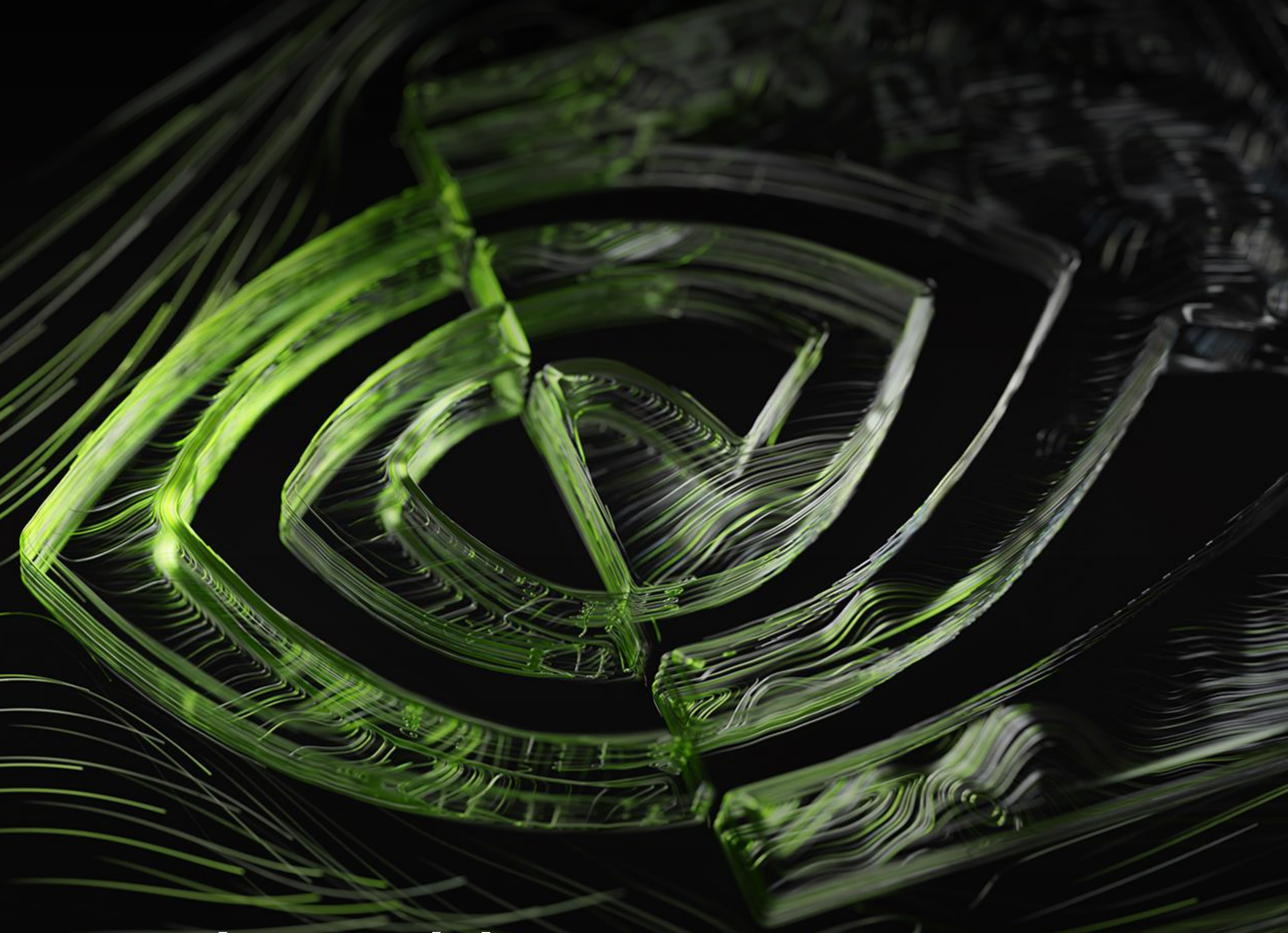
NVIDIA



Overview of GPU Programming Models

Jeff Larkin, Principal HPC Application Architect; October 2022

Goals of this Session

- This session will provide a brief description of several GPU programming models
- It is not a tutorial, but simply scratches the surface
- Where possible, I have linked out to resources for more information.
- This is not a complete survey of all possible GPU programming models

Agenda

- Programming the NVIDIA Platform
- Standard Language Approaches
- Compiler Directives
- Python Approaches
- CUDA C++ and Fortran
- Conclusions and Additional Resources

Programming the NVIDIA Platform

GPU Computing in a Nutshell

All GPU programming models follow this pattern

Program flow and resource allocation is managed by the CPU

Data & GPU “kernels” offload to the GPU

Rest of Sequential CPU Code

GPU

Parallelize using CUDA Programming Model

CPU

CPU & GPU work together

Programming the NVIDIA Platform

CPU, GPU, and Network

ACCELERATED STANDARD LANGUAGES

ISO C++, ISO Fortran

```
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

```
#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

CUDA

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

ACCELERATION LIBRARIES

Core

Math

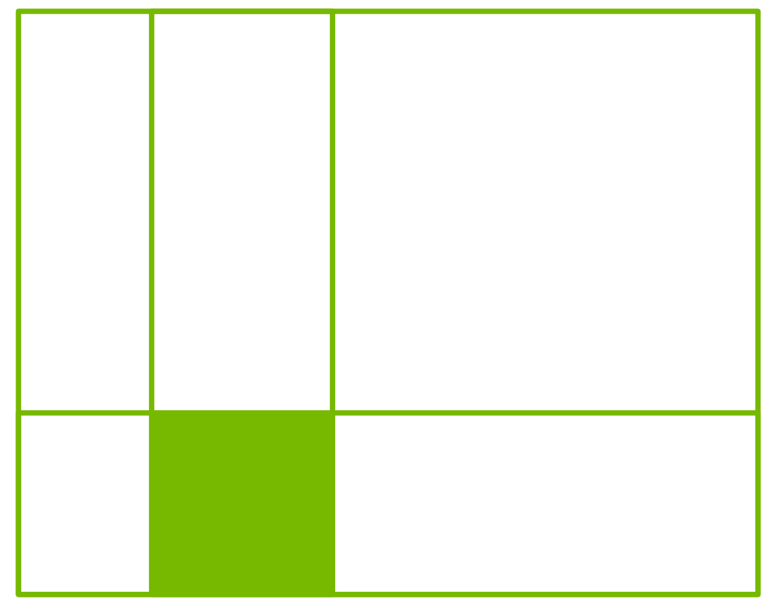
Communication

Data Analytics

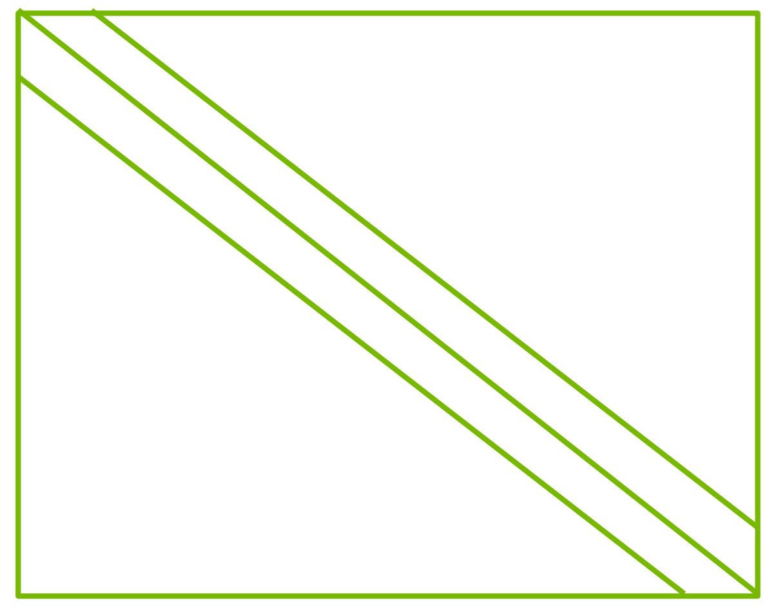
AI

Quantum

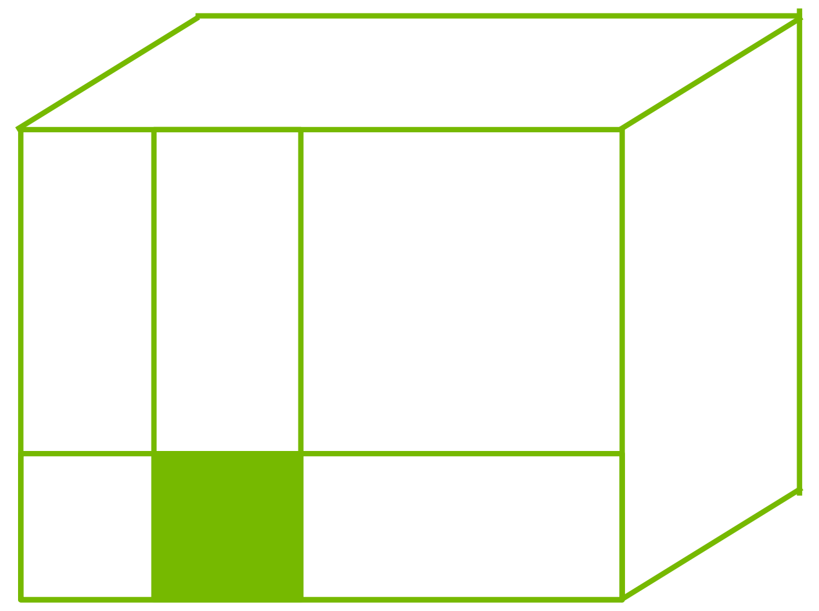
NVIDIA Math Libraries



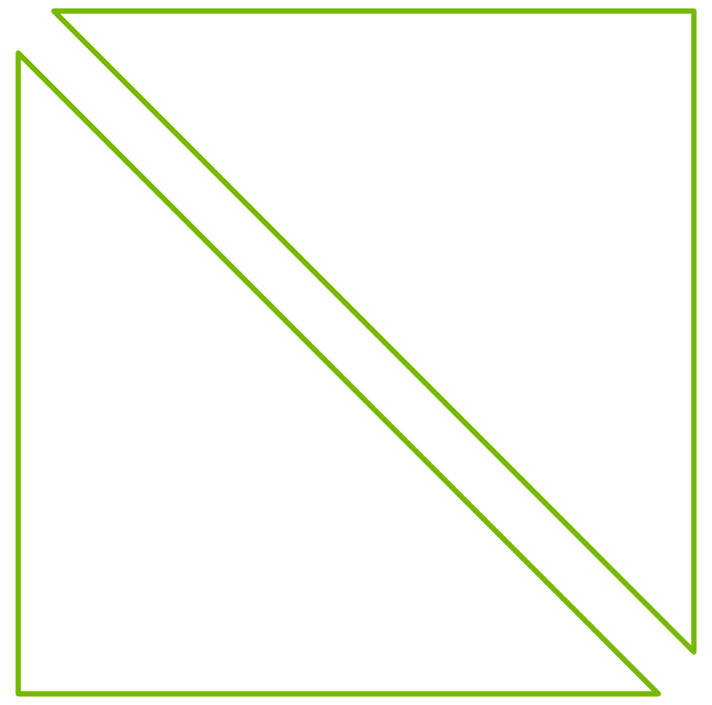
cuBLAS



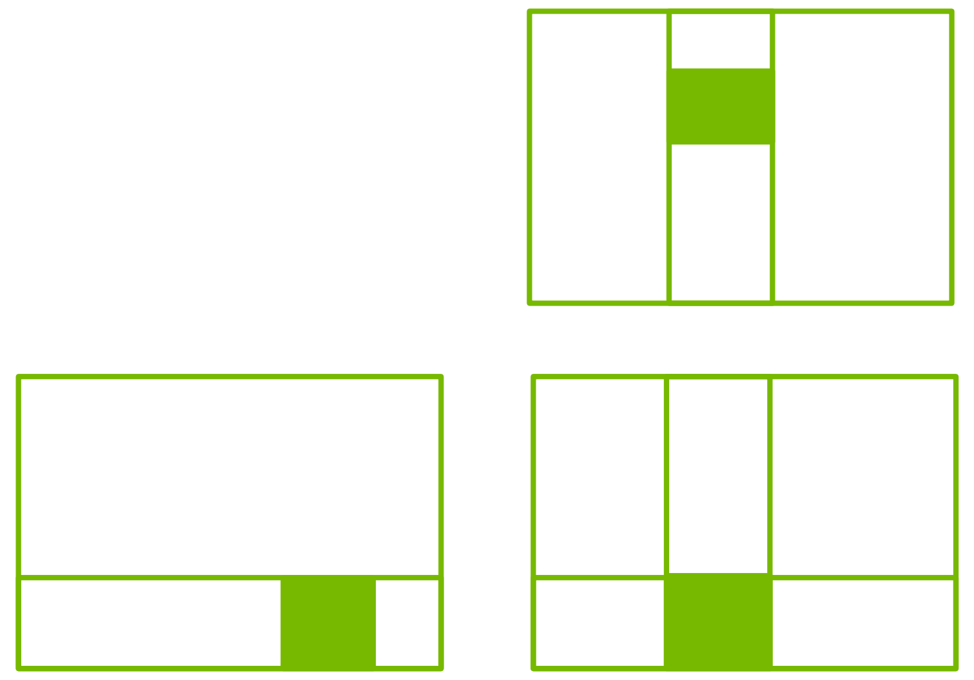
cuSPARSE



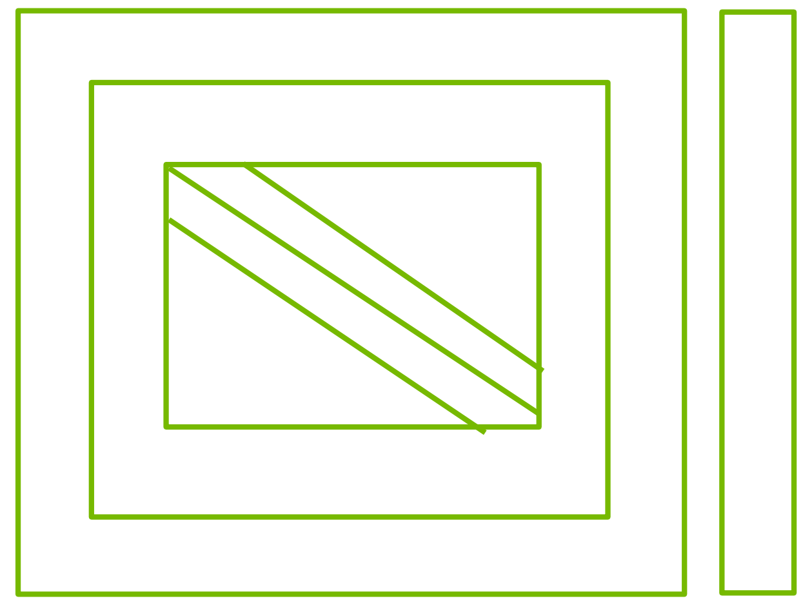
cuTENSOR



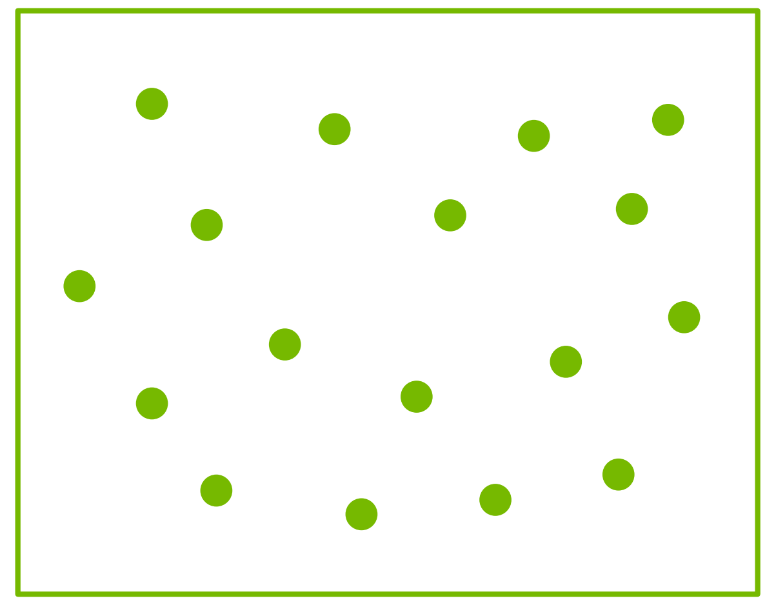
cuSOLVER



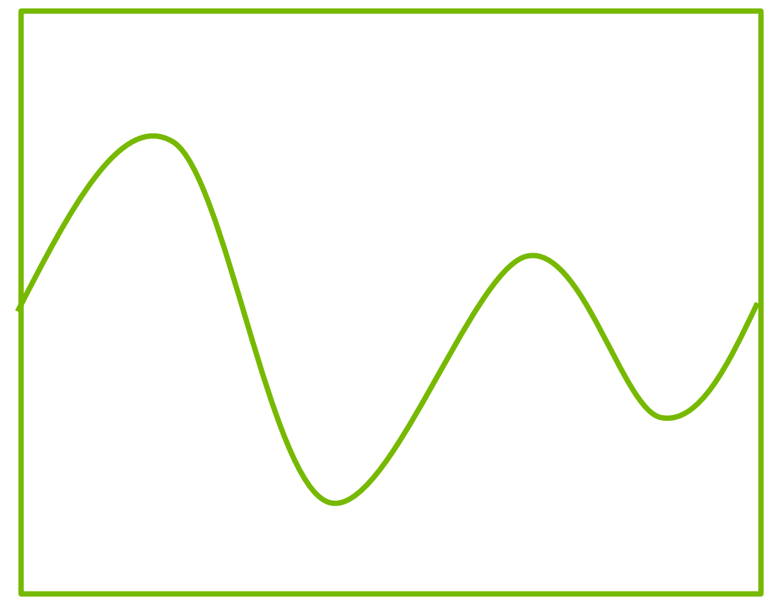
CUTLASS



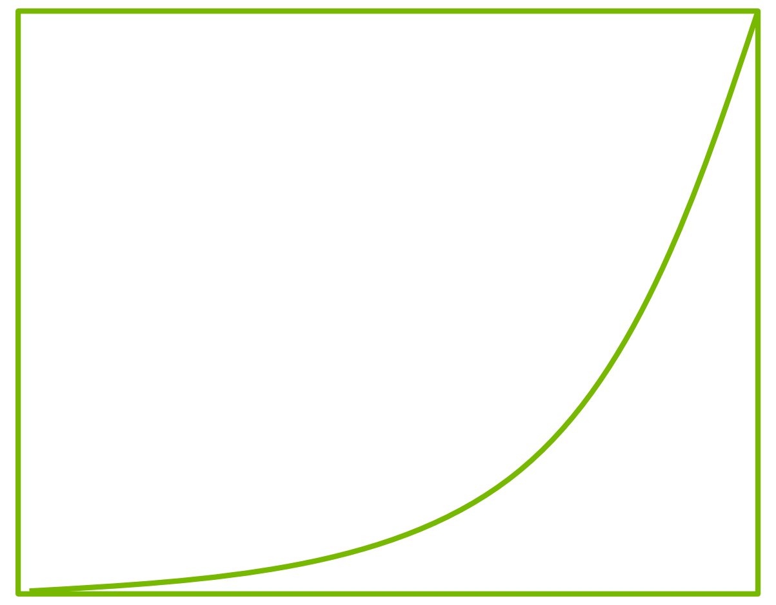
AMGX



cuRAND



cuFFT



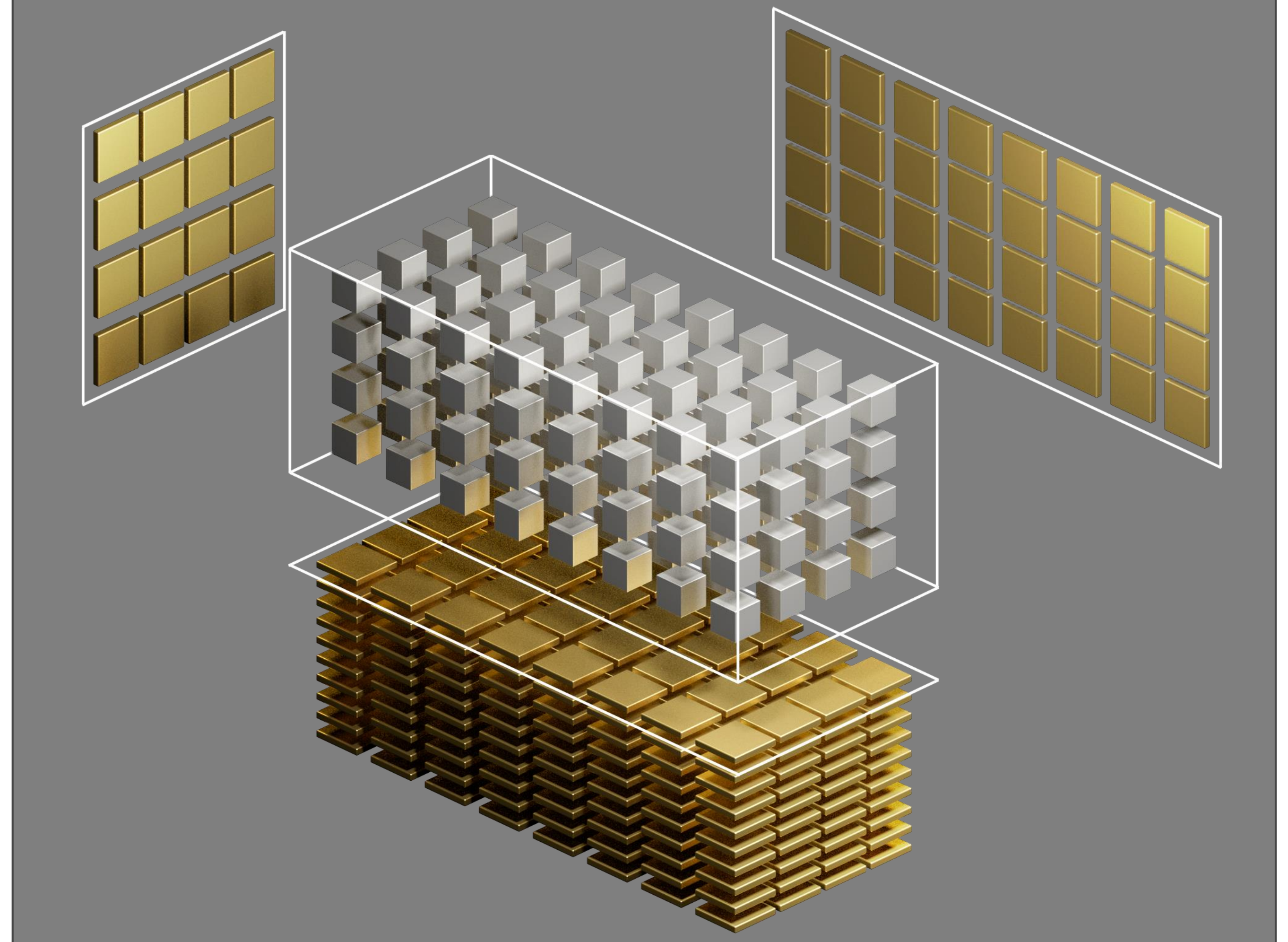
Math API

NVIDIA PERFORMANCE LIBRARIES

Core and Math Library Directions

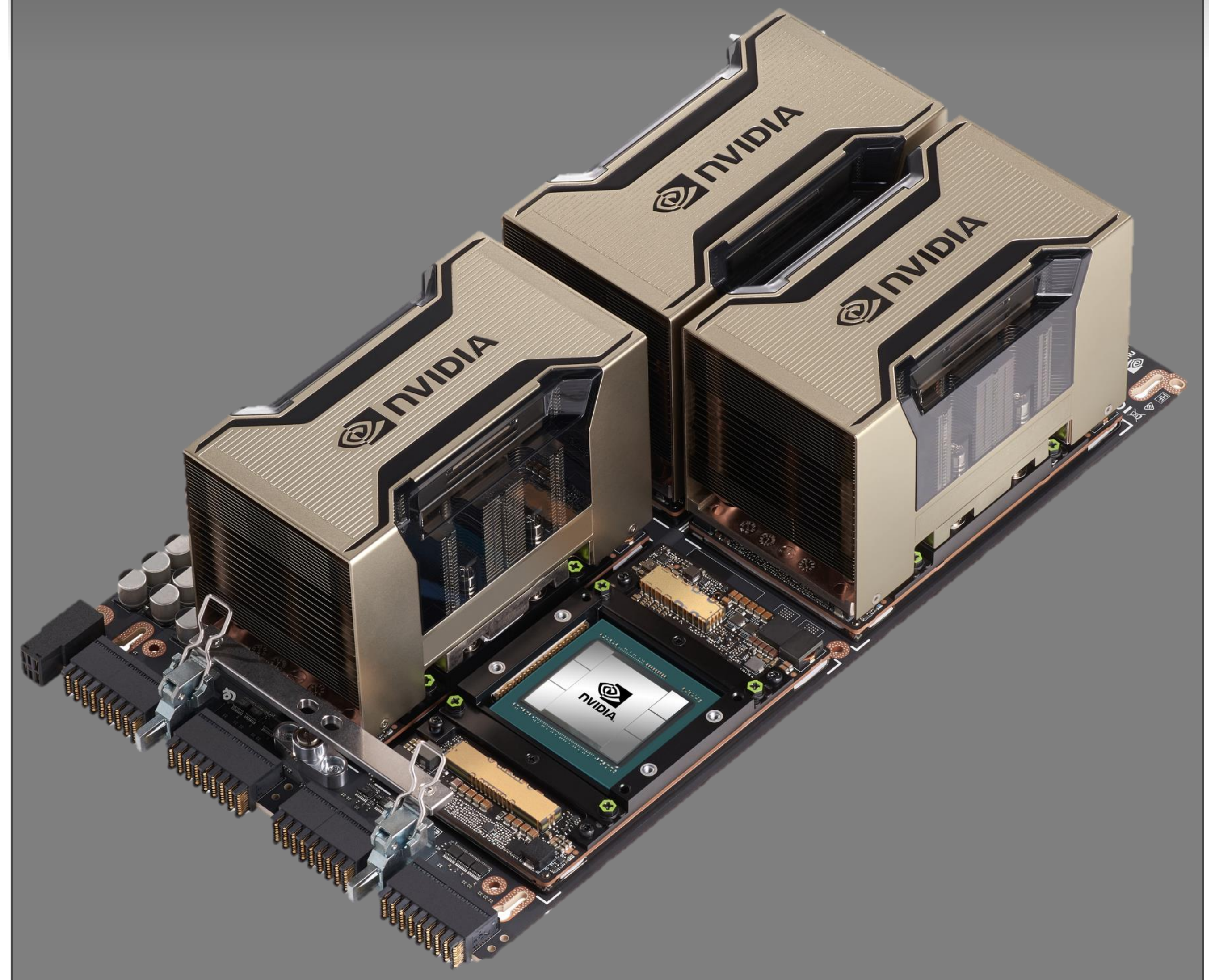
Seamless Acceleration

Tensor Cores, GH C2C



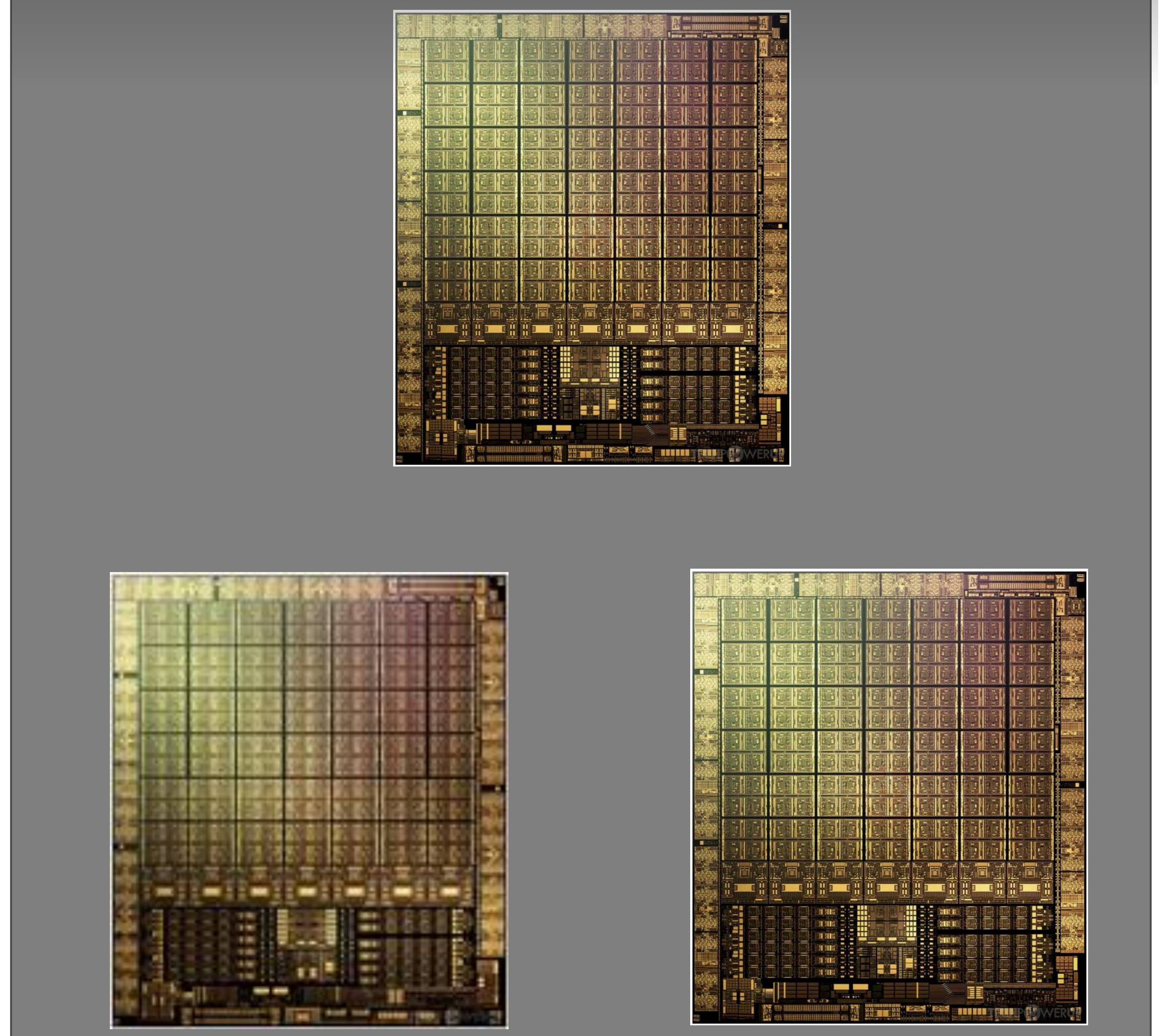
Scaling Up

Multi-GPU and Multi-Node Libraries



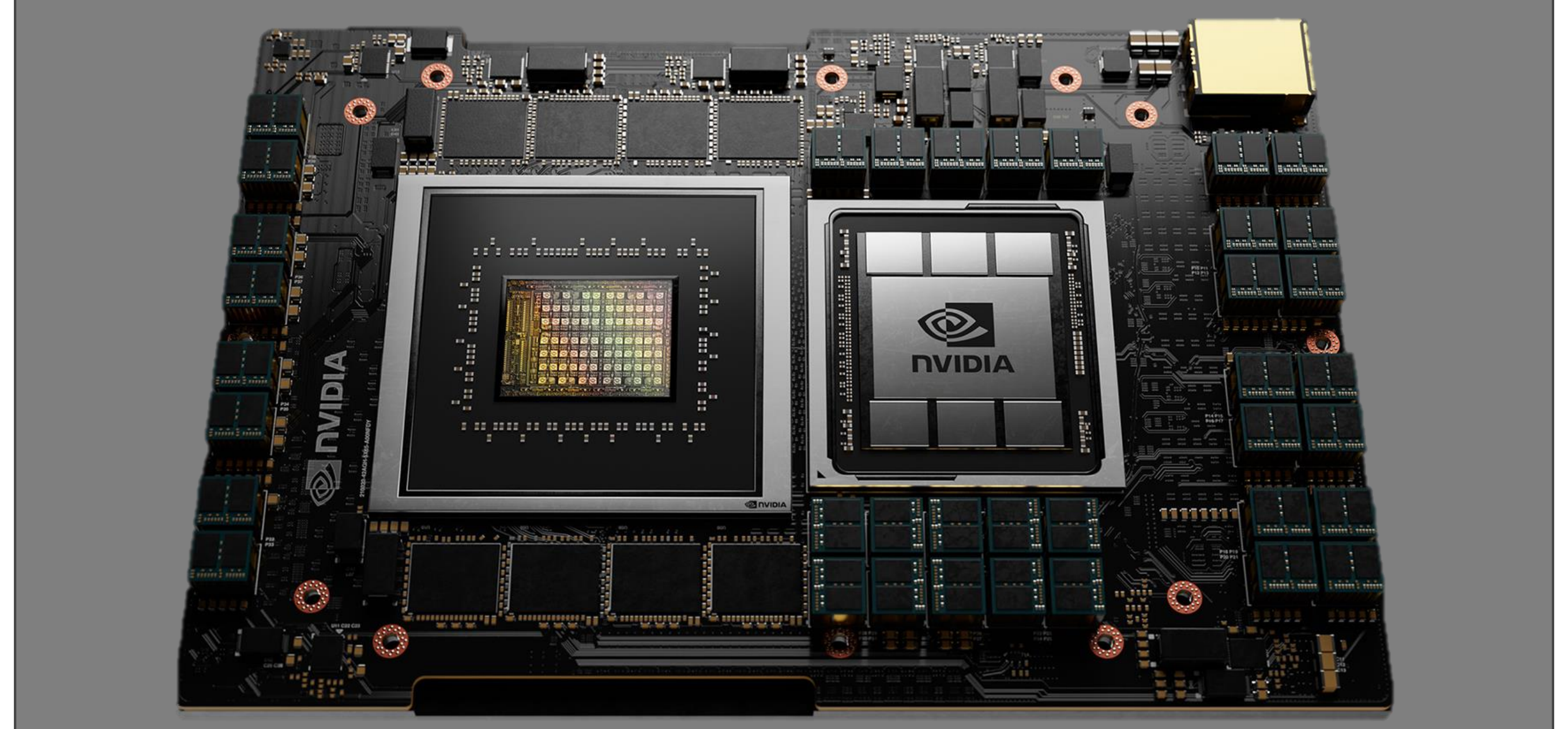
Composability

Device Functions



Arm Execution

High Performance CPU Libraries



Standard Language Approaches

HPC PROGRAMMING IN ISO C++

ISO is the place for portable concurrency and parallelism

C++17 & C++20

Parallel Algorithms

- In NVC++
- Parallel and vector concurrency

Forward Progress Guarantees

- Extend the C++ execution model for accelerators

Memory Model Clarifications

- Extend the C++ memory model for accelerators

Ranges

- Simplifies iterating over a range of values

Scalable Synchronization Library

- Express thread synchronization that is portable and scalable across CPUs and accelerators
- In libcu++:
  - `std::atomic<T>`
  - `std::barrier`
  - `std::counting_semaphore`
  - `std::atomic<T>::wait/notify_*`
  - `std::atomic_ref<T>`

Preview support coming to NVC++

C++23

`std::mdspan/mdarray`

- HPC-oriented multi-dimensional array abstractions.
- Preview Implementation In Progress!

Range-Based Parallel Algorithms

- Improved multi-dimensional loops

Extended Floating Point Types

- First-class support for formats new and old: `std::float16_t/float64_t`

And Beyond

Executors / Senders-Receivers

- Simplify launching and managing parallel work across CPUs and accelerators
- Preview Implementation In Progress!

Linear Algebra

- C++ standard algorithms API to linear algebra
- Maps to vendor optimized BLAS libraries
- Preview Implementation In Progress!

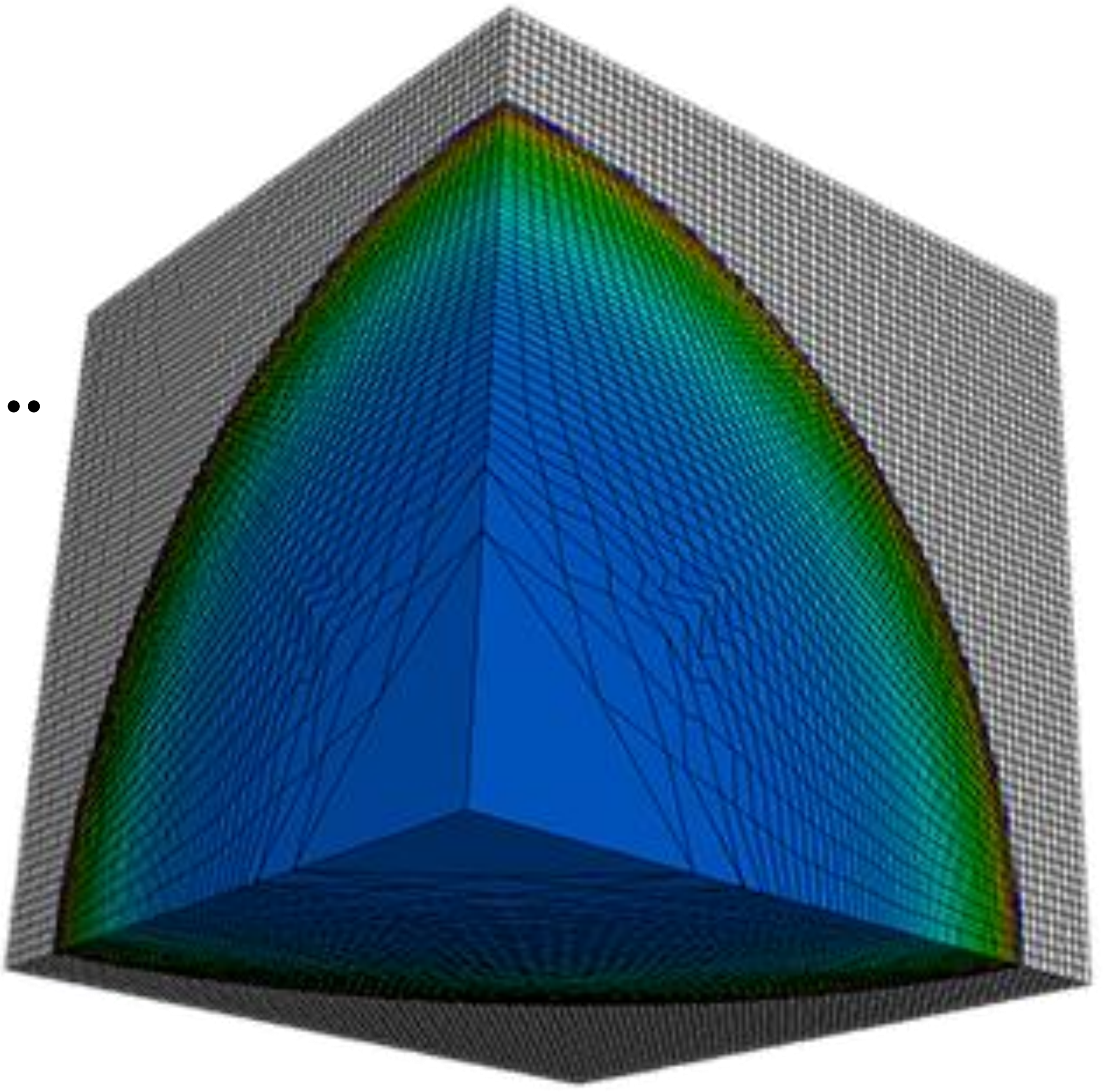
Lulesh with Standard C++

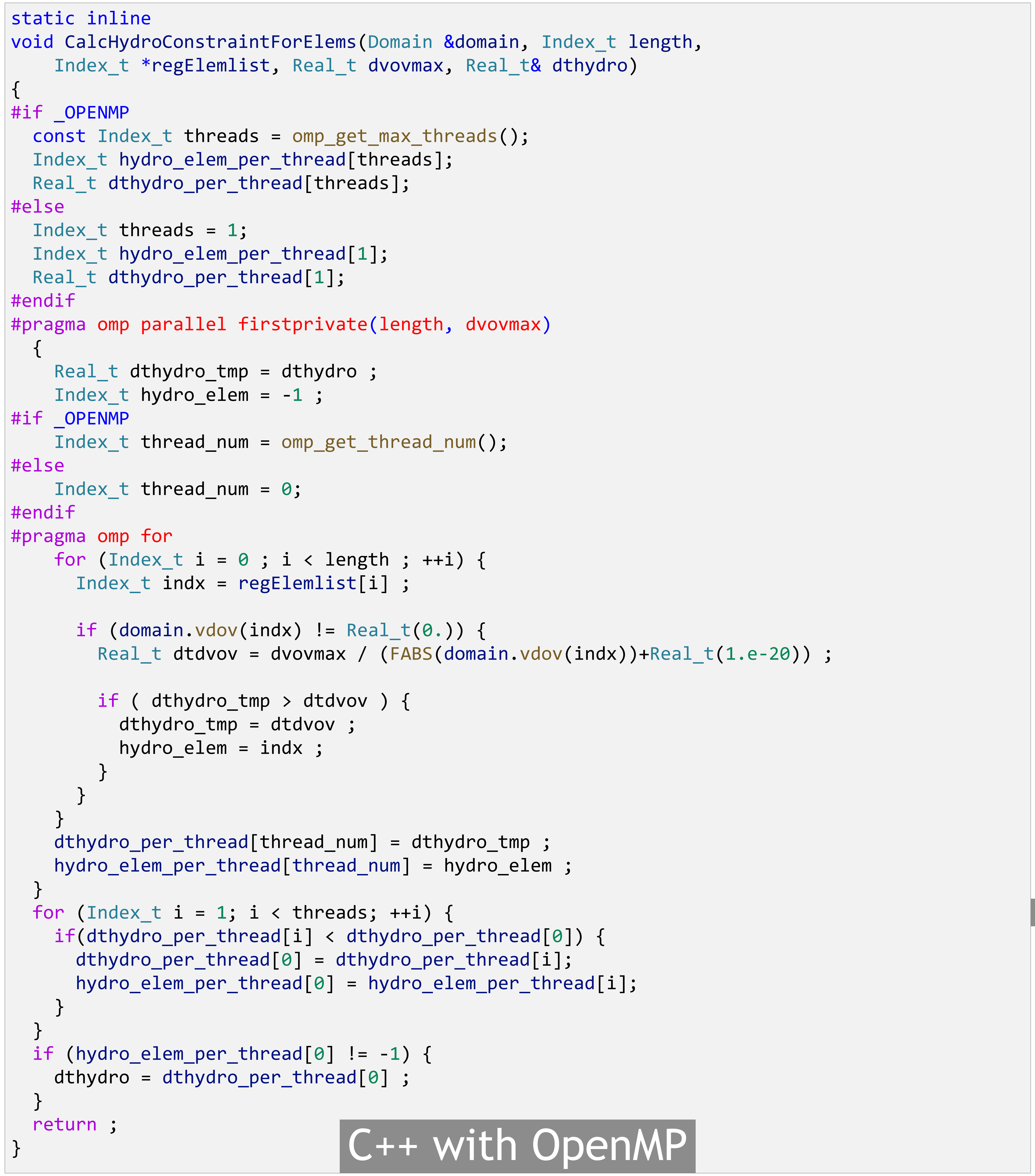
About Lulesh

- Hydrodynamics Mini-App from LLNL
- ~9000 LOC, C++, OpenMP, CUDA, RAJA, …

With Standard C++:

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



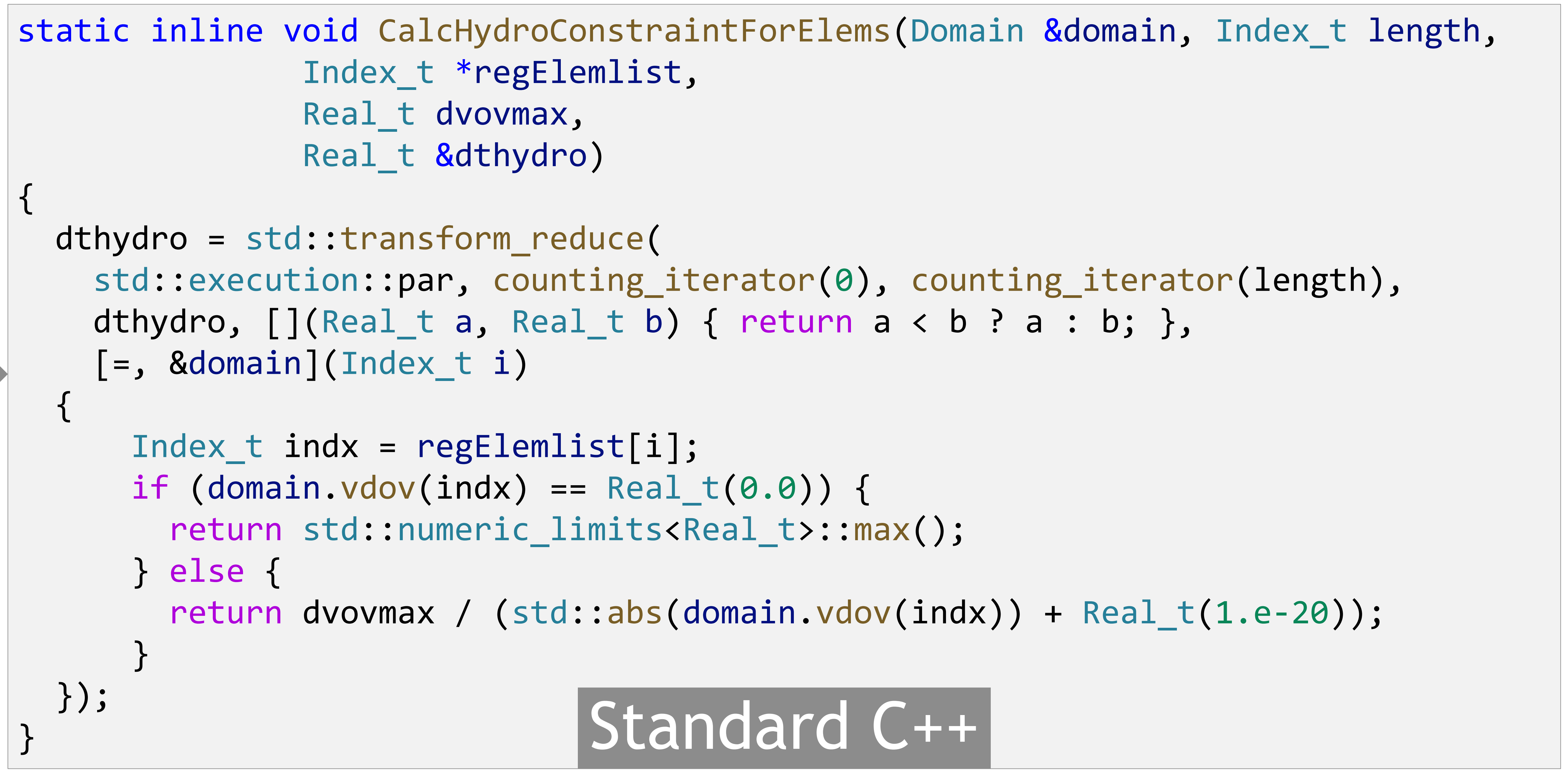


```
static inline
void CalcHydroConstraintForElems(Domain &domain, Index_t length,
    Index_t *regElemlist, Real_t dvovmax, Real_t& dthydro)
{
#if _OPENMP
  const Index_t threads = omp_get_max_threads();
  Index_t hydro_elem_per_thread[threads];
  Real_t dthydro_per_thread[threads];
#else
  Index_t 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



```
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));
      }
  });
}
```

Standard C++

codesign.llnl.gov/lulesh

C++ Standard Parallelism

Lulesh Performance

Lulesh Speed-up
NVC++
GCC
Intel
16.00X
14.00X
12.00X
10.00X
8.00X
6.00X
4.00X
2.00X
0.00X
1.19X
1.00X
1.00X
1.98X
1.91X
2.71X
14.75X
g++
icpc
nvc++
g++
icpc
nvc++ (CPU)
nvc++ (GPU)
OpenMP (default)
ISO C++

Same ISO C++ Code

AMD EPYC 7742 CPU, NVIDIA A100 GPU. g++ version 10.3.0, icpc version 2021.5.0, nvc++ version 22.3

HPC PROGRAMMING IN ISO FORTRAN

ISO is the place for portable concurrency and parallelism

Preview support available now in NVFORTRAN

Fortran 2018

Fortran Array Intrinsics

- NVFORTRAN 20.5
- Accelerated matmul, reshape, spread, …

DO CONCURRENT

- NVFORTRAN 20.11
- Auto-offload & multi-core

Co-Arrays

- Not currently available
- Accelerated co-array images

Fortran 202x

DO CONCURRENT Reductions

- NVFORTRAN 21.11
- `REDUCE` subclause added
- Support for `+`, `*`, `MIN`, `MAX`, `IAND`, `IOR`, `IEOR`.
- Support for `.AND.`, `.OR.`, `.EQV.`, `.NEQV` on `LOGICAL` values

MiniWeather

Standard Language Parallelism in Climate/Weather Applications

MiniWeather

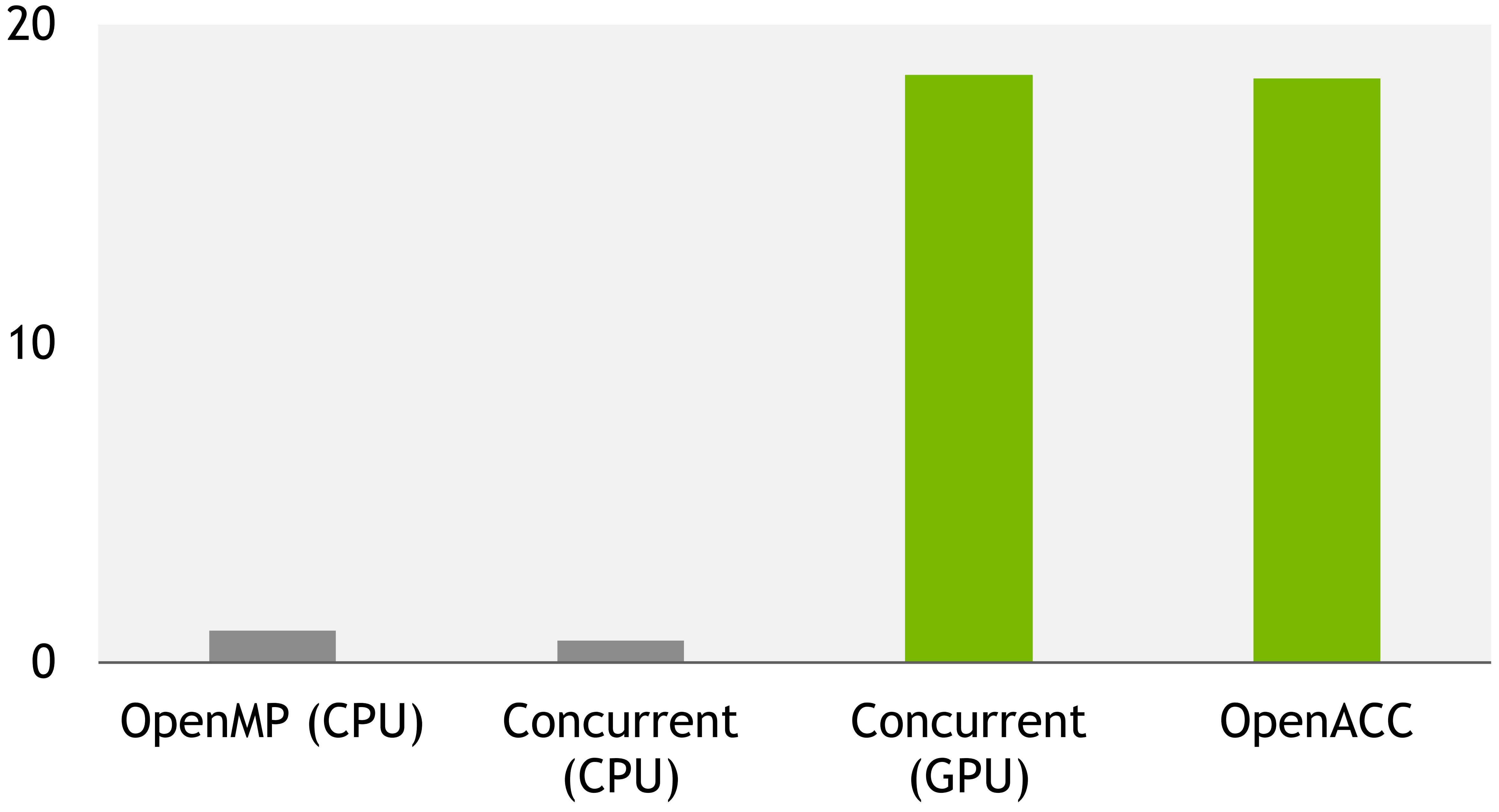
Mini-App written in C++ and Fortran that simulates weather-like fluid flows using Finite Volume and Runge-Kutta methods.

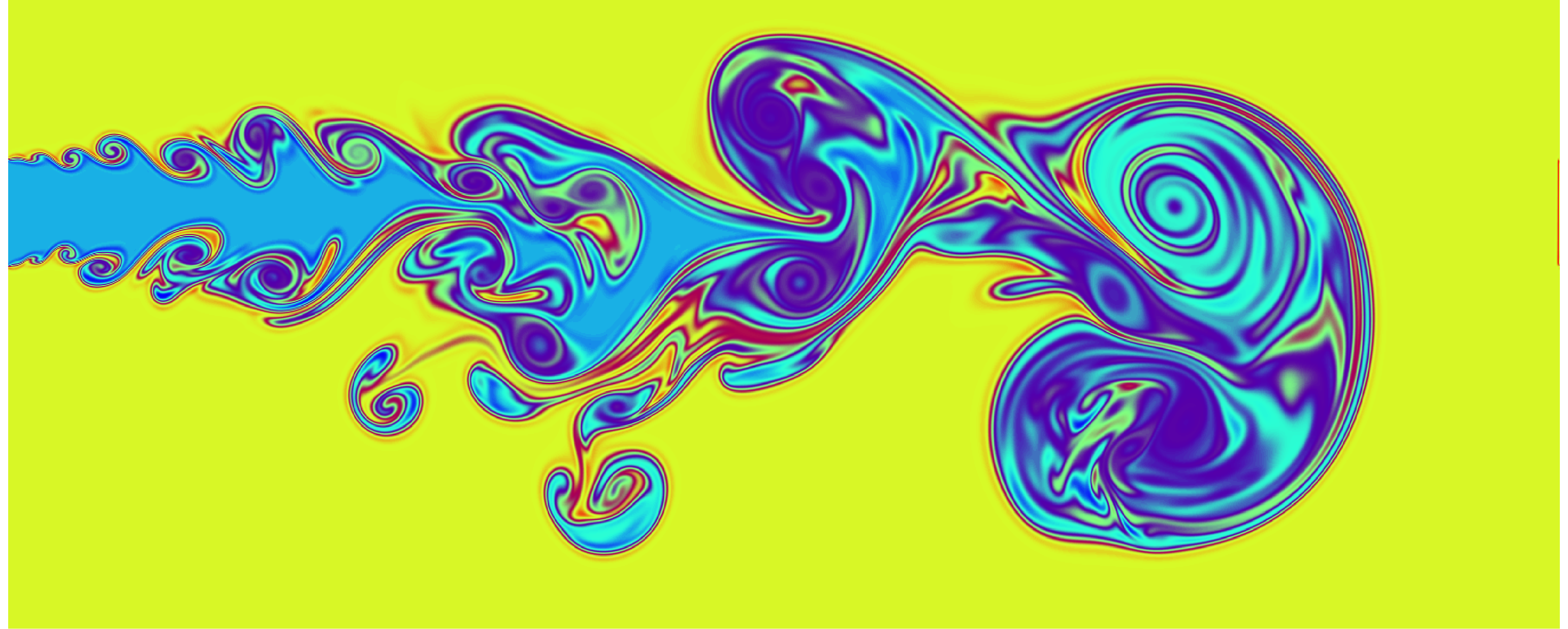
Existing parallelization in MPI, OpenMP, OpenACC, …

Included in the SPEChpc benchmark suite*

Open-source and commonly-used in training events.

https://github.com/mrnorman/miniWeather/





```
do concurrent (ll=1:NUM_VARS, k=1:nz, i=1:nx)
   local(x,z,x0,z0,xrad,zrad,amp,dist,wpert)

 if (data_spec_int == DATA_SPEC_GRAVITY_WAVES) then
  x = (i_beg-1 + i-0.5_rp) * dx
  z = (k_beg-1 + k-0.5_rp) * dz
    x0 = xlen/8
  z0 = 1000
  xrad = 500
  zrad = 500
  amp = 0.01_rp
  dist = sqrt( ((x-x0)/xrad)**2 + ((z-z0)/zrad)**2 )
     * pi / 2._rp
  if (dist <= pi / 2._rp) then
   wpert = amp * cos(dist)**2
  else
   wpert = 0._rp
  endif
  tend(i,k,ID_WMOM) = tend(i,k,ID_WMOM)
           + wpert*hy_dens_cell(k)
 endif
 state_out(i,k,ll) = state_init(i,k,ll)
          + dt * tend(i,k,ll)

enddo
```

*SPEChpc is a trademark of The Standard Performance Evaluation Corporation

Source: HPC SDK 22.1, AMD EPYC 7742, NVIDIA A100. MiniWeather: NX=2000, NZ=1000, SIM_TIME=5. OpenACC version uses -gpu=managed option.

Compiler Directive Approaches

What is OpenACC?

OpenACC is a directive-based parallel programming model designed for productivity, performance, and portability

APPLICATIONS	PLATFORMS SUPPORTED	COMMUNITY
250+ 3 out of Top 5	NVIDIA GPU X86 CPU POWER CPU Sunway ARM CPU AMD GPU FPGA	**~3000** Slack Members

Parallelize with OpenACC

```
  while ( error > tol && iter < iter_max )
  {
   double error = 0.0;
#pragma acc parallel loop reduction(max:error)
   for (int j = 1; j < n - 1; j++)
   {
    for (int i = 1; i < m - 1; i++)
    {
     Anew[OFFSET(j, i, m)] = 0.25 * \
          (A[OFFSET(j, i + 1, m)] + A[OFFSET(j, i - 1, m)] + \
           A[OFFSET(j - 1, i, m)] + A[OFFSET(j + 1, i, m)]);
     error = fmax(error, fabs(Anew[OFFSET(j, i, m)] - A[OFFSET(j, i, m)]));
    }
   }

#pragma acc parallel loop
   for (int j = 1; j < n - 1; j++)
   {
    for (int i = 1; i < m - 1; i++)
    {
     A[OFFSET(j, i, m)] = Anew[OFFSET(j, i, m)];
    }
   }

   if (iter % 100 == 0)
    printf("%5d, %0.6f\n", iter, error);

   iter++;
  }
```

Parallelize first loop nest, max *reduction* required.

Parallelize second loop.

We didn’t detail *how* to parallelize the loops, just *which* loops to parallelize.

Parallelize with OpenMP Offloading

```
    while ( error > tol && iter < iter_max )
    {
      double error = 0.0;
#pragma omp target teams loop reduction(max:error) collapse(2)
      for (int j = 1; j < n - 1; j++)
      {
        for (int i = 1; i < m - 1; i++)
        {
          Anew[OFFSET(j, i, m)] = 0.25 * \
                    (A[OFFSET(j, i + 1, m)] + A[OFFSET(j, i - 1, m)] + \
                     A[OFFSET(j - 1, i, m)] + A[OFFSET(j + 1, i, m)]);
          error = fmax(error, fabs(Anew[OFFSET(j, i, m)] - A[OFFSET(j, i, m)]));
        }
      }

#pragma omp target teams loop collapse(2)
      for (int j = 1; j < n - 1; j++)
      {
        for (int i = 1; i < m - 1; i++)
        {
          A[OFFSET(j, i, m)] = Anew[OFFSET(j, i, m)];
        }
      }

      if (iter % 100 == 0)
        printf("%5d, %0.6f\n", iter, error);

      iter++;
    }
```

OpenMP Target Offloading looks similar to OpenACC, but requires more understanding from the developer due to having a myriad combination of possible directives to use.

Python Appoaches

Overview of cupy

- **CuPy** supports a subset of numpy.ndarray interface which includes:

- ✓ Basic & advance indexing, and Broadcasting
- ✓ Data types (int32, float32, uint64, complex64,…)
- ✓ Array manipulation routine (reshape)
- ✓ Linear Algebra functions (dot, matmul, etc)
- ✓ Reduction along axis (max, sum, argmax, etc)

For more details on broadcasting visit

(https://numpy.org/doc/stable/user/basics.broadcasting.html)

```
>>> import numpy as np
>>> X = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
#Basic indexing and slicing
>>> X[5:]
array([5, 6, 7, 8, 9])
>>> X[1:7:2]
array([1, 3, 5])

#Advance indexing
>>> X = np.array([[1, 2],[3, 4],[5, 6]])
>>> X[[0, 1, 2], [0, 1, 0]]
array([1, 4, 5])

#reduction and Linear Algebra function
>>> max(X)
9.0
>>> B = np.array([1,2,3,4], dtype=np.float32)
>>> C = np.array([5,6,7,8], dtype=np.float32)
>>> np.matmul(B, C)
70.0
#data type and array manipulation routine
>>> A =1j*np.arange(9, dtype=np.complex64).reshape(3,3)
[[0.+0.j 0.+1.j 0.+2.j]
 [0.+3.j 0.+4.j 0.+5.j]
 [0.+6.j 0.+7.j 0.+8.j]]
```

Overview of cupy

- **CuPy** supports a subset of numpy.ndarray interface which includes:

✓ Basic & advance indexing, and Broadcasting

✓ Data types (int32, float32, uint64, complex64,…)

✓ Array manipulation routine (reshape)

✓ Linear Algebra functions (dot, matmul, etc)

✓ Reduction along axis (max, sum, argmax, etc)

For more details on broadcasting visit

(https://numpy.org/doc/stable/user/basics.broadcasting.html)

```
>>> import cupy as cp
>>> X = cp.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
#Basic indexing and slicing
>>> X[5:]
array([5, 6, 7, 8, 9])
>>> X[1:7:2]
array([1, 3, 5])

#Advance indexing
>>> X = cp.array([[1, 2],[3, 4],[5, 6]])
>>> X[[0, 1, 2], [0, 1, 0]]
array([1, 4, 5])

#reduction and Linear Algebra function
>>> max(X)
9.0
>>> B = cp.array([1,2,3,4], dtype=np.float32)
>>> C = cp.array([5,6,7,8], dtype=np.float32)
>>> cp.matmul(B, C)
70.0
#data type and array manipulation routine
>>> A =1j*cp.arange(9, dtype=np.complex64).reshape(3,3)
[[0.+0.j 0.+1.j 0.+2.j]
 [0.+3.j 0.+4.j 0.+5.j]
 [0.+6.j 0.+7.j 0.+8.j]]
```

cunumeric

Automatic NumPy Acceleration and Scalability

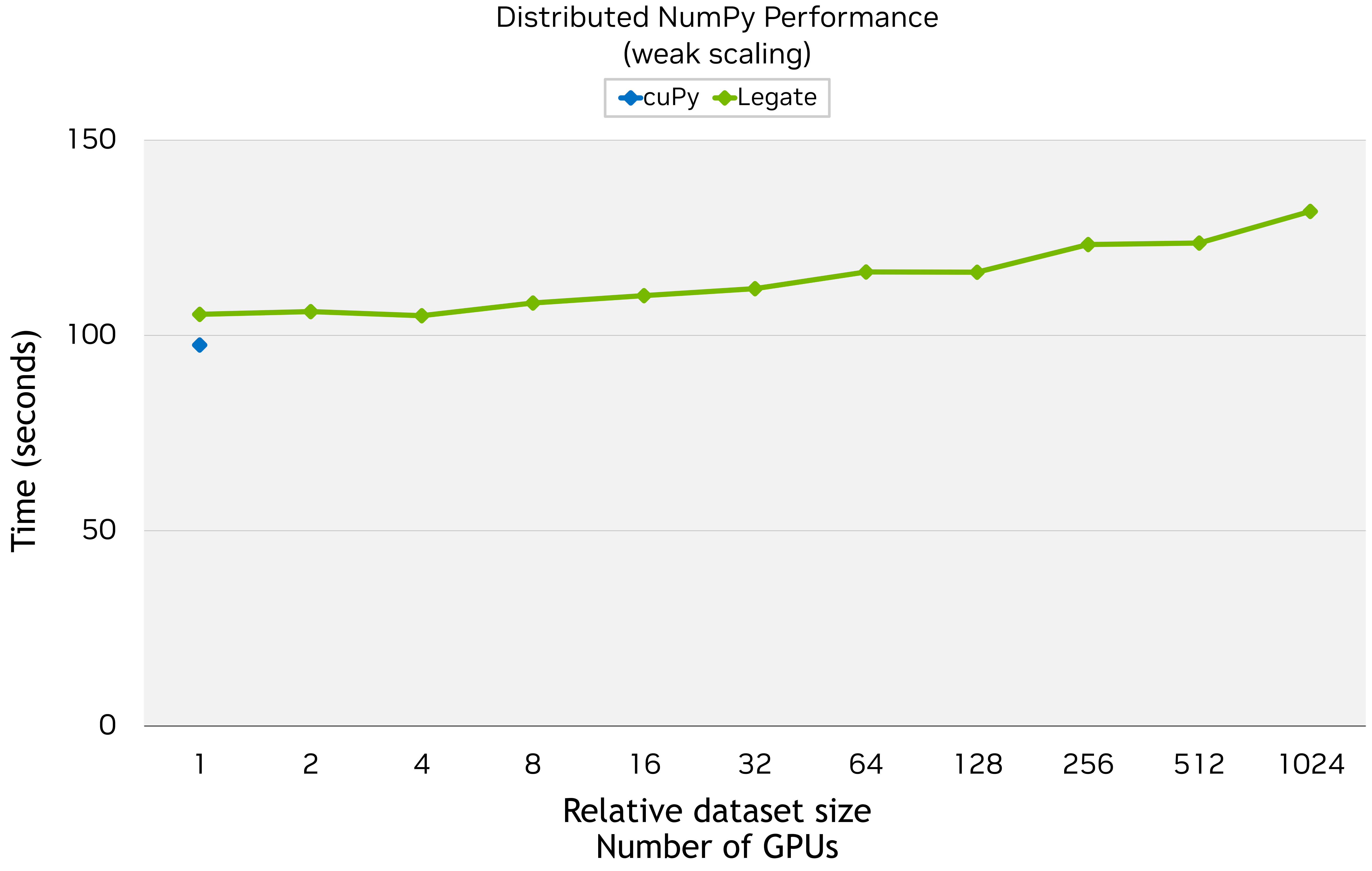
cuNumeric

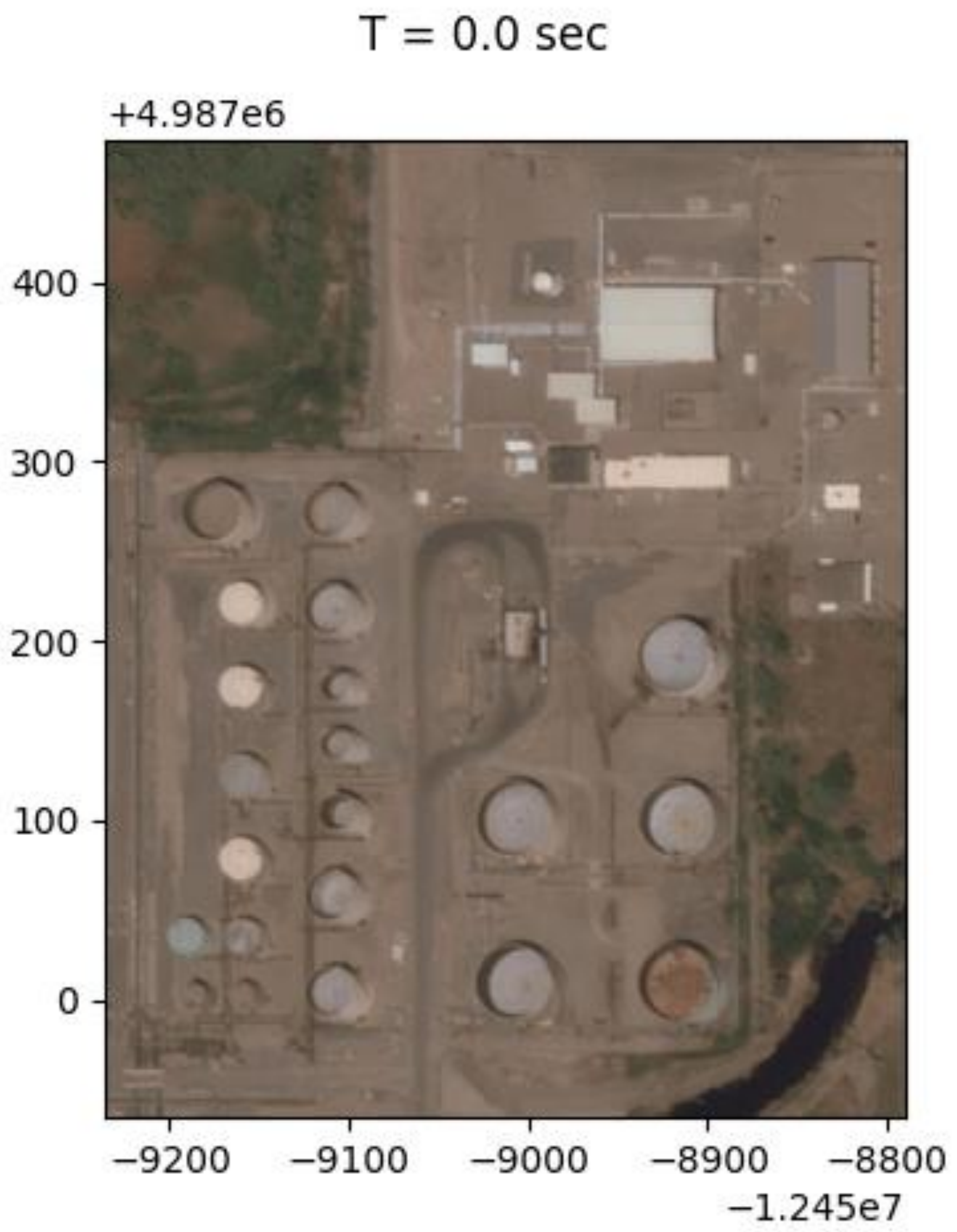
CuNumeric transparently accelerates and scales existing Numpy workloads

Program from the edge to the supercomputer in Python by changing as little as 1 import line

Pass data between Legate libraries without worrying about distribution or synchronization requirements

Alpha release available at github.com/nv-legate





```
for _ in range(iter):
    un = u.copy()

    vn = v.copy()
    b = build_up_b(rho, dt, dx, dy, u, v)
    p = pressure_poisson_periodic(b, nit, p, dx, dy)
```

…

Extracted from “CFD Python” course at https://github.com/barbagroup/CFDPython
Barba, Lorena A., and Forsyth, Gilbert F. (2018). CFD Python: the 12 steps to Navier-Stokes equations. *Journal of Open Source Education*, **1**(9), 21, https://doi.org/10.21105/jose.00021

Numba Example

```
import numba.cuda as cuda
import numpy as np

N = 500000
threadsperblock = 1204

@cuda.jit
def arrayAdd(array_A, array_B, array_out):
   tid = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
   if tid < N:
      array_out[tid] = array_A[tid] + array_B[tid]


array_A   = np arange(N, dtype = np.int)
array_B   = np arange(N, dtype = np.int)
array_out = np zeros(N,  dtype = np.int)

blockpergrid  = N + (threadsperblock - 1) // threadsperblock

arrayAdd[blockpergrid, threadsperblock](array_A, array_B, array_out)
```

Numba relies on method decorators and just-in-time compilation to build methods for GPU execution

CUDA C++ and Fortran

Writing a CUDA Kernel

```
// __global__ is a CUDA keyword denoting
// this function is a GPU kernel
__global__ void add( int *a, int *b, int *c )
{




  // Calculate this thread’s offset into
  // the calculation
  int index = threadIdx.x +
              blockIdx.x * blockDim.x;

  // Perform the calculation
  c[index] = a[index] + b[index];
}
```

```
! attributes(global) is a CUDA keyword
! denoting this subroutine is a GPU kernel
attributes(global) subroutine add(n, a, b, c)

  integer, value :: n
  real(8), device :: a(n), b(n), c(n)
  integer :: idx

  ! Calculate this thread’s offset into
  ! the calculation
  idx = threadidx%x + &
       (blockidx%x-1)*blockdim%x

  ! Perform the calculation
  c(idx) = a(idx) + b(idx)

end subroutine add
```

This routine is called by every thread launch on the device

Calling a CUDA Kernel

*Note: In these examples I know the threads per block divide evenly into N

```
int main( void ) {
  int *a, *b, *c;
  const int tpb = 512,
            N=(2048*2048);




  // Allocate "managed" arrays
  cudaMallocManaged( &a, N*sizeof(int) );
  cudaMallocManaged( &b, N*sizeof(int) );
  cudaMallocManaged( &c, N*sizeof(int) );





  // Launch Kernel
  add<<< (N/tpb, tpb >>>( a, b, c );

  // Ensure GPU work completes
  cudaDeviceSynchronize();


  // Free arrays
  cudaFree( a );
  cudaFree( b );
  cudaFree( c );
  return 0;
}
```

```
program main
  use cudafor

  real, managed, allocatable, dimension(:) :: &
      a, b, c
  integer, parameter :: N = (2048*2048)
  type(dim3) :: blockSize, gridSize

  ! Allocate "managed" arrays
  allocate(a(N))
  allocate(b(N))
  allocate(c(N))

  blockSize = dim3(512,1,1)
  gridSize = dim3(n/blockSize%x ,1,1)

  ! Launch Kernel
  call add<<<gridSize, blockSize>>>(n, a, b, c)

  ! Ensure GPU work completes
  cudaDeviceSynchronize()


  ! Free arrays
  deallocate(a)
  deallocate(b)
  deallocate(c)

end program main
```

Calling a CUDA Kernel & Explicitly Managing Data

```
int main( void ) {
  int *a, *b, *c, *d_a, *d_b, *d_c;
  const int tpb = 512,
            N=(2048*2048);

  // Allocate "host" arrays
  a = (int*)malloc(N*sizeof(int));
  b = (int*)malloc(N*sizeof(int));
  c = (int*)malloc(N*sizeof(int));

  // Allocate "device" arrays
  cudaMalloc( &d_a, N*sizeof(int) );
  cudaMalloc( &d_b, N*sizeof(int) );
  cudaMalloc( &d_c, N*sizeof(int) );

  // Copy data from Host to Device
  cudaMemcpy(d_a, a, N*sizeof(int), cudaMemcpyHostToDevice);
  cudaMemcpy(d_b, b, N*sizeof(int), cudaMemcpyHostToDevice);




  // Launch Kernel
  add<<< (N/tpb, tpb >>>( a, b, c );

  cudaDeviceSynchronize(); // Ensure GPU work completes

  // Copy data from Device to Host
  cudaMemcpy(c, d_c, N*sizeof(int), cudaMemcpyDeviceToHost);

  // Free arrays
  cudaFree( d_a ); free(a);
  cudaFree( d_b ); free(b);
  cudaFree( d_c ); free(c);
  return 0;
}
```

```
program main
  use cudafor

  real, allocatable, dimension(:) :: a, b, c
  real, device, allocatable, dimension(:) :: &
      d_a, d_b, d_c
  integer, parameter :: N = (2048*2048)
  type(dim3) :: blockSize, gridSize

  ! Allocate "host" arrays
  allocate(a(N),b(N),c(N))

  ! Allocate "device" arrays
  allocate(d_a(N),d_b(N),d_c(N))

  ! Copy data from Host to Device
  d_a = a
  d_b = b

  blockSize = dim3(512,1,1)
  gridSize = dim3(n/blockSize%x ,1,1)

  ! Launch Kernel
  call add<<<gridSize, blockSize>>>(n, a, b, c)

  cudaDeviceSynchronize() ! Ensure GPU work completes

  ! Copy data from Device to Host
  c = d_c

  ! Free arrays
  deallocate(a,b,c)
  deallocate(d_a,d_b,d_c)

end program main
```

Closing remarks

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

x86_64 | Arm | OpenPOWER

7-8 Releases Per Year | Freely Available

Conclusions

- NVIDIA provides a wide range of mature GPU programming models
- Developers can mix this programming models to obtain the right level of productivity, portability, and performance for their needs
- I encourage you to dig in more on the programming models I presented that feel best to you

GTC 2022 Sessions to Watch

For more information on these topics

GTC22 Fall

- A Deep Dive into the Latest HPC Software [A41133]
- CUDA: New Features and Beyond [A41100]
- How CUDA Programming Works [A41101]
- Developing HPC Applications with Standard C++, Fortran, and Python [A41087]

GTC22 Spring

- C++ Standard Parallelism [S41960]
- Future of Standard and CUDA C++ [S41961]
- Shifting through the Gears of GPU Programming: Understanding Performance and Portability Trade-offs [S41620]
- From Directives to DO CONCURRENT: A Case Study in Standard Parallelism [S41318]
- Evaluating Your Options for Accelerated Numerical Computing in Pure Python [S41645]
- How to Develop Performance Portable Codes using the Latest Parallel Programming Standards [S41618]

Additional Resources

- CUDA C++ Programming Guide
- CUDA Fortran Programming Guide
- NVIDIA HPC SDK
- OpenACC Getting Started Guide
- C++ Parallel Algorithms
- CuPy
- cuNumeric
- Numba