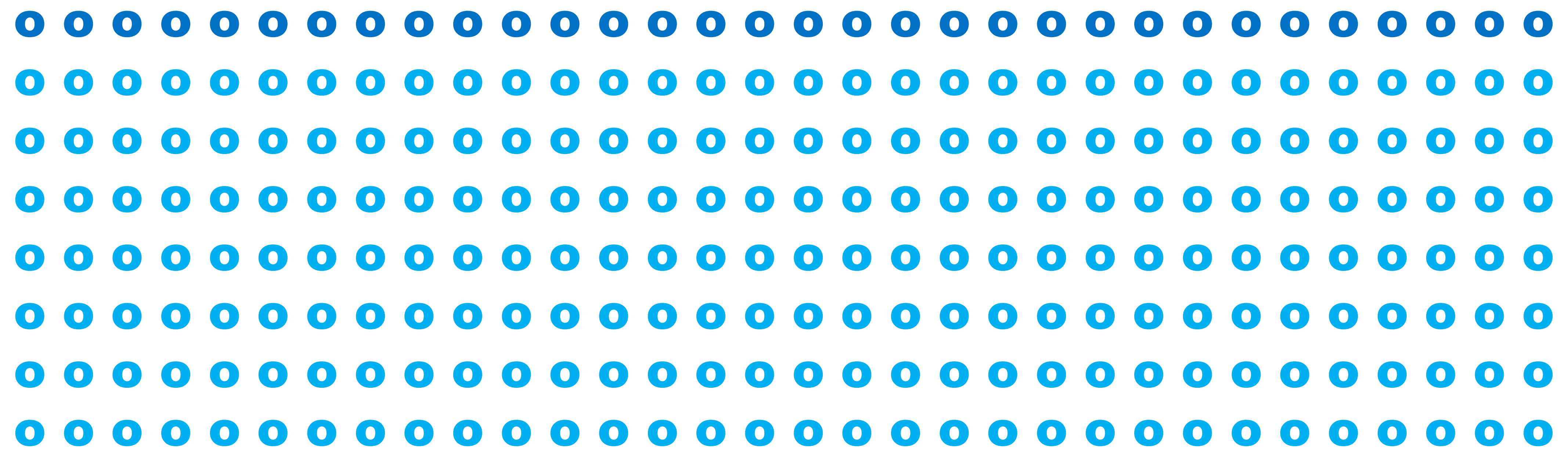
- High-level Parallelism still via MPI
- Performance improvements via manufacturing/process improvements, ILP, more registers, faster clocks.
- Vectorization of loops becomes important to take advantage of SIMD lanes
- Some programmers resort to SIMD intrinsics
- Still code to the main core/sequencer.

MULTICORE ARCHITECTURE WITH SIMD INSTRUCTIONS



- High-level Parallelism via MPI and perhaps OpenMP
- Clock rates begin to slow
- NUMA issues begin. libnuma/pthreads makes its way into the linux kernel
- Memory bandwidth does not keep up with compute speed
- Main memory has to be shared among the cores
- More and bigger caches

MANYCORE WITH AVX-512



- High-level Parallelism via MPI and perhaps OpenMP
- AVX-512 HW slow to take hold, initial implementations not optimal
- NUMA issues continue.
- Vectorizing compilers are still important
- SIMD intrinsics still in use
- Memory bandwidth does not keep up with compute speed
- Main memory has to be shared among the cores
- More and bigger caches
- SW grows in complexity, relies on features like dynamic memory allocation, large heaps, large data/call stacks.
- Still code to the main core/sequencer.

AMPERE A100

AMPERE A100 AND AVX-512 CPU

AMPERE HW CHARACTERISTICS

- High-level Parallelism via MPI and perhaps OpenMP
- Multiple CUDA contexts, CUDA streams, MIG, MPS allow sharing the GPU resource
- Synchronization between columns in the diagram is “hard”
- Offloading compilers are important, kernel scheduling, tuning and flexibility of launch parameters is key.

- Memory bandwidth is many times higher than a CPU
- Memory latency is high, caches are relatively small
- Programmer-managed shared memory (cache) is useful for performance and to communicate between cores in an SM
- Massively oversubscribing the cores is a key to performance

- Code to the core/lane. OS/low-level runtime handles divergence, at a cost
- Each core/lane loads and stores its own data. OS/low-level runtime ideally coalesces those into contiguous blocks

- Each core/lane has a small stack, limited number of registers, compared to a CPU core.
- Overheads can adversely affect performance since each core/lane only targets a small number of array elements

GENERAL RECOMMENDATIONS

https://docs.nvidia.com/cuda/ampere-tuning-guide/index.html

- 1.2. CUDA Best Practices

The performance guidelines and best practices described in the CUDA C++ Programming Guide and the CUDA C++ Best Practices Guide apply to all CUDA-capable GPU architectures. Programmers must primarily focus on following those recommendations to achieve the best performance.

The high-priority recommendations from those guides are as follows:

- Find ways to parallelize sequential code.
- Minimize data transfers between the host and the device.
- Adjust kernel launch configuration to maximize device utilization.
- Ensure global memory accesses are coalesced.
- Minimize redundant accesses to global memory whenever possible.
- Avoid long sequences of diverged execution by threads within the same warp.

PROGRAMMING THE NVIDIA PLATFORM

CPU, GPU, and Network

ACCELERATED STANDARD LANGUAGES

ISO C++, ISO Fortran, Python

```
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 cunumeric as np
…
def saxpy(a, x, y):
    y[:] += a*x
```

INCREMENTAL PORTABLE OPTIMIZATION

OpenACC, OpenMP

```
!$acc data copy(y(1:n)), copyin(x(1:n))
...
do concurrent (i = 1:n)
   y(i) = y(i) + a*x(i)
enddo
...
!$acc end data
```

```
!$omp target data map(tofrom:y), map(to:x)
...
do concurrent (i = 1:n)
   y(i) = y(i) + a*x(i)
enddo
...
!$omp end target data
```

PLATFORM SPECIALIZATION

CUDA

```
attributes(global) subroutine saxpy(n,a,x,y)
integer, value :: n
real, value :: a
real, device :: x(n), y(n)
i = (blockIdx.x-1)*blockDim.x + threadIdx.x
if (i.le.n) y(i) = y(i) + a*x(i)
end subroutine
```

```
real, device :: x(n), y(n)
...
!$cuf kernel do<<<*, 128>>>
do i = 1, n
  x(i) = real(i)
end do
y = 2.0
call saxpy<<<(N+255)/256,256>>>(n,a,x,y)
```

ACCELERATION LIBRARIES

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

FORTRAN DO CONCURRENT IS STANDARD FORTRAN

Background
Fortran introduced the ‘DO CONCURRENT’ construct in 2008. We assume the programmer guarantees that there are no dependencies between iterations so that we can run it in parallel on either a GPU or CPU.

```
# This option enables GPU offload
% nvfortran -stdpar source.f90
```

The syntax:

```
DO CONCURRENT (concurrent-header) [locality-spec]
   loop-body
END DO
```

where *locality-spec* is one of the following:

```
local(variable-name-list)
local_init(variable-name-list)
shared(variable-name-list)
default(none)
```

FORTRAN DO CONCURRENT IN MINI-WEATHER

use the local clause, similar to privatizing arrays in OpenACC and OpenMP

```
 !Compute fluxes in the x-direction for each cell
 do concurrent (k=1:nz, i=1:nx+1) local(d3_vals,vals,stencil,ll,s,r,u,t,p,w)
    !Use fourth-order interpolation from four cell averages to compute the
value at the interface in question
    do ll = 1 , NUM_VARS
      do s = 1 , sten_size
        stencil(s) = state(i-hs-1+s,k,ll)
      enddo
      !Fourth-order-accurate interpolation of the state
      vals(ll) = -stencil(1)/12 + 7*stencil(2)/12 + 7*stencil(3)/12 -
stencil(4)/12
      !First-order-accurate interpolation of the third spatial derivative of
the state (for artificial viscosity)
      d3_vals(ll) = -stencil(1) + 3*stencil(2) - 3*stencil(3) + stencil(4)
    enddo

    !Compute density, u-wind, w-wind, potential temperature, and pressure
(r,u,w,t,p respectively)
    r = vals(ID_DENS) + hy_dens_cell(k)
    u = vals(ID_UMOM) / r
    w = vals(ID_WMOM) / r
    t = ( vals(ID_RHOT) + hy_dens_theta_cell(k) ) / r
    p = C0*(r*t)**gamma

    !Compute the flux vector
    flux(i,k,ID_DENS) = r*u     - hv_coef*d3_vals(ID_DENS)
    flux(i,k,ID_UMOM) = r*u*u+p - hv_coef*d3_vals(ID_UMOM)
    flux(i,k,ID_WMOM) = r*u*w   - hv_coef*d3_vals(ID_WMOM)
    flux(i,k,ID_RHOT) = r*u*t   - hv_coef*d3_vals(ID_RHOT)
 enddo
```

```
Minfo Output:

compute_tendencies_x:

    253, Generating NVIDIA GPU code

        253, Loop parallelized across CUDA thread blocks,

                   CUDA threads(32) ! blockidx%x threadidx%x

             Loop parallelized across CUDA thread blocks,

                   CUDA threads(4) blockidx%y threadidx%y

        255, Loop run sequentially

        256, Loop run sequentially

    253, Local memory used for stencil,vals,d3_vals
```

FORTRAN DO CONCURRENT IN MINI-WEATHER

nvfortran supports the reduce clause starting with version 21.11

```fortran
do concurrent (k=1:nz, i=1:nx) reduce(+:mass,te)
   r  =   state(i,k,ID_DENS) + hy_dens_cell(k)             ! Density
   u  =   state(i,k,ID_UMOM) / r                           ! U-wind
   w  =   state(i,k,ID_WMOM) / r                           ! W-wind
   th = ( state(i,k,ID_RHOT) + hy_dens_theta_cell(k) ) / r ! Theta-temp
   p  = C0*(r*th)**gamma      ! Pressure
   t  = th / (p0/p)**(rd/cp)  ! Temperature
   ke = r*(u*u+w*w)           ! Kinetic Energy
   ie = r*cv*t                ! Internal Energy
   mass = mass + r            *dx*dz ! Accumulate domain mass
   te   = te   + (ke + r*cv*t)*dx*dz ! Accumulate domain total energy
enddo

call mpi_allreduce((/mass,te/),glob,2,MPI_REAL8,MPI_SUM,MPI_COMM_WORLD,ierr)
mass = glob(1)
te   = glob(2)
```

```
Minfo Output:

reductions:

    844, Generating NVIDIA GPU code

        844,   ! blockidx%x threadidx%x auto-collapsed

             Loop parallelized across CUDA thread blocks,

                  CUDA threads(128) collapse(2) ! blockidx%x threadidx%x

             Generating reduction(+:te,mass)
```

FORTRAN DO CONCURRENT CURRENT LIMITATIONS

- Do Concurrent requires function and subroutine calls to be pure

- We follow OpenACC and OpenMP defaults for scalars (first-private/local) and arrays (shared)
  - In fact, -stdpar currently enables OpenACC, is built on top of OpenACC.

- Do Concurrent lacks control over GPU scheduling which we have found useful
  - Forcing a “loop seq” inside the region
  - Offloading a serial kernel
  - No control equivalent to OpenACC’s gang, worker, vector, CUDA Fortran’s grid and block launch configurations

- Interoperability with CUDA is not all there yet
  - We still need to mark some useful device functions as pure (we do support CUDA atomics)
  - No control over the stream which the offloaded region runs on
  - Not interoperable yet with CUDA Fortran device attributed data

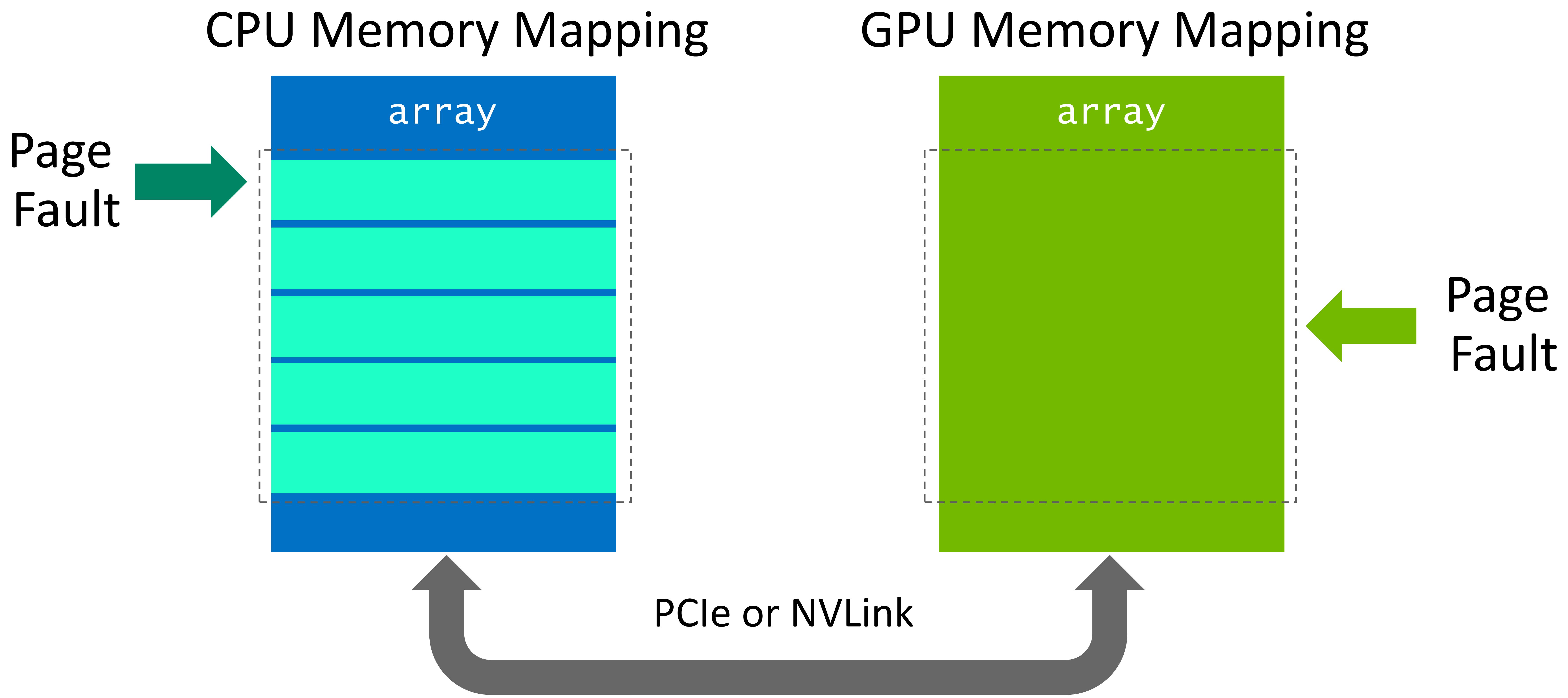
- The –stdpar option enables automatic use of managed memory

HOW CUDA UNIFIED MEMORY WORKS ON NVIDIA GPUs

Servicing CPU *and* GPU Page Faults for Allocatable Data

```
cudaMallocManaged(&array, size);
memset(array, size);
setValue<<<...>>>(array, size/2, 5);
...
```

```
__global__
void setValue(char *ptr, int index, char val)
{
  ptr[index] = val;
}
```

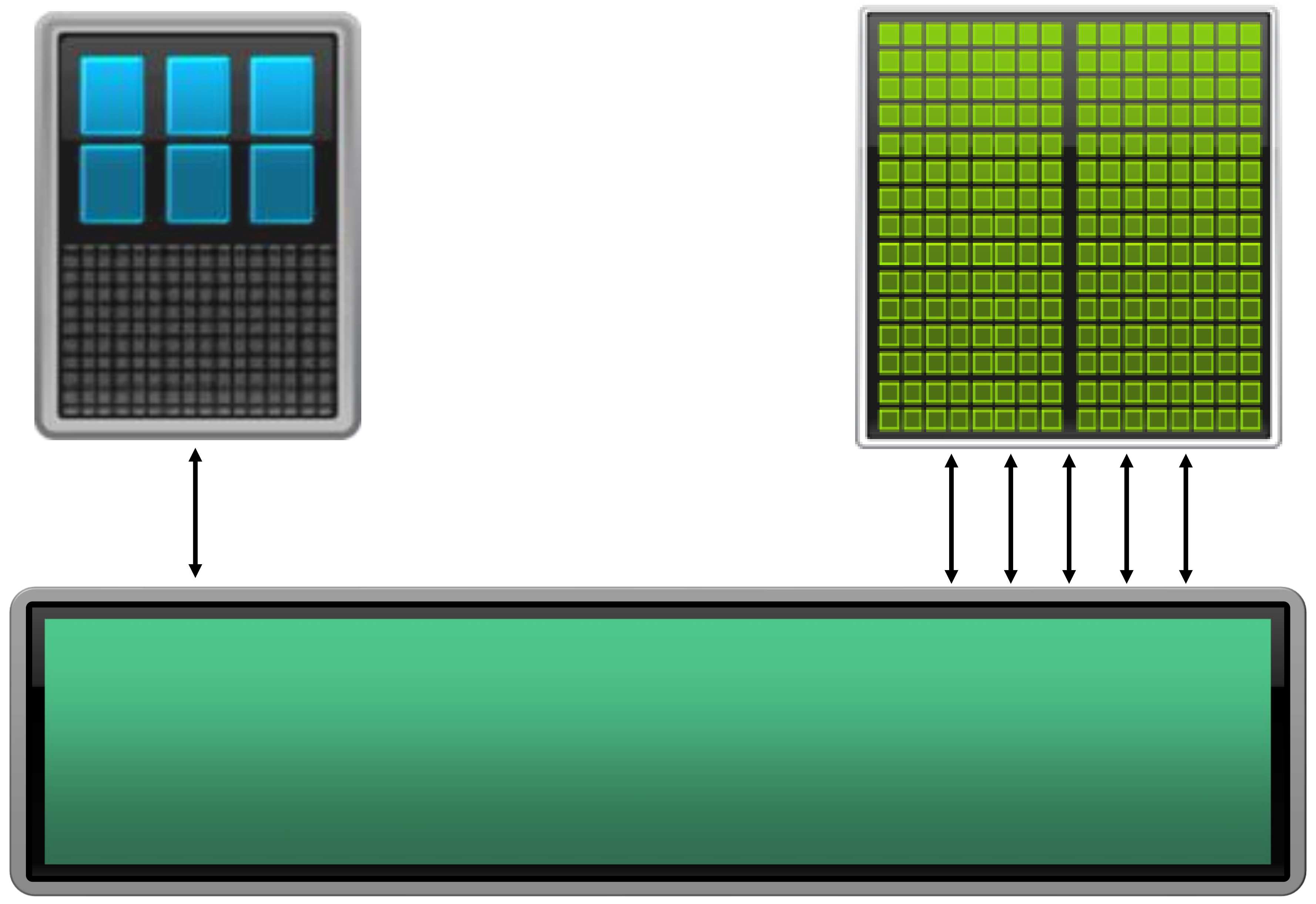


NVHPC STDPAR, OPENACC, OPENMP AND CUDA UNIFIED MEMORY

Compiling with the -gpu=managed option

```
#pragma acc data copyin(a,b) copyout(c)
{
 ...
 #pragma acc parallel
 {
 #pragma acc loop gang vector
     for (i = 0; i < n; ++i) {
         c[i] = a[i] + b[i];
         ...
     }
 }
 ...
}
```

GPU Developer View With CUDA Unified Memory



Unified Memory

C malloc, C++ new, Fortran allocate all mapped to CUDA Unified Memory

HPC PROGRAMMING IN ISO FORTRAN

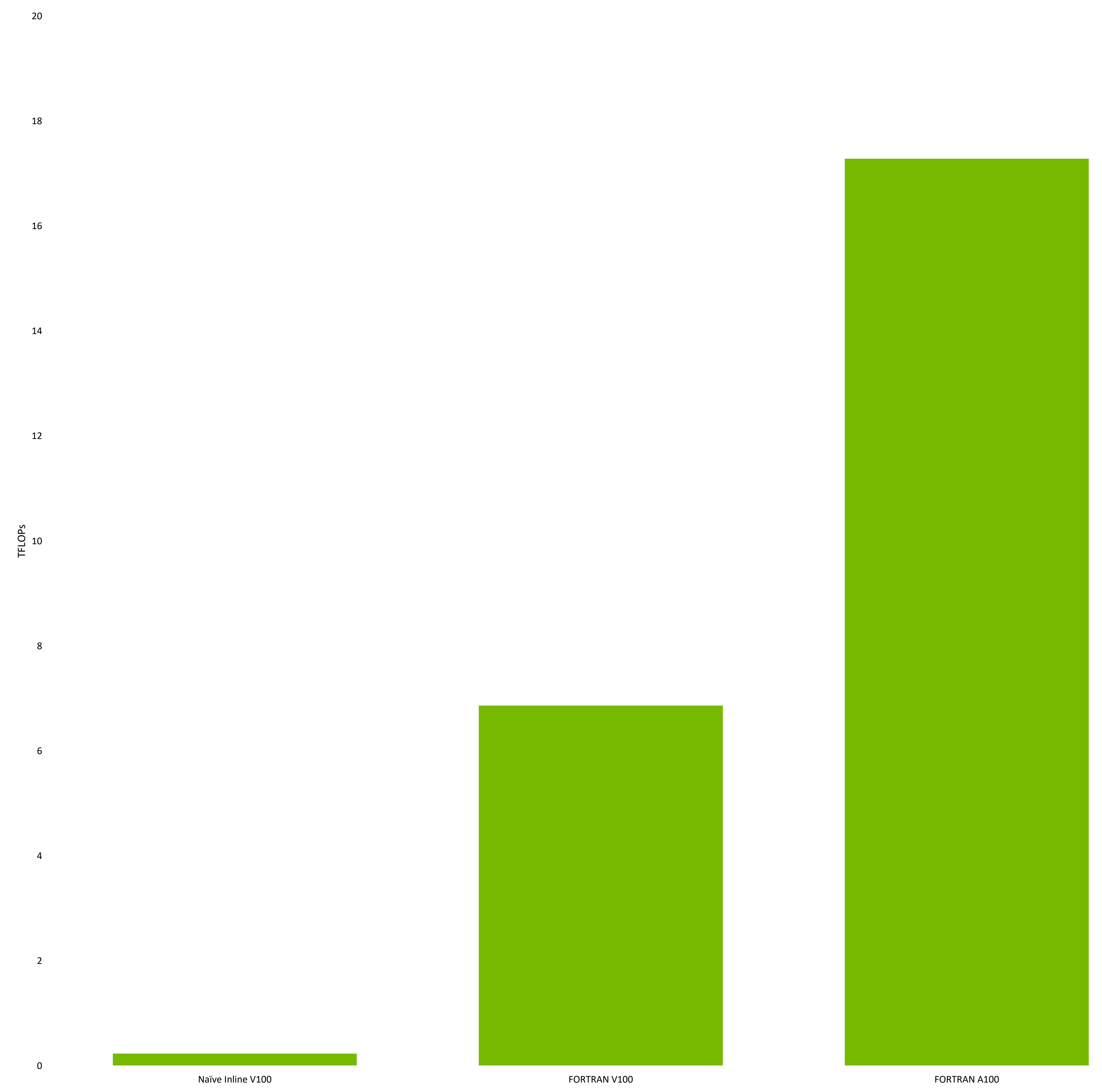
NVFORTRAN Accelerates Fortran Intrinsics with cuTENSOR Backend

```fortran
real(8), allocatable :: a(:,:)
real(8), allocatable :: b(:,:)
real(8), allocatable :: d(:,:)
!@cuf attributes(managed) :: a, b, d
. . .
allocate(a(ni,nk))
allocate(b(nk,nj))
allocate(d(ni,nj))
call random_number(a)
call random_number(b)
d = 0.0d0
do nt = 1, ntimes
  !$cuf kernel do(2) <<<*,*>>>
  do j = 1, nj
    do i = 1, ni
      do k = 1, nk
       d(i,j)= d(i,j) + a(i,k)*b(k,j)
      end do
    end do
  end do
end do
```

Inline FP64 matrix multiply

```fortran
!@cuf use cutensorex
real(8), allocatable :: a(:,:)
real(8), allocatable :: b(:,:)
real(8), allocatable :: d(:,:)
!@cuf attributes(managed) :: a, b, d
. . .
allocate(a(ni,nk))
allocate(b(nk,nj))
allocate(d(ni,nj))
call random_number(a)
call random_number(b)
d = 0.0d0
do nt = 1, ntimes
  d = d + matmul(a,b)
end do
```

MATMUL FP64 matrix multiply



MAPPING FORTRAN INTRINSICS TO CUTENSOR

Examples of Patterns Accelerated with cuTENSOR in HPC SDK since 20.7

```
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
```

https://developer.nvidia.com/blog/bringing-tensor-cores-to-standard-fortran/

NVLAMATH SIMPLIFIES FORTRAN SOLVER INTERFACES

CPU with LAPACK (OpenBLAS)

```fortran
...
real*8 , allocatable :: a(:,:)
integer, allocatable :: ipiv(:)
...
allocate(a(m,n), ipiv(m))
...
call dgetrf( m, n, a, lda, ipiv, info )
...
```

```
nvfortran –llapack -lblas
```

GFLOPs: ~496

GPU with cuSOLVER

```fortran
...
real*8 , allocatable :: a(:,:)
integer, allocatable :: ipiv(:)
integer :: istat, lwork
type( cusolverDnHandle ) :: handle
real, allocatable :: work(:)
integer :: devinfo(1)
...
allocate(a(m,n), ipiv(m))
...
istat = cusolverDnGetHandle( handle )
istat = cusolverDnDgetrf_bufferSize( handle, m,
n, a, lda, lwork )
allocate( work( lwork ) )
istat = cusolverDnDgetrf( handle, m, n, a, lda,
work, ipiv, devinfo(1) )
deallocate( work )
...
```

```
nvfortran –mp=gpu -gpu=managed
-cudalib=cusolver
```

GFLOPs: ~3238

GPU with NVLAmath

```fortran
...
real*8 , allocatable :: a(:,:)
integer, allocatable :: ipiv(:)
...
allocate(a(m,n), ipiv(m))
...
call dgetrf( m, n, a, lda, ipiv, info )
...
```

```
nvfortran –mp=gpu -gpu=managed
-cudalib=nvlamath
```

GFLOPs: ~3241

Matrix size: 20k x 20k
CPU: Xeon Gold 6148 w/ multi-threading; GPU: V100

FORTRAN STANDARD LANGUAGE POSSIBLE FUTURE WORK

- Add (non-standard, NVIDIA-specific) capabilities to DO CONCURRENT
- More F90 intrinsic function support in the vein of Matmul, Reshape, Spread, such as Pack and Merge
  - Requires some support for computing the mask argument efficiently
- Add more supported routines to NVLAMATH
  - Some new multi-gpu libraries might be wrapped under SCALAPACK or other interfaces
- Take advantage of new HW and SW Features

BASIC USE OF COMPUTE CONSTRUCTS IN OPENACC AND OPENMP

A smorgasbord; a plethora

```
! OpenMP
!$omp target teams loop collapse(2)
    do j=1,m-2
      do i=1,n-2
        Anew(i,j) = 0.25_fp_kind * ( A(i+1,j  ) + A(i-1,j  ) + &
                                     A(i  ,j-1) + A(i  ,j+1) )
        error = max( error, abs(Anew(i,j)-A(i,j)) )
      end do
    end do
```

```
!$omp target teams distribute map(tofrom:error)
    do j=1,m-2
      !$omp parallel do
      do i=1,n-2
        Anew(i,j) = 0.25_fp_kind * ( A(i+1,j  ) + A(i-1,j  ) + &
                                     A(i  ,j-1) + A(i  ,j+1) )
        !$omp atomic
        error = max( error, abs(Anew(i,j)-A(i,j)) )
      end do
    end do
```

```
!$omp target teams distribute reduction(max:error)
    do j=1,m-2
      !$omp simd reduction(max:error)
      do i=1,n-2
        Anew(i,j) = 0.25_fp_kind * ( A(i+1,j  ) + A(i-1,j  ) + &
                                     A(i  ,j-1) + A(i  ,j+1) )
        error = max( error, abs(Anew(i,j)-A(i,j)) )
      end do
    end do
```

```
! OpenACC
!$acc kernels loop
    do j=1,m-2
      do i=1,n-2
        Anew(i,j) = 0.25_fp_kind * ( A(i+1,j  ) + A(i-1,j  ) + &
                                     A(i  ,j-1) + A(i  ,j+1) )
        error = max( error, abs(Anew(i,j)-A(i,j)) )
      end do
    end do
```

```
!$acc parallel loop gang vector collapse(2) reduction(max:error)
    do j=1,m-2
      do i=1,n-2
        Anew(i,j) = 0.25_fp_kind * ( A(i+1,j  ) + A(i-1,j  ) + &
                                     A(i  ,j-1) + A(i  ,j+1) )
        error = max( error, abs(Anew(i,j)-A(i,j)) )
      end do
    end do
```

```
 51, Generating implicit reduction(max:error)
```

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)
```

DESCRIPTIVE OR PRESCRIPTIVE PARALLELISM

Prescriptive

Programmer explicitly parallelizes the code, compiler obeys

Requires little/no analysis by the compiler

Substantially different architectures require different directives

Fairly consistent behavior between implementations, though interpretations of simd/parallel differ

Descriptive

Compiler parallelizes the code with guidance from the programmer

Compiler must make decisions from available information

Compiler uses information from the programmer and heuristics about the architecture to make decisions

Quality of implementation greatly affects results.

PASSING DEVICE POINTERS TO CUDA LIBRARIES IN OPENACC AND OPENMP

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
 type(curandGenerator) :: g

 istat = curandCreateGenerator(g,CURAND_RNG_PSEUDO_XORWOW)

 !$acc data create(a)

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

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

 istat = curandCreateGenerator(g,CURAND_RNG_PSEUDO_XORWOW)

 !$omp target data map(alloc:a)

 !$omp target data use_device_ptr(a)
 istat = curandGenerate(g, a, N)
 !$omp end target data
```

CALLING USER ROUTINES IN DEVICE CODE

OpenACC is more explicit than OpenMP

```
! 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

 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

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

```
! OpenMP
 real function fs(a)
  !$omp declare target
  fs = a + 1.0
  end function

 subroutine fv(a,j,n)
  !$omp declare target
  real :: a(n,n)
  do i = 1, n
  !$omp parallel do
   a(i,j) = fs(a(i,j))
  enddo
 end subroutine

 subroutine fg(a,n)
  !$omp declare target
  real :: a(n,n)
  do j = 1, n
   call fv(a,j,n)
  enddo
 end subroutine
```

```
NVFORTRAN-F-1196-OpenMP - Standalone 'omp parallel' in a 'declare target'
routine is not supported yet.
NVFORTRAN/x86-64 Linux 21.11-0: compilation aborted
```

FORTRAN ARRAY SYNTAX IN DEVICE CODE

Currently not available in our OpenMP compiler, would require support for workshare in target regions

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

  istat = curandCreateGenerator(g,CURAND_RNG_PSEUDO_XORWOW)

  !$acc data create(a) copyout(h)

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

  !$acc kernels
  a = mod(abs(a),HN) + 1
  !$acc end kernels

  !$acc kernels
  h(:) = 0
  !$acc end kernels
```

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

  istat = curandCreateGenerator(g,CURAND_RNG_PSEUDO_XORWOW)

 !$omp target data map(alloc:a) map(from:h)

  !$omp target data use_device_ptr(a)
  istat = curandGenerate(g, a, N)
  !$omp end target data

  !$omp target teams loop
  do idum=1,1
  a = mod(abs(a),HN) + 1
  end do

  !$omp target teams loop
  do idum = 1, size(h)
    h(i) = 0
  end do
```

ASYNCHRONOUS BEHAVIOR, QUEUES, DEPENDENCIES, STREAMS

1-1 correspondence between OpenACC async numbers and streams. OpenMP uses dependencies.

```
! 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
```

```
! OpenMP
 !$omp target enter data map(alloc:a,b,c)

  ierr = cufftPlan2D(iplan1,n,m,CUFFT_C2C)
  nstream = omp_get_cuda_stream(omp_get_default_device(), .true.)
  ierr = cufftSetStream(iplan1,nstream)

  !$omp target update to(a) depend(out:nstream) nowait

  !$omp target data use_device_ptr(a,b,c)
  ierr = ierr + cufftExecC2C(iplan1,a,b,CUFFT_FORWARD)
  ierr = ierr + cufftExecC2C(iplan1,b,c,CUFFT_INVERSE)
  !$omp end target data

  ! scale c
  !$omp target teams distribute depend(inout:nstream) nowait
  do j = 1, n
    !$omp parallel do
    do i = 1, m
      c(i,j) = c(i,j) / (m*n)
    end do
  end do

  !$omp target update from(c) depend(in:nstream) nowait
  !$omp taskwait

  !$omp target exit data map(delete:a,b,c)
```

USING SHARED MEMORY FOR PERFORMANCE

```
! 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
```

CUDA FORTRAN

```
real, dimension(:,:) :: A, B, C
real, device, allocatable, dimension(:,:) ::
         Adev,Bdev,Cdev
. . .
allocate (Adev(N,M), Bdev(M,L), Cdev(N,L))
Adev = A(1:N,1:M)
Bdev = B(1:M,1:L)

call mm_kernel <<<dim3(N/16,M/16),dim3(16,16)>>>
               ( Adev, Bdev, Cdev, N, M, L )

C(1:N,1:L) = Cdev
deallocate ( Adev, Bdev, Cdev )

. . .
```

```
attributes(global) subroutine mm_kernel
            ( A, B, C, N, M, L )
real :: A(N,M), B(M,L), C(N,L), Cij
integer, value :: N, M, L
integer :: i, j, kb, k, tx, ty
real, shared :: Asub(16,16),Bsub(16,16)
tx = threadidx%x
ty = threadidx%y
i = (blockidx%x-1) * 16 + tx
j = (blockidx%y-1) * 16 + ty
Cij = 0.0
do kb = 1, M, 16
   Asub(tx,ty) = A(i,kb+ty-1)
   Bsub(tx,ty) = B(kb+tx-1,j)
   call syncthreads()
   do k = 1,16
      Cij = Cij + Asub(tx,k) * Bsub(k,ty)
   enddo
   call syncthreads()
enddo
C(i,j) = Cij
end subroutine mmul_kernel
```

CPU Code

GPU Code

CUDA FORTRAN

!@CUF for Portability

```fortran
module madd_device_module
!@cuf use cudafor
contains
  subroutine madd_dev(a,b,c,sum,n1,n2)
    real,dimension(:,:) :: a,b,c
!@cuf attributes(managed) :: a,b,c
    real :: sum
    integer :: n1, n2
    integer :: i, j
!$cuf kernel do (2) <<<(*,*),(32,4)>>>
    do j = 1,n2
      do i = 1,n1
        a(i,j) = b(i,j) + c(i,j)
        sum = sum + a(i,j)
      enddo
    enddo
  end subroutine
```

```fortran
module madd_device_module
  use cudafor
  implicit none
contains
  attributes(global) subroutine madd_kernel(a,b,c,blocksum,n1,n2)
    real, dimension(:,:) :: a,b,c
    real, dimension(:) :: blocksum
    integer, value :: n1,n2
    integer :: i,j,tindex,tneighbor,bindex
    real :: mysum
    real, shared :: bsum(256)
! Do this thread's work
    mysum = 0.0
    do j = threadidx%y + (blockidx%y-1)*blockdim%y, n2, blockdim%y*griddim%y
      do i = threadidx%x + (blockidx%x-1)*blockdim%x, n1, blockdim%x*griddim%x
        a(i,j) = b(i,j) + c(i,j)
        mysum = mysum + a(i,j) ! accumulates partial sum per thread
      enddo
    enddo
! Now add up all partial sums for the whole thread block
! Compute this thread's linear index in the thread block
! We assume 256 threads in the thread block
    tindex = threadidx%x + (threadidx%y-1)*blockdim%x
! Store this thread's partial sum in the shared memory block
    bsum(tindex) = mysum
    call syncthreads()
! Accumulate all the partial sums for this thread block to a single value
    tneighbor = 128
    do while( tneighbor >= 1 )
      if( tindex <= tneighbor ) &
        bsum(tindex) = bsum(tindex) + bsum(tindex+tneighbor)
      tneighbor = tneighbor / 2
      call syncthreads()
    enddo
! Store the partial sum for the thread block
    bindex = blockidx%x + (blockidx%y-1)*griddim%x
    if( tindex == 1 ) blocksum(bindex) = bsum(1)
  end subroutine

! Add up partial sums for all thread blocks to a single cumulative sum
  attributes(global) subroutine madd_sum_kernel(blocksum,dsum,nb)
    real, dimension(:) :: blocksum
    real :: dsum
    integer, value :: nb
    real, shared :: bsum(256)
    integer :: tindex,tneighbor,i
! Again, we assume 256 threads in the thread block
! accumulate a partial sum for each thread
    tindex = threadidx%x
    bsum(tindex) = 0.0
    do i = tindex, nb, blockdim%x
      bsum(tindex) = bsum(tindex) + blocksum(i)
    enddo
    call syncthreads()
! This code is copied from the previous kernel
! Accumulate all the partial sums for this thread block to a single value
! Since there is only one thread block, this single value is the final result
    tneighbor = 128
    do while( tneighbor >= 1 )
      if( tindex <= tneighbor ) &
        bsum(tindex) = bsum(tindex) + bsum(tindex+tneighbor)
      tneighbor = tneighbor / 2
      call syncthreads()
    enddo
    if( tindex == 1 ) dsum = bsum(1)
  end subroutine

  subroutine madd_dev(a,b,c,dsum,n1,n2)
    real, dimension(:,:), device :: a,b,c
    real, device :: dsum
    real, dimension(:), allocatable, device :: blocksum
    integer :: n1,n2,nb
    type(dim3) :: grid, block
    integer :: r
! Compute grid/block size; block size must be 256 threads
    grid = dim3((n1+31)/32, (n2+7)/8, 1)
    block = dim3(32,8,1)
    nb = grid%x * grid%y
    allocate(blocksum(1:nb))
    call madd_kernel<<< grid, block >>>(a,b,c,blocksum,n1,n2)
    call madd_sum_kernel<<< 1, 256 >>>(blocksum,dsum,nb)
    r = cudaThreadSynchronize() ! don't deallocate too early
    deallocate(blocksum)
  end subroutine
end module
```

SUMMARIZING: PROGRAMMING THE NVIDIA PLATFORM

CPU, GPU, and Network

ACCELERATED STANDARD LANGUAGES

ISO C++, ISO Fortran, Python

```
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 cunumeric as np
…
def saxpy(a, x, y):
    y[:] += a*x
```

INCREMENTAL PORTABLE OPTIMIZATION

OpenACC, OpenMP

```
!$acc data copy(y(1:n)), copyin(x(1:n))
...
do concurrent (i = 1:n)
   y(i) = y(i) + a*x(i)
enddo
...
!$acc end data
```

```
!$omp target data map(tofrom:y), map(to:x)
...
do concurrent (i = 1:n)
   y(i) = y(i) + a*x(i)
enddo
...
!$omp end target data
```

PLATFORM SPECIALIZATION

CUDA

```
attributes(global) subroutine saxpy(n,a,x,y)
integer, value :: n
real, value :: a
real, device :: x(n), y(n)
i = (blockIdx.x-1)*blockDim.x + threadIdx.x
if (i.le.n) y(i) = y(i) + a*x(i)
end subroutine
```

```
real, device :: x(n), y(n)
...
!$cuf kernel do<<<*, 128>>>
do i = 1, n
  x(i) = real(i)
end do
y = 2.0
call saxpy<<<(N+255)/256,256>>>(n,a,x,y)
```

ACCELERATION LIBRARIES

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