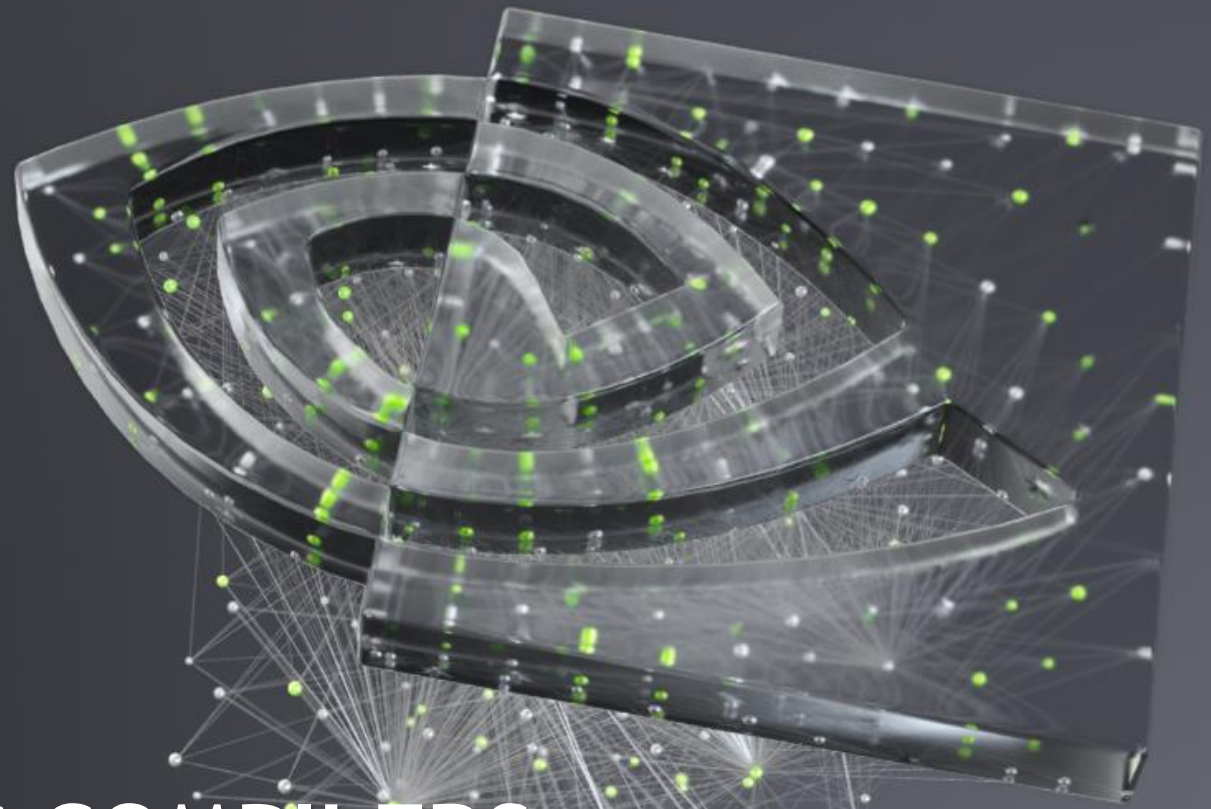


USING NVIDIA'S HPC COMPILERS (aka PGI)

August, 2020

Michael Wolfe – mwolfe@nvidia.com

GETTING STARTED WITH NVIDIA OpenACC

Download and Install

https://developer.nvidia.com/hpc-sdk

Install tree:

.../Linux_aarch64/20.7/compilers/bin/

.../Linux_ppc64le/20.7/compilers/bin/

.../Linux_x86_64/20.7/compilers/bin/

- includes .../20.7/comm_libs (MPI, NCCL)
- includes .../20.7/cuda (complete CUDA toolkit)
- includes .../20.7/math_libs (cuBLAS, cuFFT, cuSolver, cuSparse, cuTensor)
- includes .../20.7/profilers (NSIGHT Compute, NSIGHT Systems)

GETTING STARTED WITH NVIDIA OpenACC

NVIDIA Compiler Names (PGI names still work)

`nvc` – C compiler (fka pgcc)

`nvc++` – C++ compiler (fka pgc++)

`nvfortran` – Fortran compiler (fka pgfortran/pgf90/pgf95/pgf77)

Arm Linux, Power Linux, X86 Linux (X86 Windows in progress)

NVIDIA Kepler, Maxwell, Pascal, Volta, Ampere GPUs

GETTING STARTED WITH NVIDIA OpenACC

NVIDIA Compiler Options (PGI Options still work)

`-acc` – enable OpenACC directives

`-acc=multicore` – map OpenACC parallelism to a multicore CPU

`-acc=gpu` – map OpenACC parallelism to an NVIDIA GPU

`-acc=gpu -gpu=cudaX.Y` – compile using the CUDA X.Y toolchain

`-acc=gpu -gpu=ccWZ` – compile targeting compute capability WZ

`-gpu=pinned` – use CUDA pinned memory for all allocatables

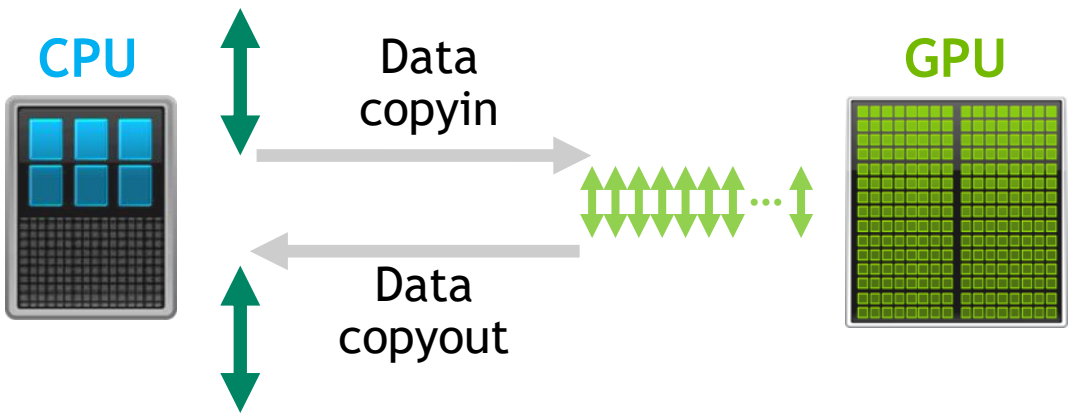
`-gpu=managed` – place all allocatables in CUDA Unified Memory

`-gpu=autocompare` – compare at copyout or update host

OPENACC DEBUGGING FEATURES

OpenACC AUTO-COMPARE

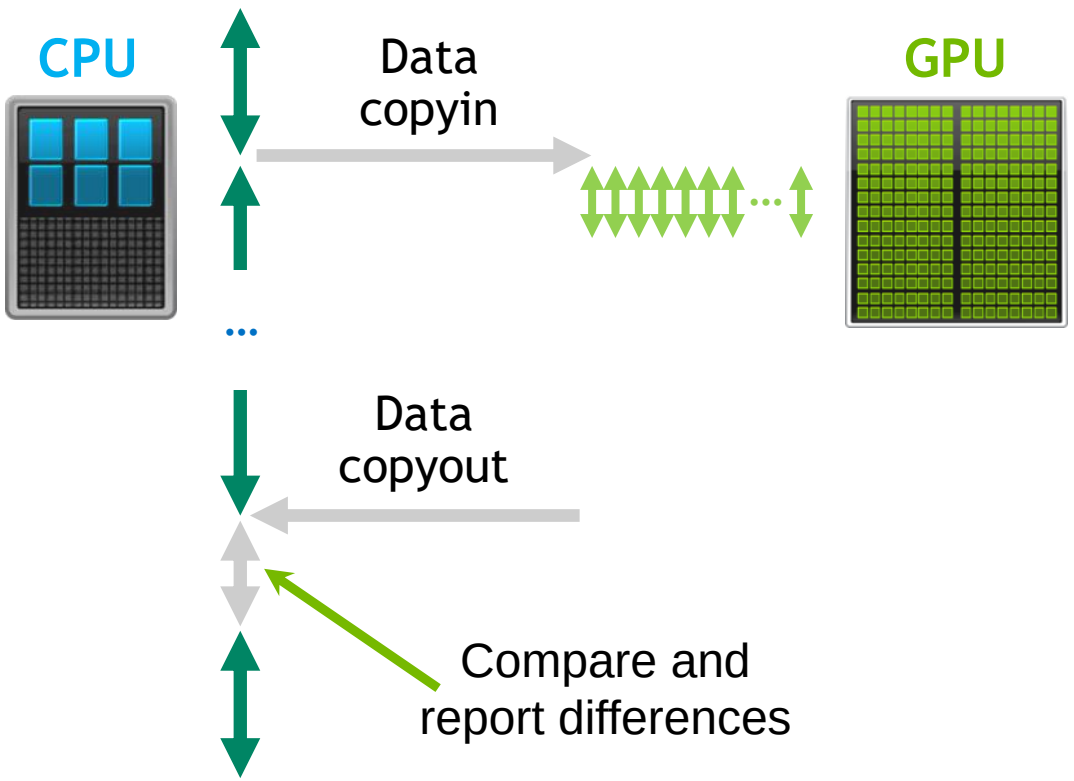
Find where CPU and GPU numerical results diverge



Normal OpenACC execution mode, no auto-compare

OpenACC AUTO-COMPARE

Find where CPU and GPU numerical results diverge



`-ta=tesla:autocompare`

Compute regions run redundantly on CPU and GPU

Results compared when data copied from GPU to CPU

OpenACC AUTO-COMPARE OUTPUT

Directives / environment variables to control fidelity

```
$ pgcc -ta=tesla:autocompare -o a.out example.c


$ PGI_COMPARE=summary,compare,abs=1 ./a.out
PCAST a1 comparison-label:0 Float
        idx: 0 FAIL ABS  act: 8.40187728e-01
exp: 1.00000000e+00 tol: 1.00000001e-01
        idx: 1 FAIL ABS  act: 3.94382924e-01
exp: 1.00000000e+00 tol: 1.00000001e-01
        idx: 2 FAIL ABS  act: 7.83099234e-01
exp: 1.00000000e+00 tol: 1.00000001e-01
        idx: 3 FAIL ABS  act: 7.98440039e-01
exp: 1.00000000e+00 tol: 1.00000001e-01
```

Test for program correctness, and determine points of divergence

Detect diverging results between CPU & GPU or multiple CPU code versions

Flexible calling and instrumentation options

PCAST: Parallel Compiler Assisted Software Testing

PCAST and OpenACC Auto-compare Summary

OpenACC, GPU to CPU comparison

`-ta=tesla,autocompare`, compare at `copyout` or `update host`

`-ta=tesla,redundant`, redundant execution

```
#pragma acc compare(a[0:n])
```

dynamically compare device-to-host data

CPU to CPU comparison

```
#pragma pgi compare(b[0:n])
```

save data to file, then to compare to saved data

SUPPORT FOR PRINTF() IN OpenACC

Debugging / tracing GPU-accelerated code regions

```
#pragma acc kernels copy(x[0:256])
{
  for (int i = 0; i < 256; i++) {
    if (x[i] < small) {
      printf("x at loc %03d is small: %d\n",i,x[i]);
      x[i] = small;
    } else if (x[i] > large) {
      printf("x at loc %03d is large: %d\n",i,x[i]);
      x[i] = large;
    }
 }
```

Support for formatted output using printf() statements in OpenACC compute regions

Debug and trace OpenACC programs during development and tuning

Works with both CPU and GPU

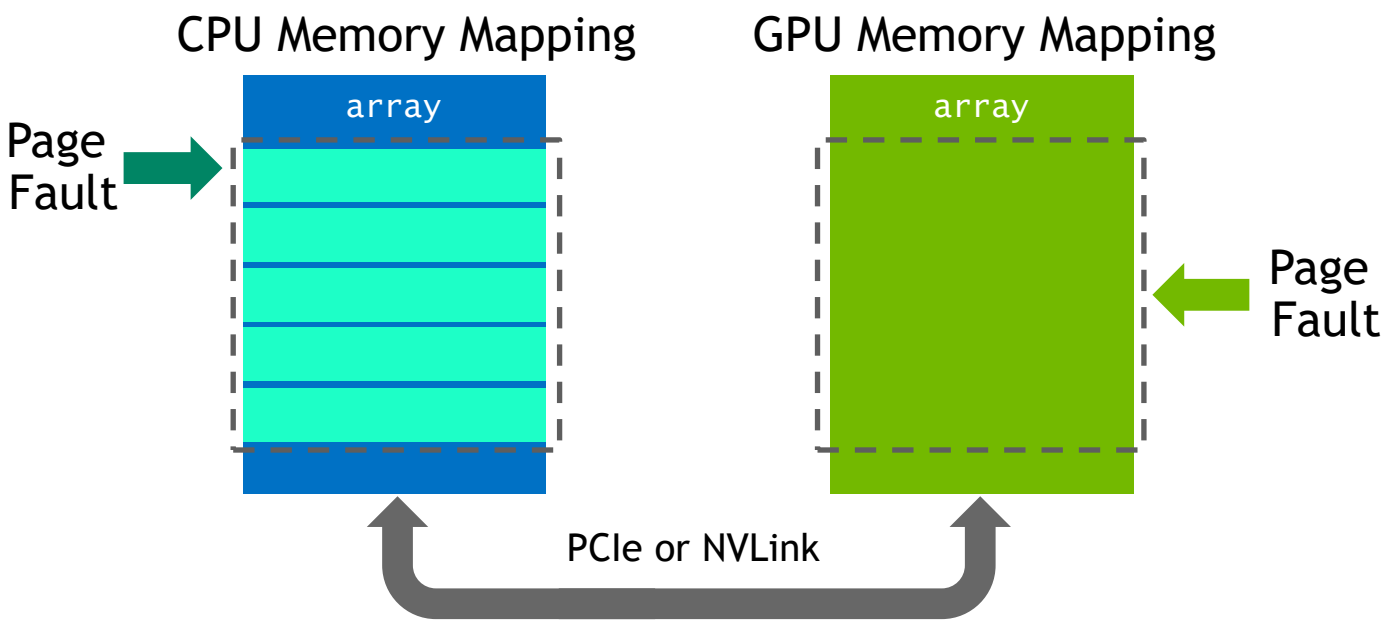
OPENACC AND CUDA UNIFIED MEMORY

HOW CUDA UNIFIED MEMORY WORKS

Servicing CPU and GPU Page Faults for Allocatable Data on Tesla GPUs

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

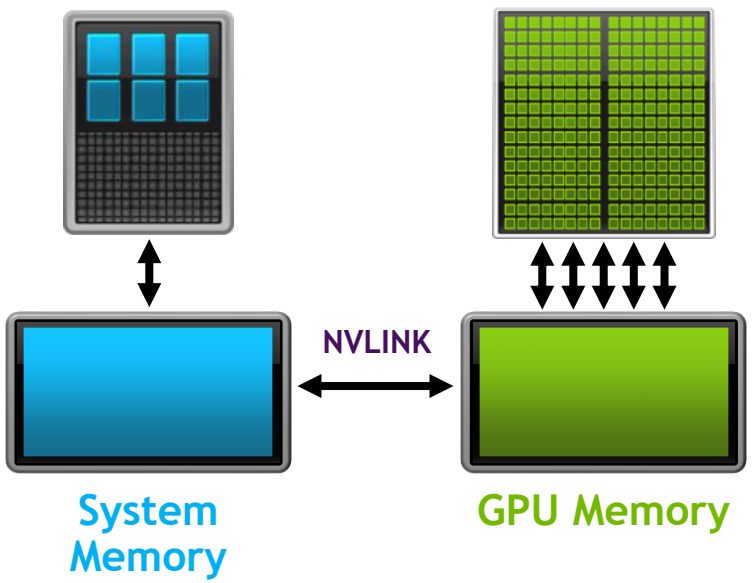
```
__global__
void setValue(char *ptr, int index, char val)
{
  ptr[index] = val;
}
```



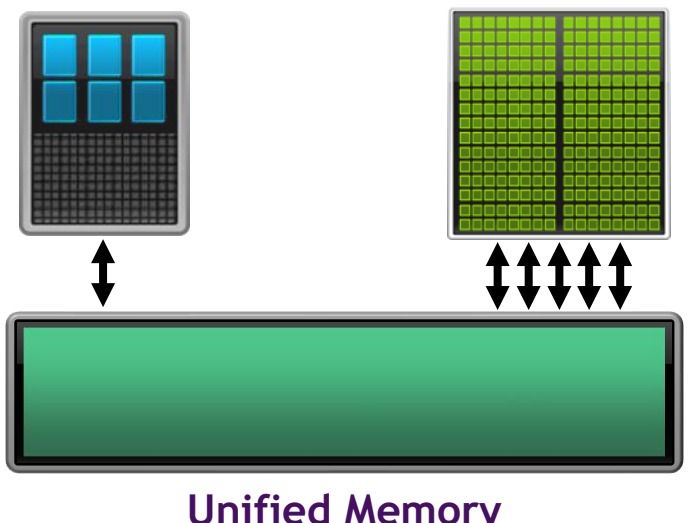
PROGRAMMING GPU-ACCELERATED SYSTEMS

CUDA Unified Memory for Dynamically Allocated Data

GPU Developer View



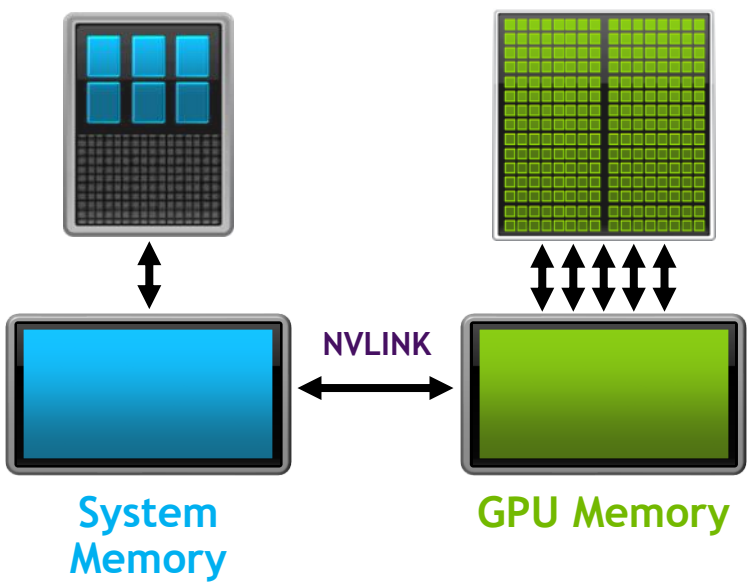
GPU Developer View With CUDA Unified Memory



COMPILING WITH -acc -ta=tesla

OpenACC data management under programmer and compiler control

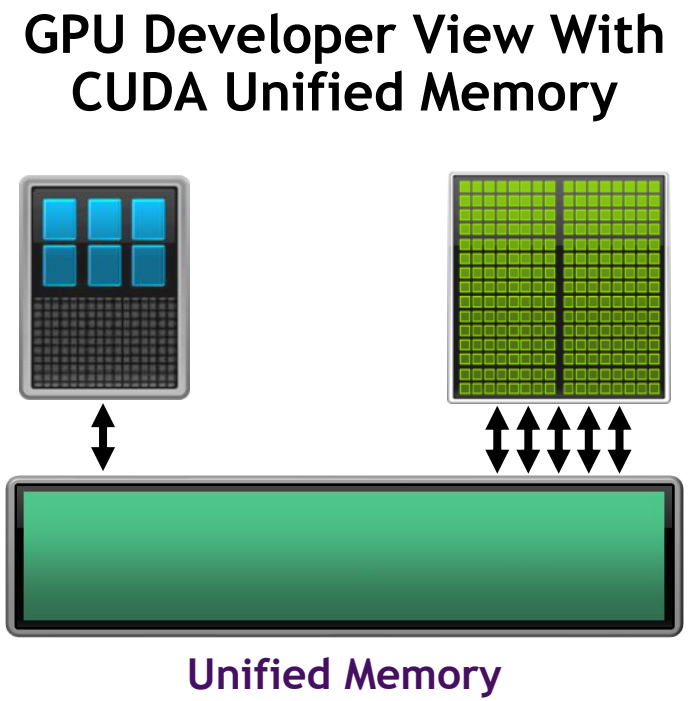
GPU Developer View




```
#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];
          ...
      }
  }
  ...
}
```

COMPILING WITH -acc -ta=tesla:managed

OpenACC data directives can be ignored on a unified memory system

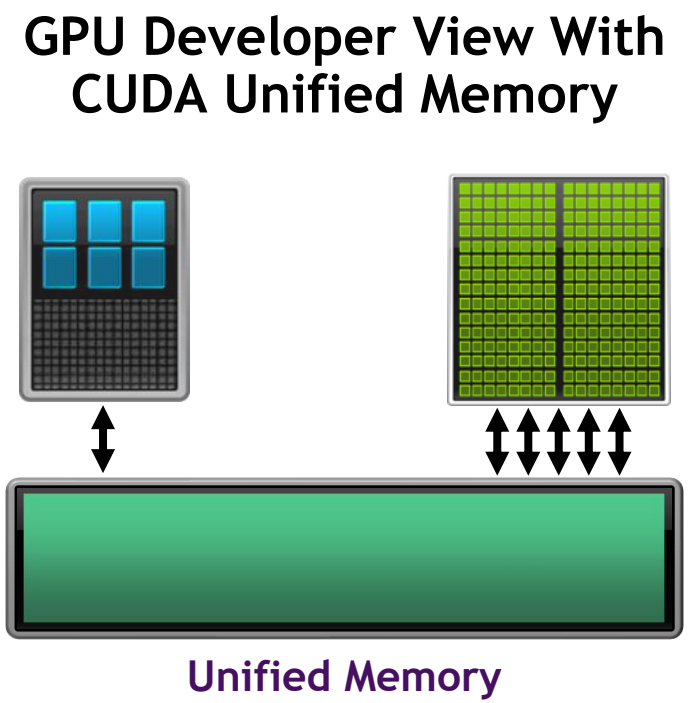



```
#pragma acc data copyin(a,b) copyout(c)
{
  ...
  #pragma acc parallel
  {
  #pragma acc loop gang vector
      for (i = 0; i < n; ++i) {
          c[i] = a[i] + b[i];
          ...
      }
  }
  ...
}
```

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

COMPILING WITH -acc -ta=tesla:managed

In fact you can leave them out entirely …

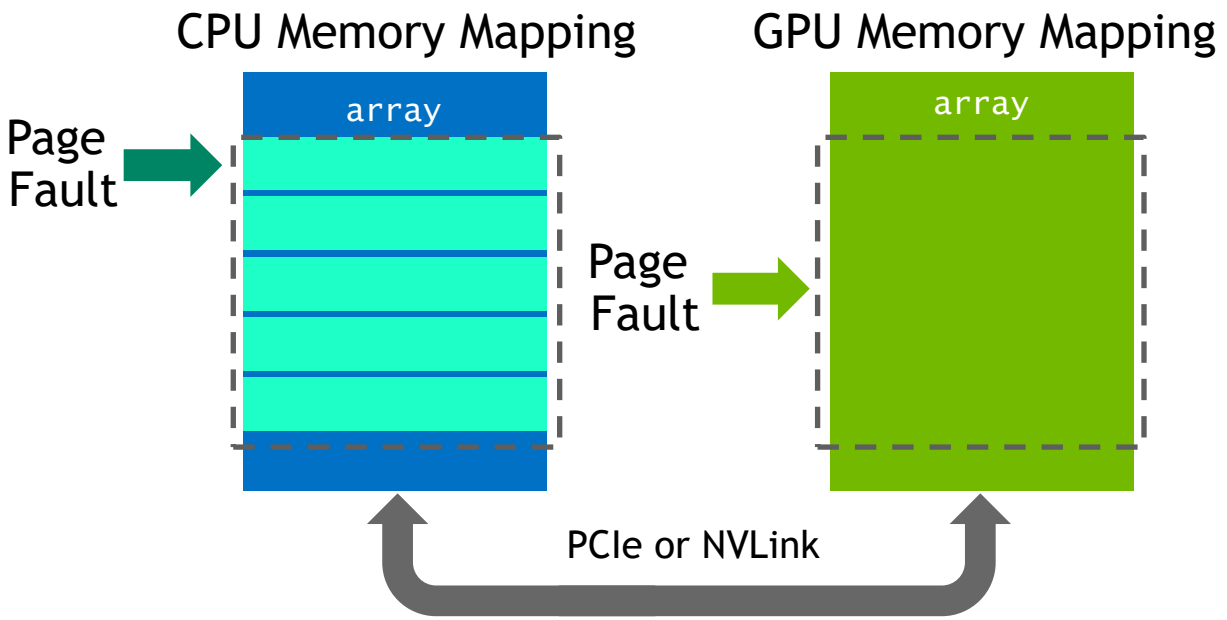



```
...
#pragma acc parallel
{
#pragma acc loop gang vector
    for (i = 0; i < n; ++i) {
        c[i] = a[i] + b[i];
        ...
    }
}
...
```

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

OpenACC AND CUDA UNIFIED MEMORY

Automatic data movement for allocatable objects




```
c = (int *)malloc(sizeof(int)*N);
for (i = 0; i < N; ++i) {
   c[i] = 0;
}
...

#pragma acc parallel
{
#pragma acc loop gang vector
  for (i = i1; i < i2; ++i) {
     c[i] = i;
  }
}
...
```

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

USING OpenACC AND UNIFIED MEMORY

Enabling Automatic Data Movement for Allocatable Data

- Compile with the -ta=tesla:managed option
  - All C malloc, C++ new, Fortran allocate data mapped into CUDA Unified Memory (no data directives needed)
- Compiler/Runtime moves most non-dynamic data implicitly
  - May need to add some data directives to the code
- Supported in Fortran, C and C++

USING OpenACC AND UNIFIED MEMORY

There are some limitations

- Places all dynamic data objects in CUDA Unified Memory, even those not used on the GPU
- Aggregates can’t include managed and non-managed members
- Global/static/stack data may require use of data directives
- Not all MPI implementations are Unified Memory-aware, or fully tuned for Unified Memory

OpenACC GPU PROGRAMMING IN 3 STEPS

PARALLELIZE

Parallelize with OpenACC for multicore CPUs

% pgc++ -ta=multicore …

```
while ( error > tol && ...
    error = 0.0;
#pragma acc parallel loop ...
    for( int j = 1; ...
#pragma acc loop
        for( int i = 1; ...
        ...
        }
...
```

ACCELERATE

Port to Tesla using OpenACC with CUDA Unified Memory

% pgc++ -ta=tesla:managed …

```
while ( error > tol && ...
    error = 0.0;
#pragma acc parallel loop ...
    for( int j = 1; ...
#pragma acc loop
        for( int i = 1; ...
        ...
        }
...
```

OPTIMIZE

Optimize and overlap data movement using OpenACC data directives

```
#pragma acc data create ...
while ( error > tol && ...
    error = 0.0;
#pragma acc parallel loop ...
    for( int j = 1; ...
#pragma acc loop
        for( int i = 1; ...
        ...
        }
...
```


CUDA FORTRAN

CUDA FORTRAN

```
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 )

. . .
```

CPU Code

```
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 * 16 + tx
j = blockidx%y * 16 + ty
Cij = 0.0
do kb = 1, M, 16
   Asub(tx,ty) = A(i,kb+tx-1)
   Bsub(tx,ty) = B(kb+ty-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
```

Tesla Code

CUDA FORTRAN

```
real(8), device, allocatable, dimension(:) :: Xd, Yd, Rd
real(8), pinned, allocatable, dimension(:) :: X, Y
real(8) :: R

. . .

allocate (Xd(N), Yd(N), Rd((N+127)/128))
Xd = X(:)
Yd = Y(:)

call daxpy1<<<(N+127)/128, 128>>>(Yd,A,Xd,N)

call dsum1<<<(N+127)/128, 128>>>(Yd,Rd,N)

call dsum1<<<1, 128>>>(Rd,Rd,(N+127)/128)

Y(:) = Yd
R = Rd(1)
deallocate (Xd, Yd, Rd)

. . .
```

CPU Code

```
attributes(global) subroutine daxpy1(Y,A,X,N)
real(8) :: Y(*), X(*)
integer, value :: N
real(8), value :: A
i = (blockidx%x-1) * blockdim%x + threadidx%x
if (i <= N) Y(i) = A * X(i) + Y(i)
end subroutine daxpy1

attributes(global) subroutine dsum1(Y,R,N)
real(8) :: Y(*), R(*)
integer, value :: N
real(8), shared :: part(128)
s = 0
j = (blockidx%x-1) * blockdim%x + threadidx%x
do i = j, N, blockdim%x * griddim%x
   s = s + Y(i)
enddo
j = threadidx%x ; k = blockdim%x
part(j) = s
do while (k > 1)
   k = k / 2
   if (j <= k) part(j) = part(j) + part(j+k)
   call syncthreads()
enddo
R(blockidx%x) = part(1)
end subroutine dsum1
```

NVIDIA GPU Code

CUDA FORTRAN

!$CUF KERNEL DO Directives

```
Real(8), device, allocatable, dimension(:) :: Xd, Yd
real(8), pinned, allocatable, dimension(:) :: X, Y
real(8) :: R

. . .



allocate (Xd(N), Yd(N))
Xd = X(:)
Yd = Y(:)
R = 0.0
!$cuf kernel do(1) <<<*,*>>>
do i = 1, n
 Yd(i) = A * Xd(i) + Yd(i)
 R = R + Yd(i)
enddo

Y(:) = Yd

deallocate (Xd, Yd)

. . .
```

CPU Code

Compiler Generated

NVIDIA GPU Code

CUDA FORTRAN

!@CUF Directives for portability

```
Real(8), allocatable, dimension(:) :: Xd, Yd
real(8), allocatable, dimension(:) :: X, Y
real(8) :: R
!@cuf attributes(device) :: Xd, Yd
!@cuf attributes(pinned) :: X, Y

. . .

allocate (Xd(N), Yd(N))
Xd = X(:)
Yd = Y(:)
R = 0.0
!$cuf kernel do(1) <<<*,*>>>
do i = 1, n
 Yd(i) = A * Xd(i) + Yd(i)
 R = R + Yd(i)
enddo

Y(:) = Yd

deallocate (Xd, Yd)

. . .
```

CPU Code

Compiler Generated

NVIDIA GPU Code

CUDA FORTRAN

!$CUF, !@CUF and managed data

```
real(8), allocatable, dimension(:) :: X, Y
real(8) :: R
!@cuf attributes(managed) :: X, Y

. . .

allocate (X(N), Y(N))

. . .

R = 0.0
!$cuf kernel do(1) <<<*,*>>>
do i = 1, n
 Y(i) = A * X(i) + Y(i)
 R = R + Y(i)
enddo

. . .

deallocate (X, Y)

. . .
```

CPU Code

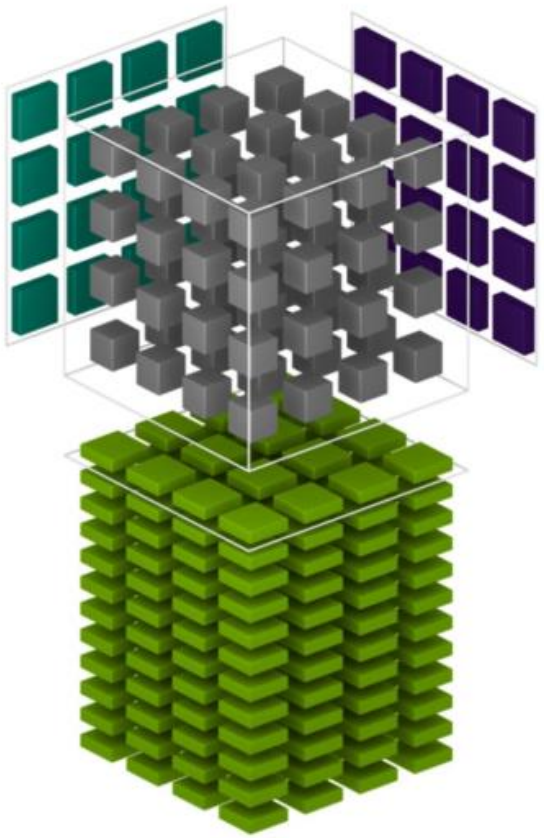
Compiler Generated

NVIDIA GPU Code

CUDA FORTRAN AND V100 TENSOR CORES

V100 TENSOR CORES

Fast matrix multiply using CUDA Fortran



Fast FP16 matrix multiply and accumulation into FP16 or FP32

8x to 16x faster than pure FP32 or FP64 in the same power envelope

Up to 125 TFlops FP16 performance

devblogs.nvidia.com/tensor-core-programming-using-cuda-fortran

CUDA FORTRAN TENSOR CORES

```
module mod1
use params ! Define matrix m, n, k
contains
  attributes(global) subroutine test1(a, b, c)
   use wmma
   real(2), device :: a(m, k), b(k, n)
   real(4), device :: c(m, n)
   WMMASubMatrix(WMMAMatrixA, 16, 16, 16, Real, WMMAColMajor):: sa
   WMMASubMatrix(WMMAMatrixB, 16, 16, 16, Real, WMMAColMajor):: sb
   WMMASubMatrix(WMMAMatrixC, 16, 16, 16, Real, WMMAKind4):: sc

   call wmmaLoadMatrix(sa, a(1,1), m)
   call wmmaLoadMatrix(sb, b(1,1), k)
   call wmmaMatmul(sc, sa, sb)
   call wmmaStoreMatrix(c(1,1), sc, m)
  end subroutine
end module
```

CUDA FORTRAN TENSOR CORES

```
subroutine volta(a,b,c)
use params  ! Define m, n and k
use mod1    ! Interface for test1

real(4) :: a(m,k), b(k,n), c(m,n)
real(2), device :: da(m,k), db(k,n)
real(4), device :: dc(m,n)

da = real(a,kind=2)
db = real(b,kind=2)
call test1<<<1,32>>>(da,db,dc)
c  = dc

end subroutine
```

Can we make this easier?

SIMPLIFYING ACCESS TO TENSOR CORES

Mapping Standard Fortran Intrinsics to cuTENSOR on V100

1x

```
real(4), dimension(ni,nk) :: a
real(4), dimension(nk,nj) :: b
real(4), dimension(ni,nj) :: c, d

...

!$acc enter data copyin(a,b,c)\
                 create(d)

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


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

Inline FP32 matrix multiply

4x

```
real(4), dimension(ni,nk) :: a
real(4), dimension(nk,nj) :: b
real(4), dimension(ni,nj) :: c, d

...

!$acc enter data copyin(a,b,c)\
                 create(d)
...

block
use cutensorEx
!$acc host_data use_device(a,b,c,d)

    d = c + matmul(a,b)

!$acc end host_data
end block

...

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

MATMUL FP32 matrix multiply

16x

```
real(2), dimension(ni,nk) :: a
real(2), dimension(nk,nj) :: b
real(2), dimension(ni,nj) :: c, d

...

!$acc enter data copyin(a,b,c)\
                 create(d)
...

block
use cutensorEx
!$acc host_data use_device(a,b,c,d)

    d = c + matmul(a,b)

!$acc end host_data
end block

...

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

MATMUL FP16 matrix multiply

MAPPING Fortran INTRINSICS TO cuTENSOR

Examples of Patterns Accelerated with cuTENSOR in PGI 20.4

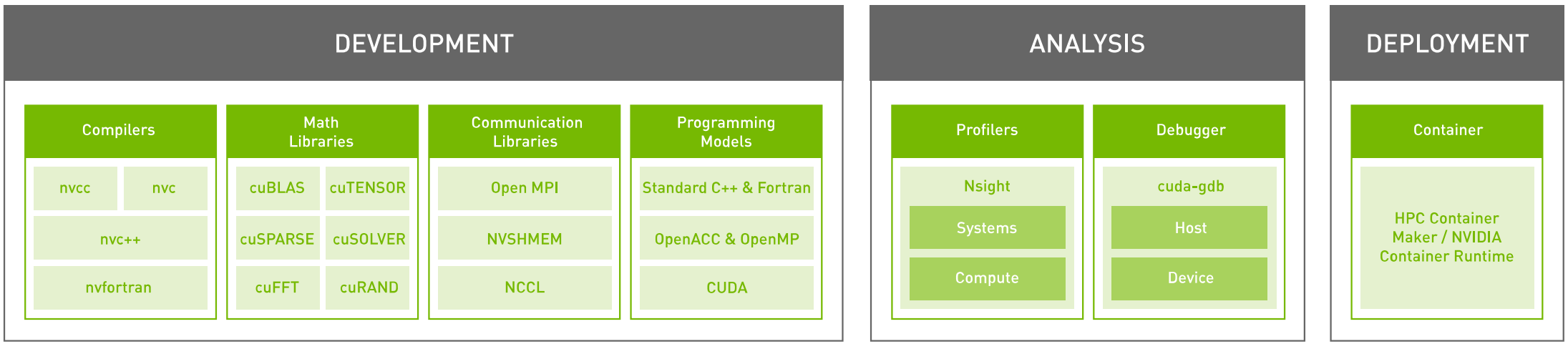
```
d = 2.5 * ceil(transpose(a)) + 3.0 * abs(transpose(b))
d = 2.5 * ceil(transpose(a)) + 3.0 * abs(b)
d = reshape(a,shape=[ni,nj,nk])
d = reshape(a,shape=[ni,nk,nj])
d = 2.5 * sqrt(reshape(a,shape=[ni,nk,nj],order=[1,3,2]))
d = alpha * conjg(reshape(a,shape=[ni,nk,nj],order=[1,3,2]))
d = reshape(a,shape=[ni,nk,nj],order=[1,3,2])
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
```


THE NVIDIA HPC SDK

THE NVIDIA HPC SDK

Comprehensive SDK for HPC developers, backward-compatible with PGI



Early Access Now, Production Release Q3'20

Replaces the PGI Compiler Suite Starting With the 20.7 Release

7-8 Releases per year, Cadence Similar to PGI Releases

NVIDIA GPU PROGRAMMING WITH STANDARD C++17

C++17 PARALLEL ALGORITHMS

Parallelism in Standard C++

- C++17 standardizes running STL algorithms in parallel, including on GPUs
- Insert `std::execution::par` as first parameter when calling algorithms
- Examples:

```
std::sort(std::execution::par, c.begin(), c.end());

double product = std::reduce(std::execution::par,
      first, last, 1.0, std::multiplies<double>());
```

C++ PROGRAMMING FOR NVIDIA GPUS

Math Libraries | Standard Languages | Directives | CUDA

C++17 Parallel Algorithms Available Now in the HPC SDK 20.5 EA

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

GPU Accelerated ISO C++

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

...

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

...

}
```

Incremental Performance Optimization with Directives

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

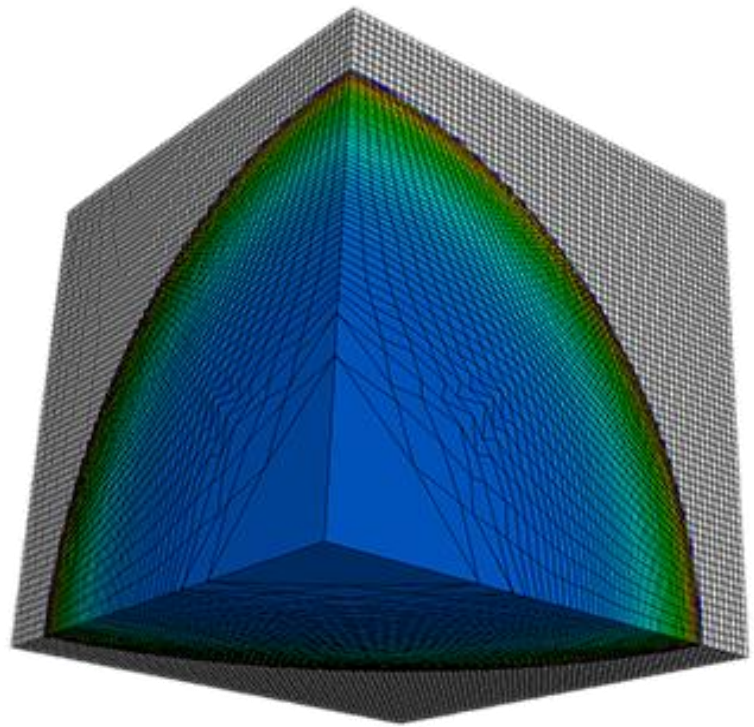
Maximize GPU Performance with CUDA C++/Fortran

GPU Accelerated Math Libraries

THE LULESH HYDRODYNAMICS MINI-APP

LLNL Hydrodynamics Proxy (Mini-App)

- ~9000 lines of C++
- Parallel versions in MPI, OpenMP, OpenACC, CUDA, RAJA, Kokkos, …
- Designed to stress compiler vectorization, parallel overheads, on-node parallelism



codesign.llnl.gov/lulesh

C++ WITH OpenACC/OpenMP PRAGMAS

LULESH CalcVelocityForNodes()

```
% nvc++ -fast -acc -gpu=managed -Minfo -c lulesh.cc
  ...
  CalcVelocityForNodes(Domain &, double, double, int):
  2103, Accelerator kernel generated
        Generating Tesla code
  2106, #pragma acc loop gang, vector(128) ! BlockIdx.x
                                           ! threadIdx.x
    ...
```

```
static inline
void CalcVelocityForNodes(Domain &domain, const Real_t dt,
                          const Real_t u_cut, Index_t numNode)
{
#pragma acc parallel loop
#pragma omp parallel for firstprivate(numNode)
   for ( Index_t i = 0 ; i < numNode ; ++i )
   {
     Real_t xdtmp, ydtmp, zdtmp ;

     xdtmp = domain.xd(i) + domain.xdd(i) * dt ;
     if( FABS(xdtmp) < u_cut ) xdtmp = Real_t(0.0);
     domain.xd(i) = xdtmp ;

     ydtmp = domain.yd(i) + domain.ydd(i) * dt ;
     if( FABS(ydtmp) < u_cut ) ydtmp = Real_t(0.0);
     domain.yd(i) = ydtmp ;

     zdtmp = domain.zd(i) + domain.zdd(i) * dt ;
     if( FABS(zdtmp) < u_cut ) zdtmp = Real_t(0.0);
     domain.zd(i) = zdtmp ;
   }
}
```

C++17 PARALLEL ALGORITHM: std::transform()

LULESH CalcVelocityForNodes()

```
% nvc++ -fast -stdpar -c lulesh.cc
  ...
```

- std::transform() wrapped around C++ lambdas
- No compiler feedback
- The C++17 parallel algorithms are a library, not a language
- Depends on CUDA Unified Memory

```
static inline
void CalcVelocityForNodes(Domain &domain, const Real_t dt,
                          const Real_t u_cut, Index_t numNode)
{
  std::transform(std::execution::par, domain.xd_begin(), domain.xd_end(),
                 domain.xdd_begin(), domain.xd_begin(),
                 [=](Real_t xd, Real_t xdd) {
                    Real_t xdnew = xd + xdd * dt;
                    if (std::abs(xdnew) < u_cut) xdnew = Real_t(0.0);
                    return xdnew;
                 });
  std::transform(std::execution::par, domain.yd_begin(), domain.yd_end(),
                 domain.ydd_begin(), domain.yd_begin(),
                 [=](Real_t yd, Real_t ydd) {
                    Real_t ydnew = yd + ydd * dt;
                    if (std::abs(ydnew) < u_cut) ydnew = Real_t(0.0);
                    return ydnew;
                  });
  std::transform(std::execution::par, domain.zd_begin(), domain.zd_end(),
                 domain.zdd_begin(), domain.zd_begin(),
                 [=](Real_t zd, Real_t zdd) {
                    Real_t zdnew = zd + zdd * dt;
                    if (std::abs(zdnew) < u_cut) zdnew = Real_t(0.0);
                    return zdnew;
                 });
}
```

```
static inline
void CalcHydroConstraintForElems(Domain &domain, Index_t length,
                                 Index_t *regElemlist, Real_t dvovmax, Real_t& dthydro)
{
#if _OPENMP
   const Index_t threads = omp_get_max_threads();
   Index_t hydro_elem_per_thread[threads];
   Real_t dthydro_per_thread[threads];
#else
   Index_t threads = 1;
   Index_t hydro_elem_per_thread[1];
   Real_t  dthydro_per_thread[1];
#endif
#pragma omp parallel firstprivate(length, dvovmax)
   {
      Real_t dthydro_tmp = dthydro ;
      Index_t hydro_elem = -1 ;
#if _OPENMP
      Index_t thread_num = omp_get_thread_num();
#else
      Index_t thread_num = 0;
#endif
#pragma omp for
      for (Index_t i = 0 ; i < length ; ++i) {
         Index_t indx = regElemlist[i] ;

         if (domain.vdov(indx) != Real_t(0.)) {
            Real_t dtdvov = dvovmax / (FABS(domain.vdov(indx))+Real_t(1.e-20)) ;

            if ( dthydro_tmp > dtdvov ) {
                  dthydro_tmp = dtdvov ;
                  hydro_elem = indx ;
            }
         }
      }
      dthydro_per_thread[thread_num]    = dthydro_tmp ;
      hydro_elem_per_thread[thread_num] = hydro_elem ;
   }
   for (Index_t i = 1; i < threads; ++i) {
      if(dthydro_per_thread[i] < dthydro_per_thread[0]) {
         dthydro_per_thread[0]    = dthydro_per_thread[i];
         hydro_elem_per_thread[0] =  hydro_elem_per_thread[i];
      }
   }
   if (hydro_elem_per_thread[0] != -1) {
      dthydro =  dthydro_per_thread[0] ;
   }
   return ;
}
```

C++ with OpenMP

PARALLEL C++17

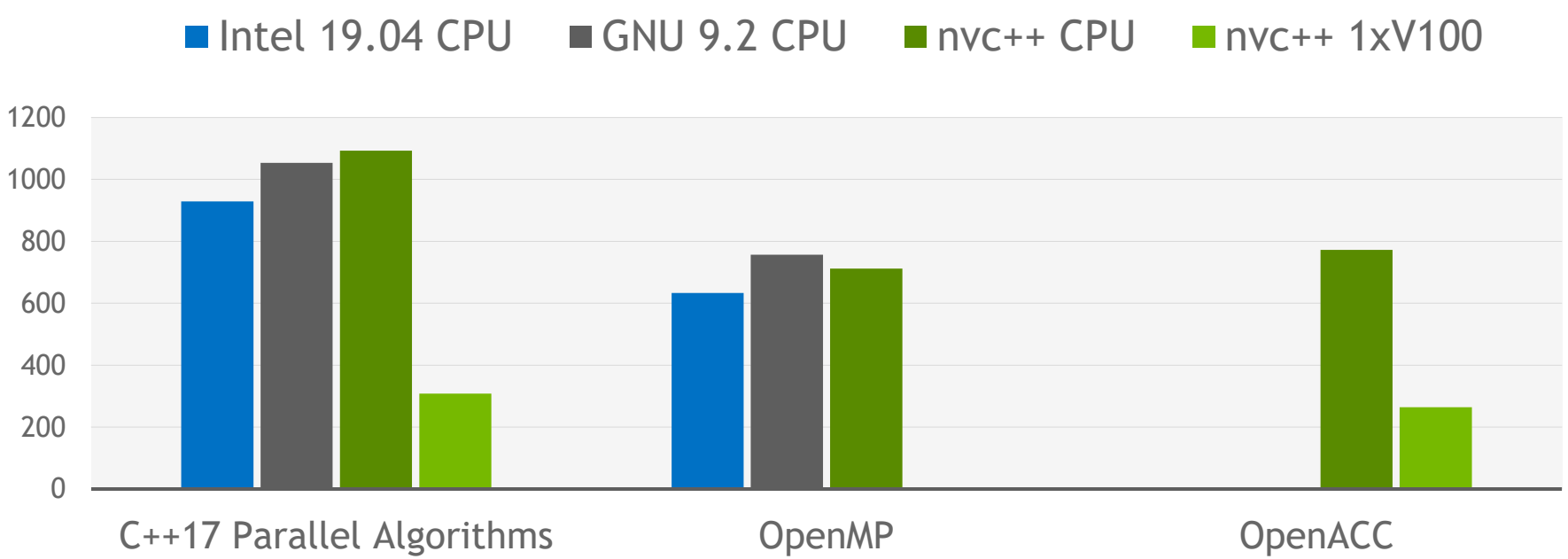
- Composable, compact and elegant
- Easy to read and maintain
- ISO Standard
- Portable – pgc++, g++, icpc, vc++, …

```
static inline void CalcHydroConstraintForElems(Domain &domain, Index_t length,
                                               Index_t *regElemlist,
                                               Real_t dvovmax,
                                               Real_t &dthydro) {
  dthydro = std::transform_reduce(
      std::execution::par, counting_iterator(0), counting_iterator(length),
      dthydro, [](Real_t a, Real_t b) { return a < b ? a : b; },
      [=, &domain](Index_t i) {
        Index_t indx = regElemlist[i];
        if (domain.vdov(indx) == Real_t(0.0)) {
          return std::numeric_limits<Real_t>::max();
        } else {
          return dvovmax / (std::abs(domain.vdov(indx)) + Real_t(1.e-20));
        }
      });
}
```

Parallel C++17

LULESH 150^3 PERFORMANCE

Time in Seconds – Smaller is Better



CPU: Two 20 core Intel Xeon Gold 6148 CPUs @ 2.4GHz w/376GB memory, hyperthreading enabled.
GPU: NVIDIA Tesla V100-PCIe-16GB GPU @ 1.53GHz.

See the SC19 talk at nvidia.com/sc19:
"GPU Programming with Standard C++17"

HPC PROGRAMMING IN 2020 AND BEYOND

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
ISO C++ and Fortran

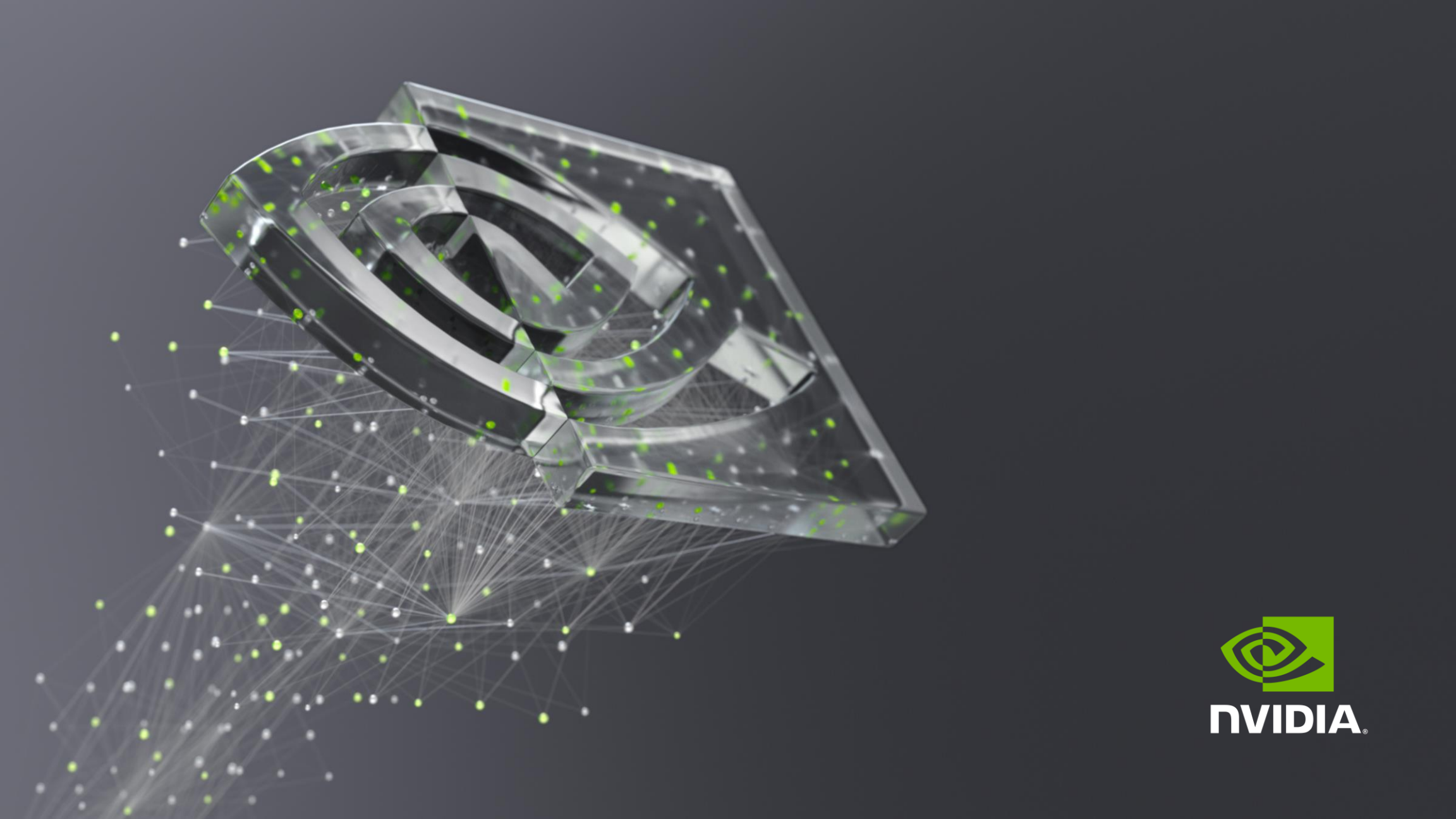
```
#pragma acc data copy(x,y)
{

...

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

...

}
```

Incremental Performance
Optimization with Directives

```
__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

GPU Accelerated Math Libraries


NVIDIA

GETTING STARTED WITH THE NVIDIA HPC SDK

Fortran/C++/C Key Compiler Options

`-fast` – enable optimization for multicore CPUs including SIMD vectorization

`-acc` – enable OpenACC directives, default target is NVIDIA GPU

`-mp` – enable OpenMP directives, default (and currently only) target is multicore CPU

`-stdpar` – enable auto-acceleration of C++ Parallel Algorithms on NVIDIA GPUs

`-gpu=…` – compilation controls for NVIDIA GPU targets

GETTING STARTED WITH OpenACC

NVIDIA HPC SDK Fortran/C++/C Key Compiler Options

`-acc` – enable OpenACC directives, default target is NVIDIA GPU

`-acc=multicore` – map OpenACC parallelism to a multicore CPU

`-gpu=cudaX.Y` – compile using the CUDA X.Y toolchain

`-gpu=ccWZ` – compile targeting compute capability WZ

`-gpu=pinned` – use CUDA pinned memory for all allocatables

`-gpu=managed` – place all allocatables in CUDA Unified Memory

`-gpu=autocompare` – compare at copyout or update host

`-gpu=redundant` – force redundant execution on CPU and GPU