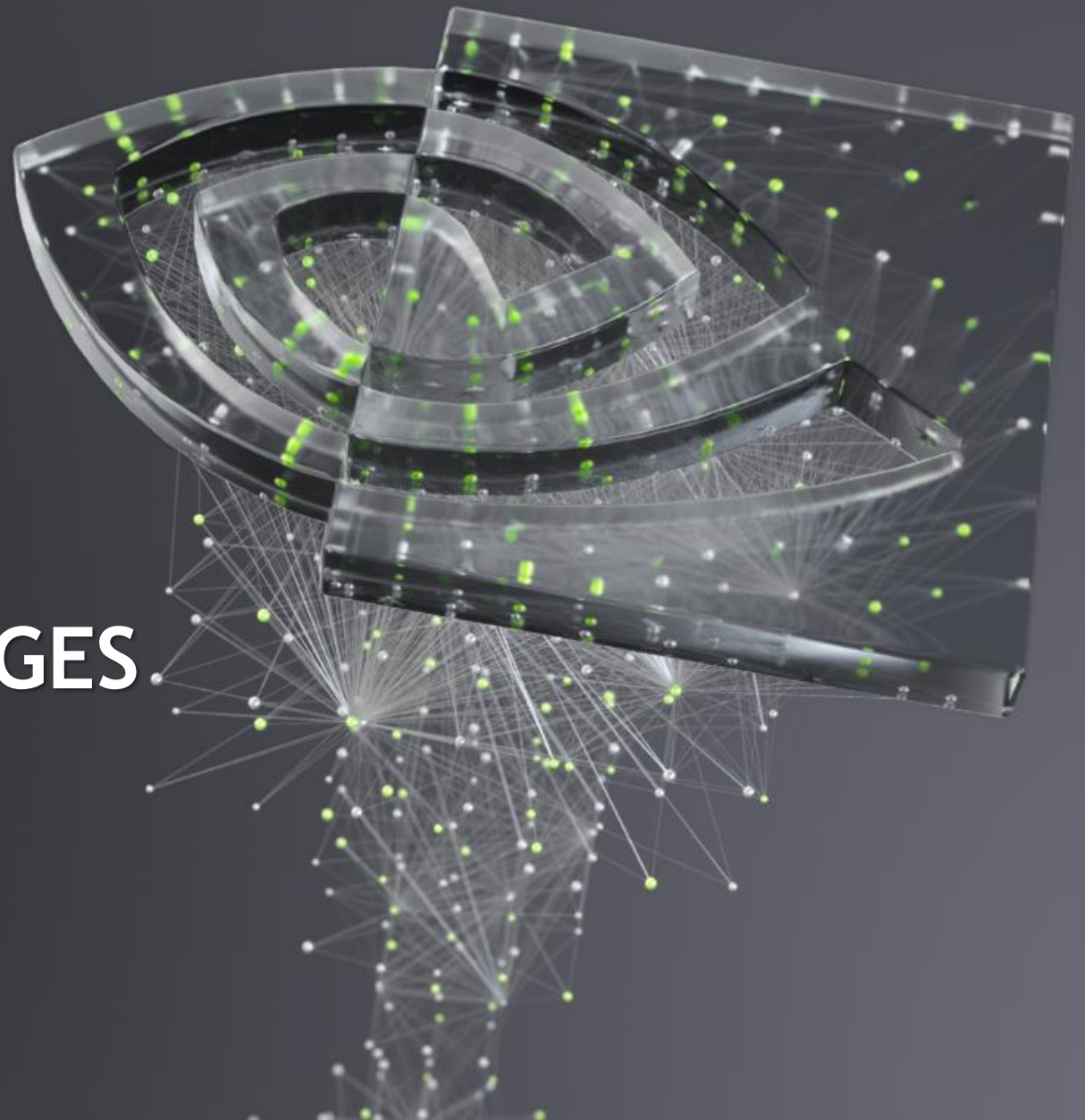


STANDARD LANGUAGES FOR PARALLEL PROGRAMMING

Jeff Larkin| Principal HPC Architect | NVIDIA

PC Cluster Consortium, April 2022

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

ACCELERATED STANDARD LANGUAGES

Parallel performance for wherever your code runs

ISO C++

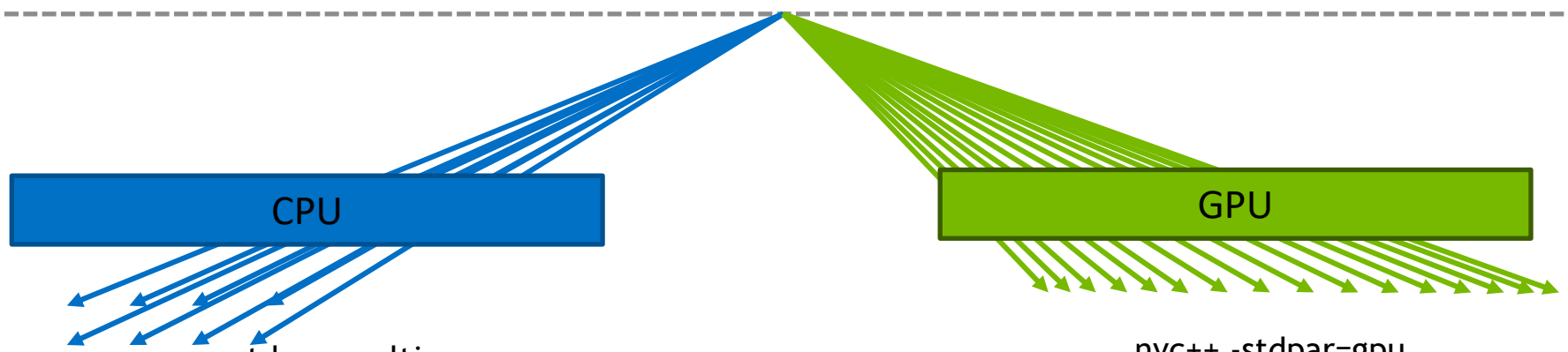
```
std::transform(par, x, x+n, y,
   y,[=](float x, float y){
      return y + a*x;
    }
);
```

ISO Fortran

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

Python

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



nvc++ -stdpar=multicore
nvfortran -stdpar=multicore
legate -cpus 16 saxpy.py

nvc++ -stdpar=gpu
nvfortran -stdpar=gpu
legate -gpus 1 saxpy.py

FUTURE OF CONCURRENCY AND PARALLELISM IN HPC: STANDARD LANGUAGES

How did we get here?

ON-GOING LONG-TERM INVESTMENT

ISO committee participation from industry, academia and government labs.

Fruit born in 2020 was planted over the previous decade.

Focus on enhancing concurrency and parallelism for all.

Open collaboration between partners and competitors.

Past investments in directives enabled rapid progress.

MAJOR FEATURES

Memory Model Enhancements

C++14 Atomics Extensions

C++17 Parallel Algorithms

C++20 Concurrency Library

C++23 Multi-Dim. Array Abstractions

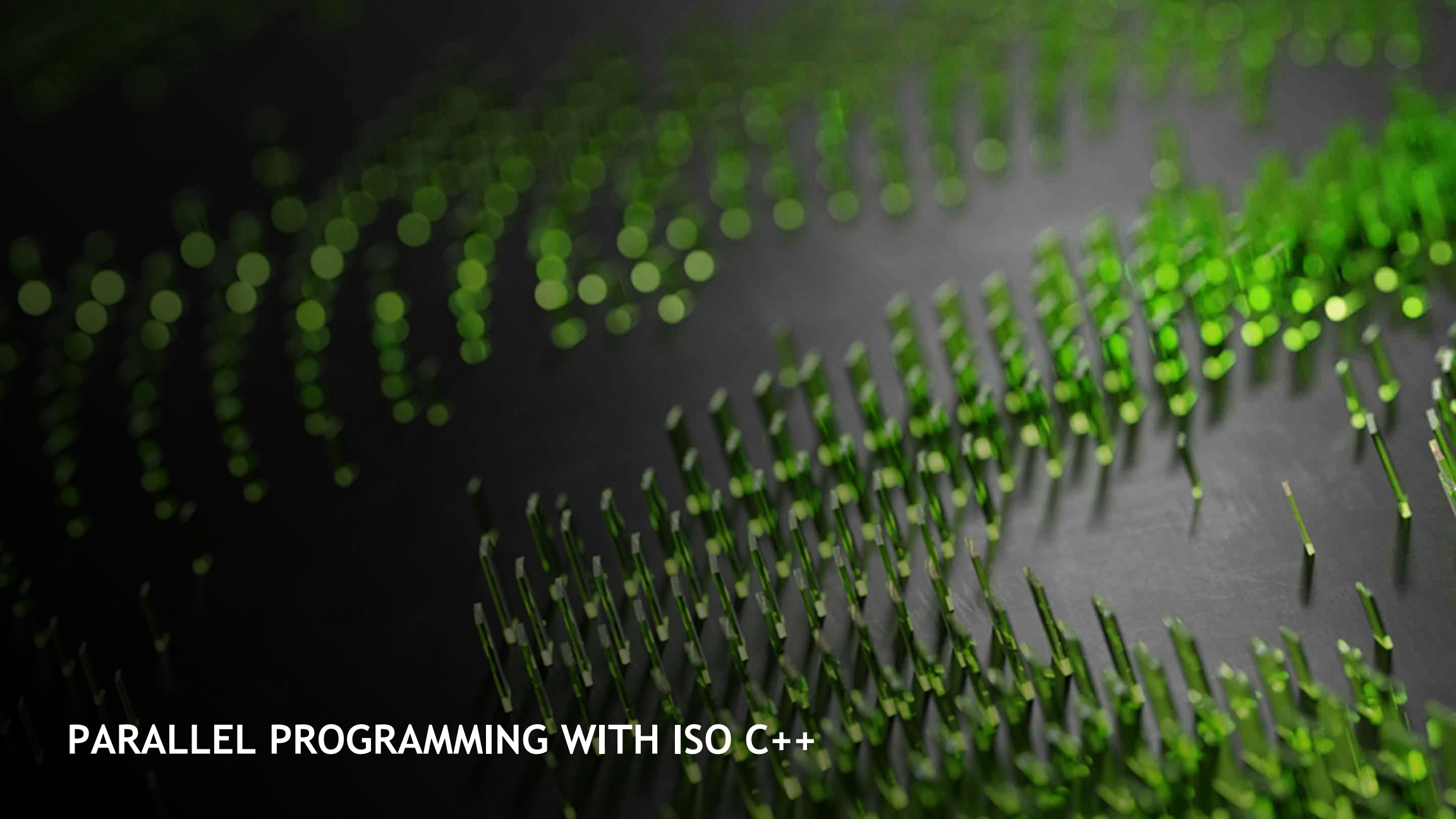
C++23 Extended Floating Point Types

C++23 Range Based Parallel Algorithms

C++2X Executors

C++2X Linear Algebra

Fortran 202X DO CONCURRENT Reduction


PARALLEL PROGRAMMING WITH ISO C++

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.

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-Recievers

- Simplify launching and managing parallel work across CPUs and accelerators

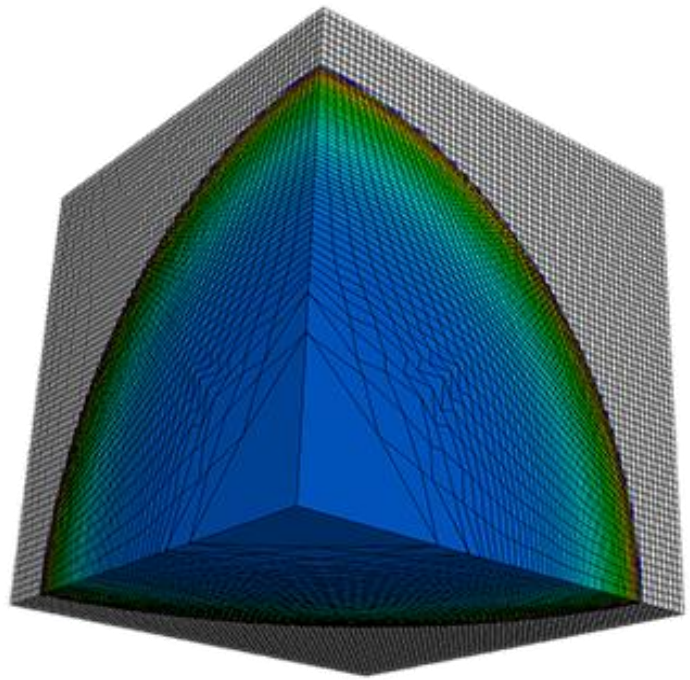
Linear Algebra

- C++ standard algorithms API to linear algebra
- Maps to vendor optimized BLAS libraries

C++17 PARALLEL ALGORITHMS

Lulesh Hydrodynamics Mini-app

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



codesign.llnl.gov/lulesh

```cpp
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

STANDARD C++

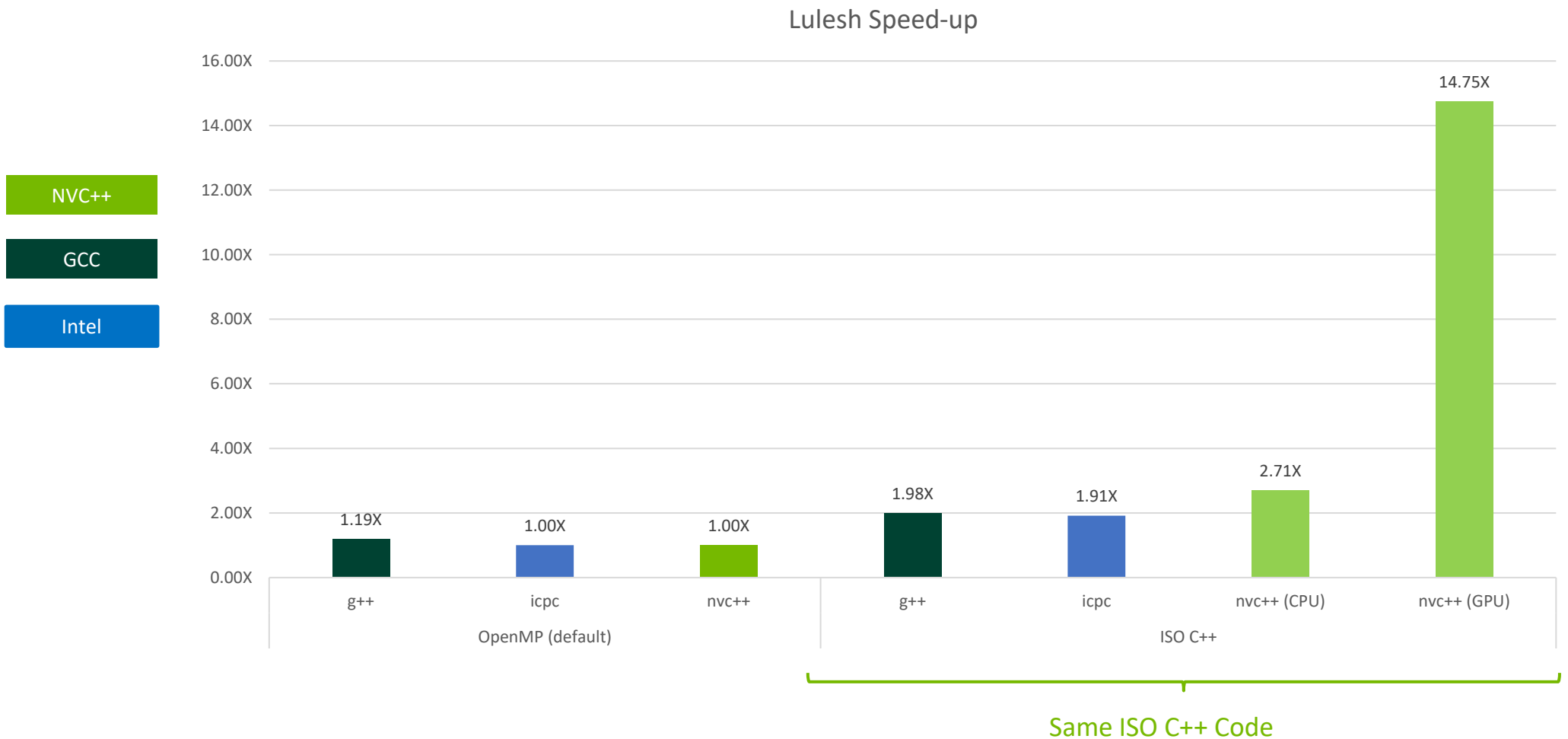
- Composable, compact and elegant
- Easy to read and maintain
- ISO Standard
- Portable – nvc++, g++, icpc, MSVC, …

```cpp
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++

C++ STANDARD PARALLELISM

Lulesh Performance

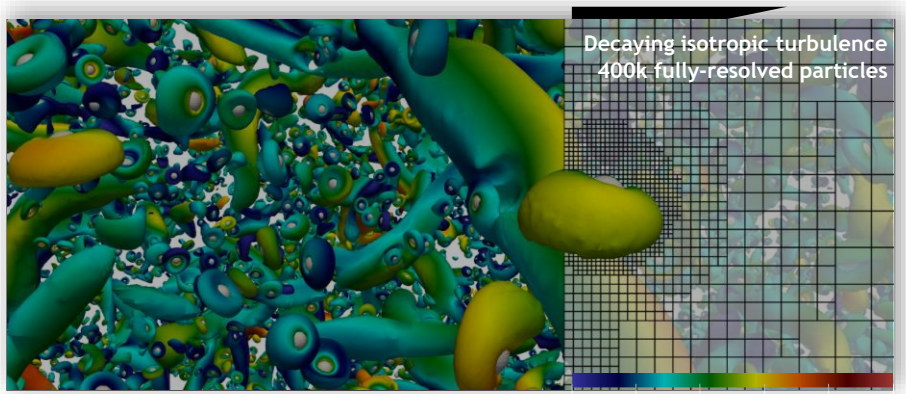


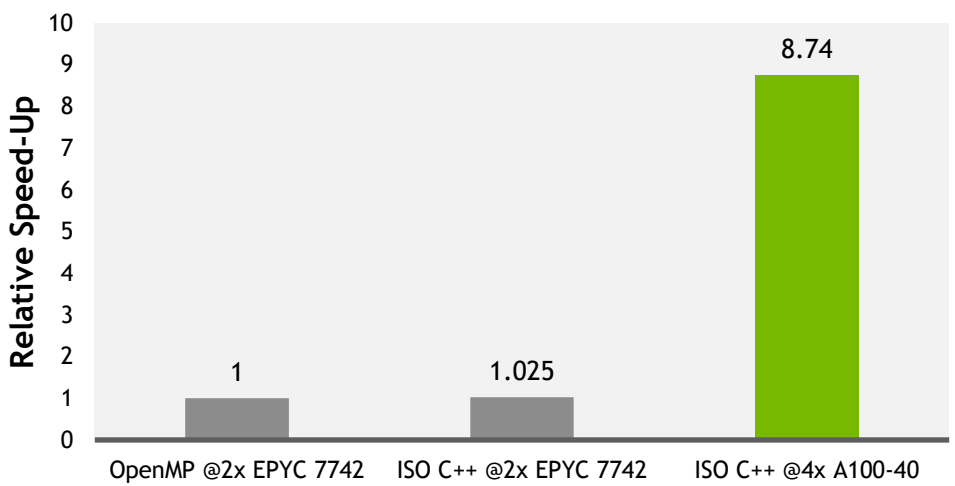
AMD EPYC 7742 CPU, NVIDIA A100 GPU. g++ version 10.3.0, icpc version 2021.5.0, nvc++ version 22.3

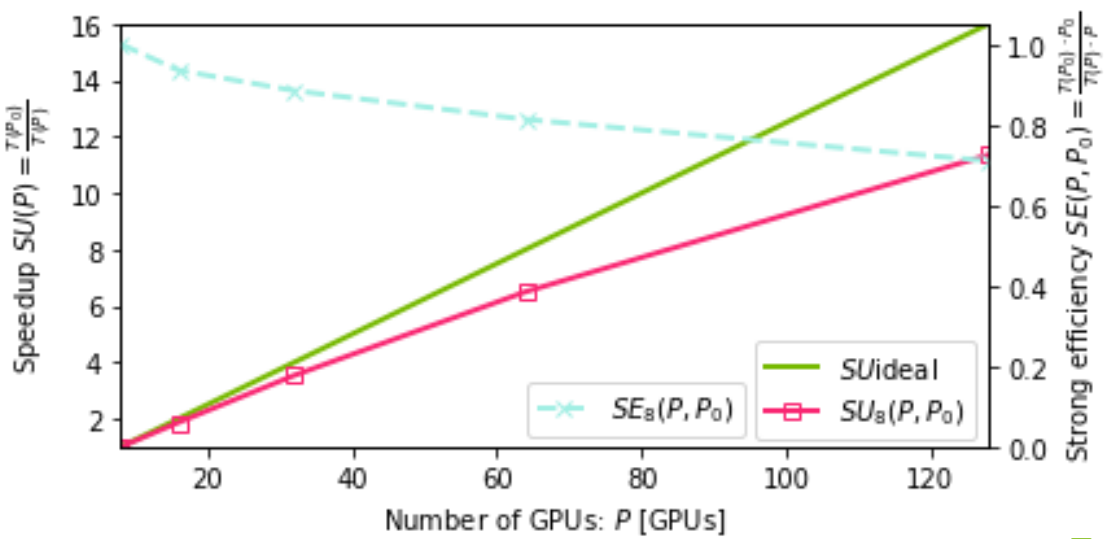
M-AIA

Multi-physics simulation framework developed at the Institute of Aerodynamics, RWTH Aachen University

- Hierarchical grids, complex moving geometries
- Adaptive meshing, load balancing
- Numerical methods: FV, DG, LBM, FEM, Level-Set, ...
- Physics: aeroacoustics, combustion, biomedical, ...
- Developed by ~20 PhDs (Mech. Eng.), ~500k LOC++
- **Programming model:** MPI + **ISO C++ parallelism**





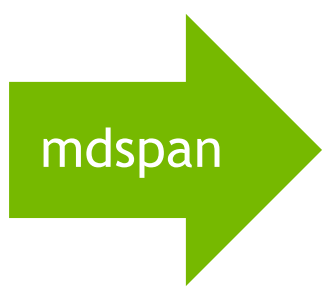


NVIDIA

WHAT'S COMING IN C++: MDSPAN

C++23 brings mdspan, a standardized way to access multi-dimensional data, which composes well with ranges.

```
std::span A{input,  N * M};
std::span B{output, M * N};

auto v = std::ranges::views::cartesian_product(
  std::ranges::views::iota(0, N),
  std::ranges::views::iota(0, M));

std::for_each(std::execution::par_unseq,
  begin(v), end(v),
  [=] (auto idx) {
    auto [i, j] = idx;
    B[i + j * N] = A[i * M + j];
  });
```




```
std::mdspan A{input,  N, M};
std::mdspan B{output, M, N};

auto v = std::ranges::views::cartesian_product(
  std::ranges::views::iota(0, N),
  std::ranges::views::iota(0, M));

std::for_each(ex::par_unseq,
  begin(v), end(v),
  [=] (auto idx) {
    auto [i, j] = idx;
    B[i, j] = A[i, j];
  });
```

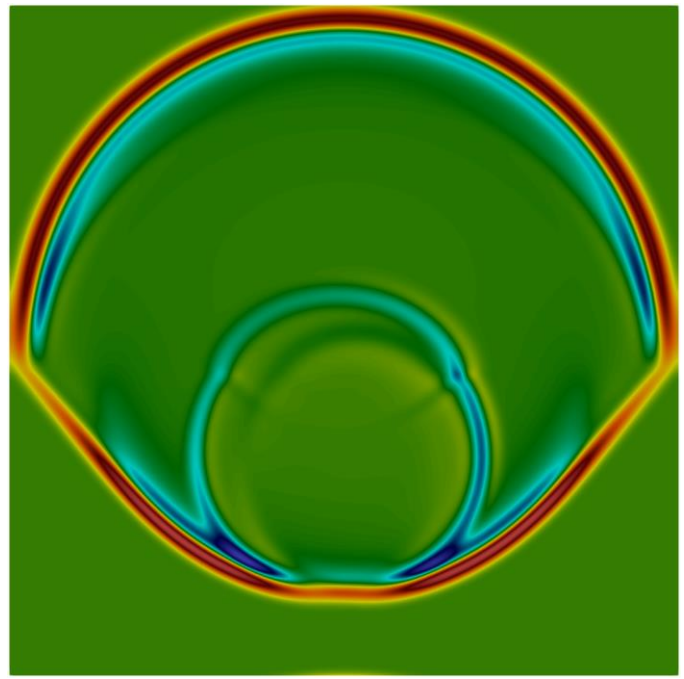
SENDERS & RECEIVERS

Maxwell's equations

```
template <ComputeSchedulerT, WriteSchedulerT>

auto maxwell_eqs(ComputeSchedulerT &scheduler, WriteSchedulerT
&writer)

{

  return repeat_n(

            n_outer_iterations,

            repeat_n(

                n_inner_iterations,

                schedule(scheduler)

              | bulk(grid.cells, update_h(accessor))

              | bulk(grid.cells, update_e(time, dt, accessor)))

            | transfer(writer)

            | then(dump_results(report_step, accessor))

            | then([]{ printf("simulation complete\n"); })

          );

}
```

Simplify Work Across CPUs and Accelerators

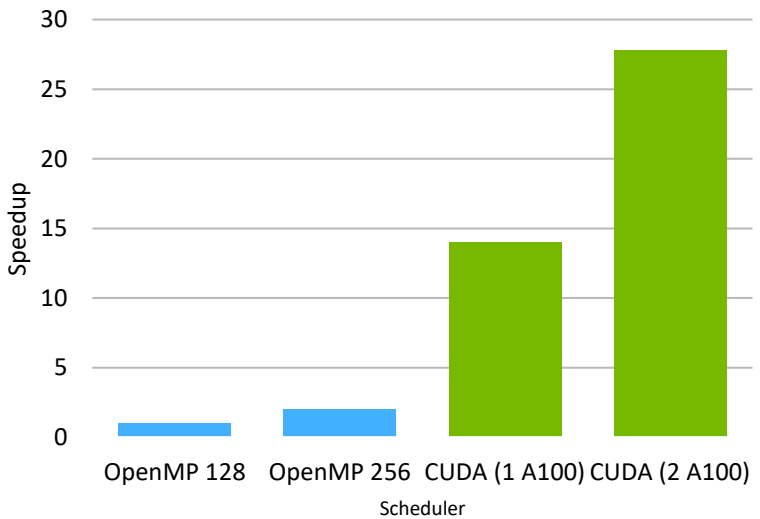
- Uniform abstraction between code and diverse resources
- ISO standard
- Write once, run everywhere

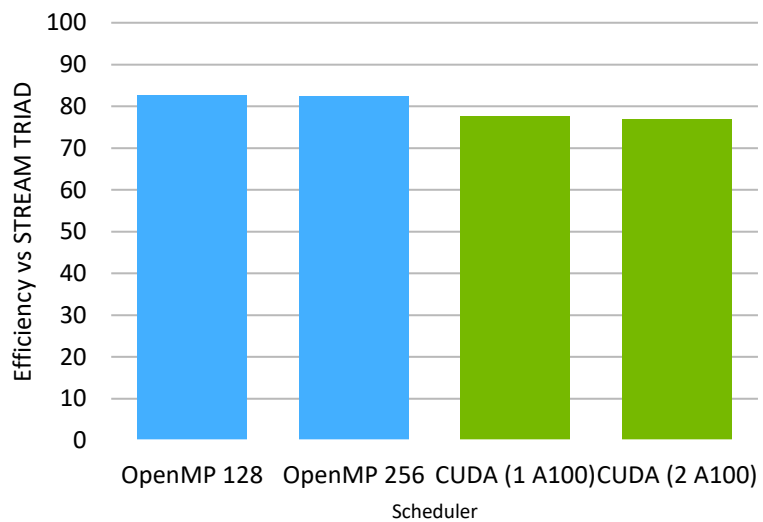


ELECTROMAGNETISM

Raw performance & % of peak

```
std::sync_wait(maxwell(inline_scheduler, inline_scheduler));
std::sync_wait(maxwell(openmp_scheduler, inline_scheduler));
std::sync_wait(maxwell(cuda,             inline_scheduler));
```

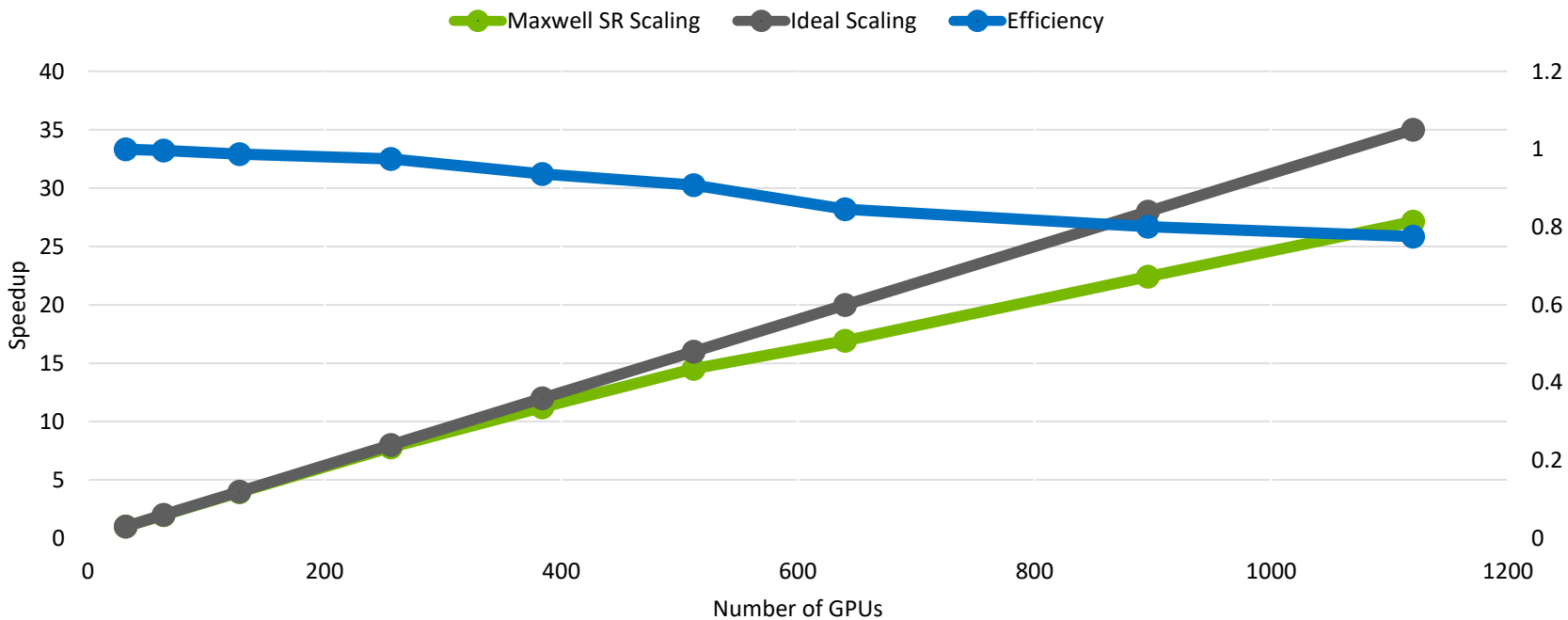




- CPUs: AMD EPYC 7742 CPUs, GPUs: NVIDIA A100-SXM4-80
- Inline (1 CPU HW thread), OpenMP-128 (1x CPU), OpenMP-256 (2x CPUs), Graph (1x GPU), Multi-GPU (2x GPUs)
- clang-12 with -O3 -DNDEBUG -mtune=native -fopenmp

STRONG SCALING USING ISO STANDARD C++

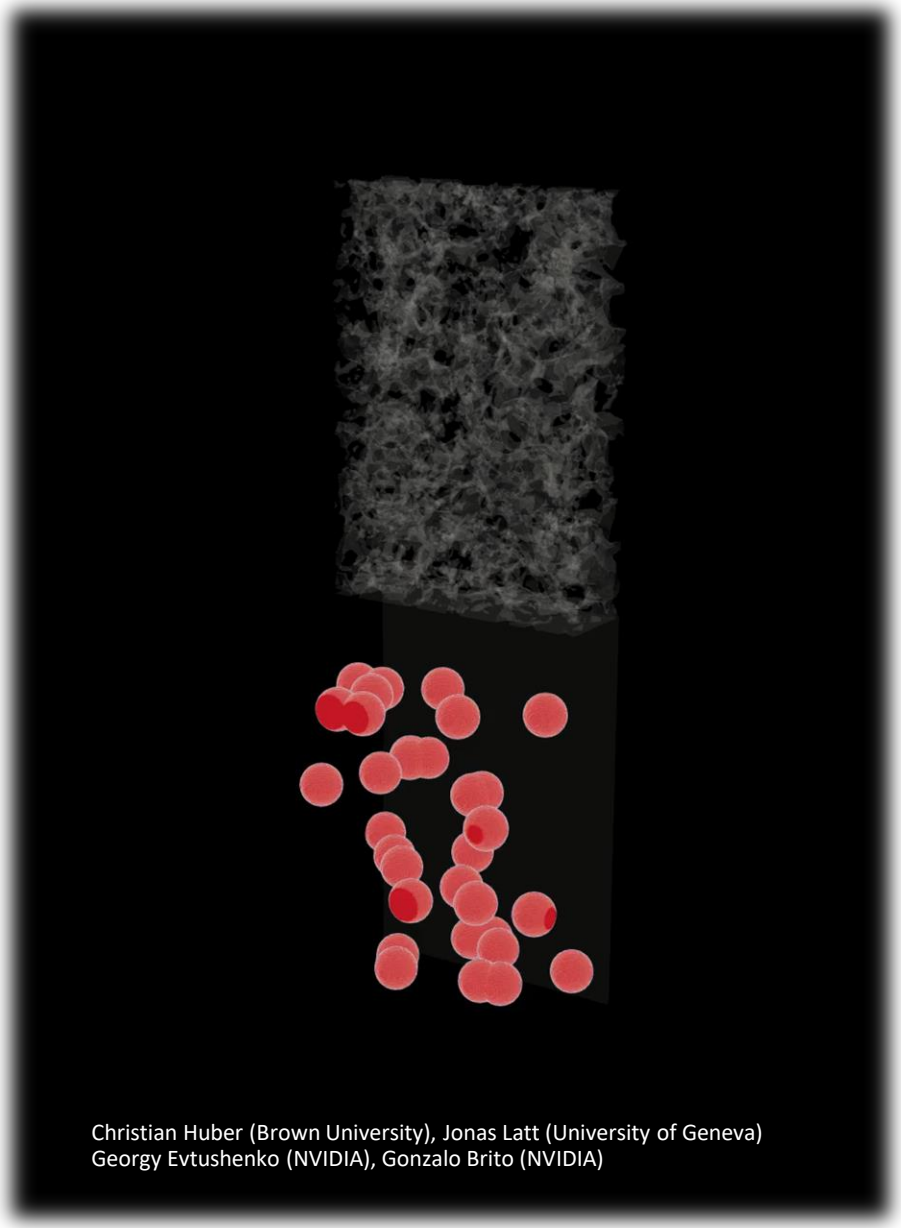
PARALLEL ALGORITHMS AND SENDERS & RECIEVERS



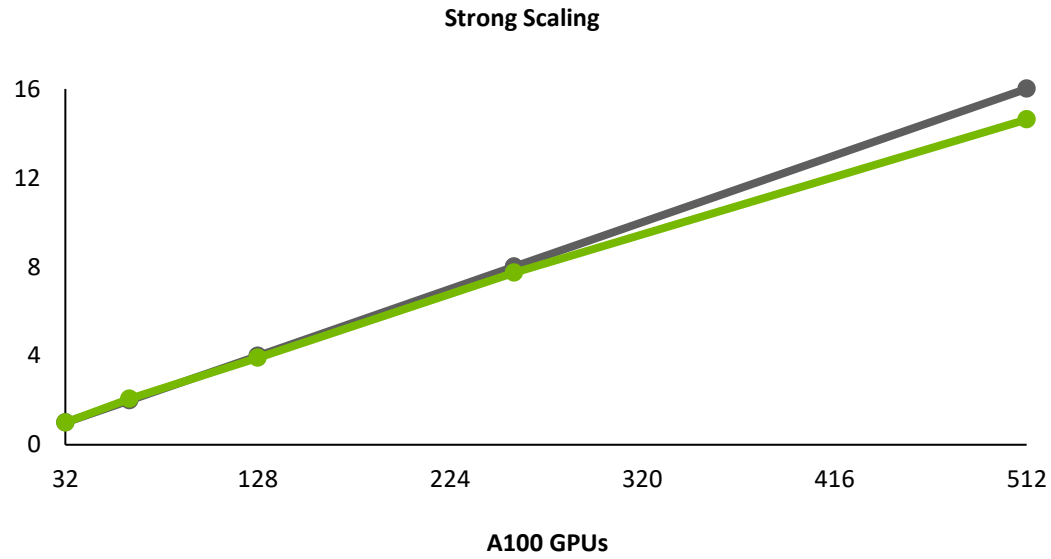
NVIDIA SUPERPOD

- 140x NVIDIA DGX-A100 640
- 1120x NVIDIA A100-SXM4-80 GPUs

PALABOS CARBON SEQUESTRATION



Christian Huber (Brown University), Jonas Latt (University of Geneva)
Georgy Evtushenko (NVIDIA), Gonzalo Brito (NVIDIA)



- Palabos is a framework for fluid dynamics simulations using Lattice-Boltzmann methods.
- Code for multi-component flow through a porous media ported to C++ Senders and Receivers.
- Application: simulating carbon sequestration in sandstone.


PARALLEL PROGRAMMING WITH ISO
FORTRAN

HPC PROGRAMMING IN ISO FORTRAN

ISO is the place for portable concurrency and parallelism

Fortran 2018

Array Syntax and 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

Preview support available now in NVFORTRAN

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

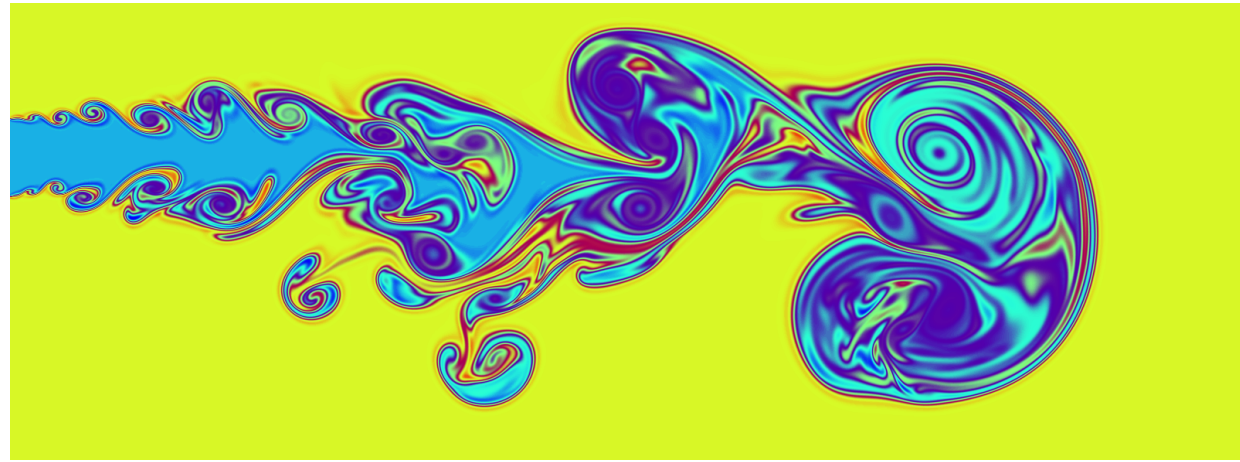
Mini-App written in C++ and Fortran that simulates weather-like fluid flows using Finite Volume and Runge-Kutta methods.

Existing parallelization in MPI, OpenMP, OpenACC, …

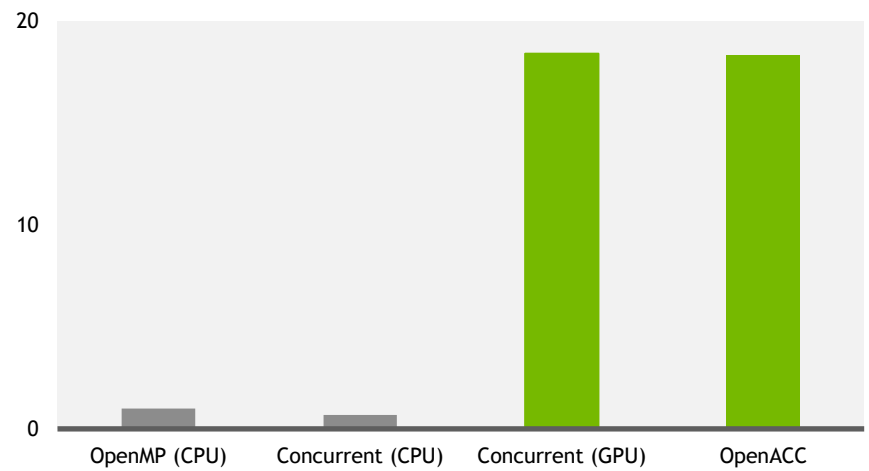
Included in the SPEChpc benchmark suite*

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
```



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

*SPEChpc is a trademark of The Standard Performance Evaluation Corporation

NVIDIA

POT3D: DO CONCURRENT + LIMITED OPENACC

POT3D

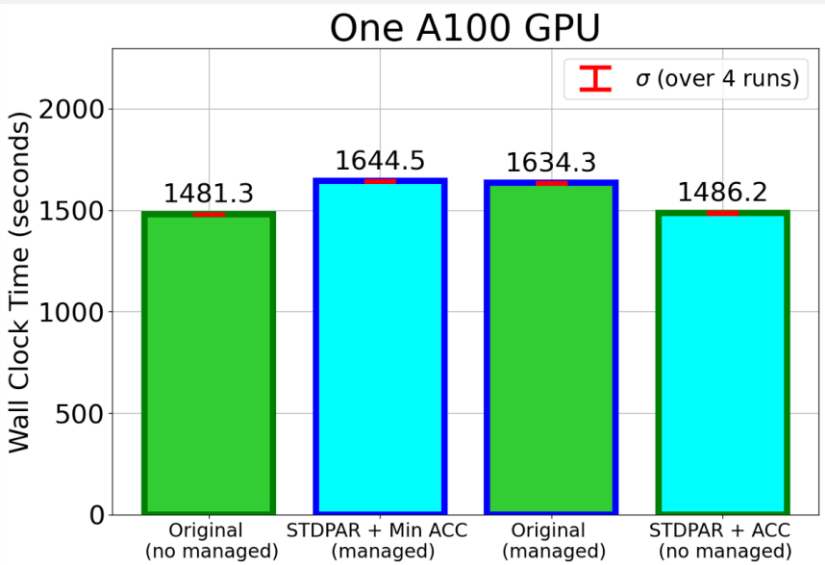
POT3D is a Fortran application for approximating solar coronal magnetic fields.

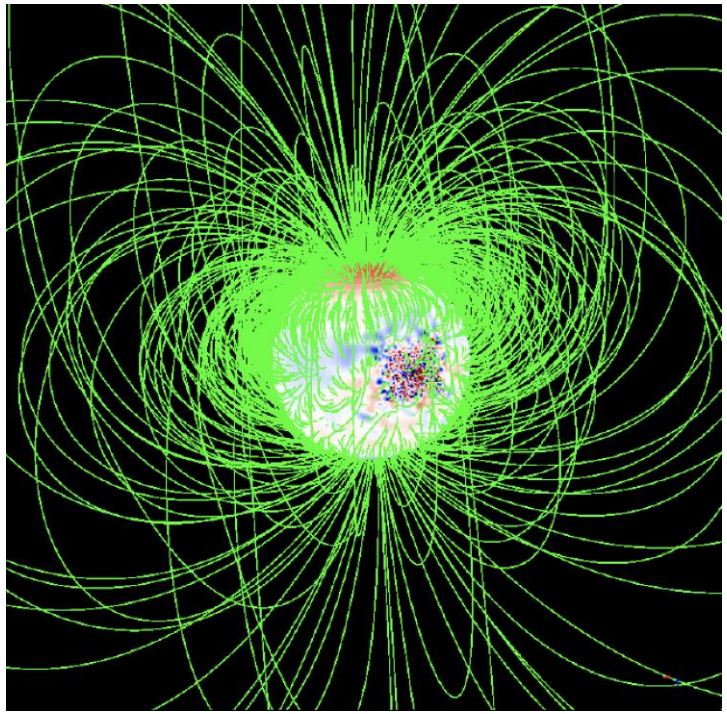
Included in the SPEChpc benchmark suite*

Existing parallelization in MPI & OpenACC

Optimized the DO CONCURRENT version by using OpenACC solely for data motion and atomics

https://github.com/predsci/POT3D





```
!$acc enter data copyin(phi,dr_i)
!$acc enter data create(br)
do concurrent (k=1:np,j=1:nt,i=1:nrm1)
  br(i,j,k)=(phi(i+1,j,k)-phi(i,j,k ))*dr_i(i)
enddo
!$acc exit data delete(phi,dr_i,br)
```

Data courtesy of Predictive Science Inc.

*SPEChpc is a trademark of The Standard Performance Evaluation Corporation

ACCELERATED PROGRAMMING IN ISO FORTRAN

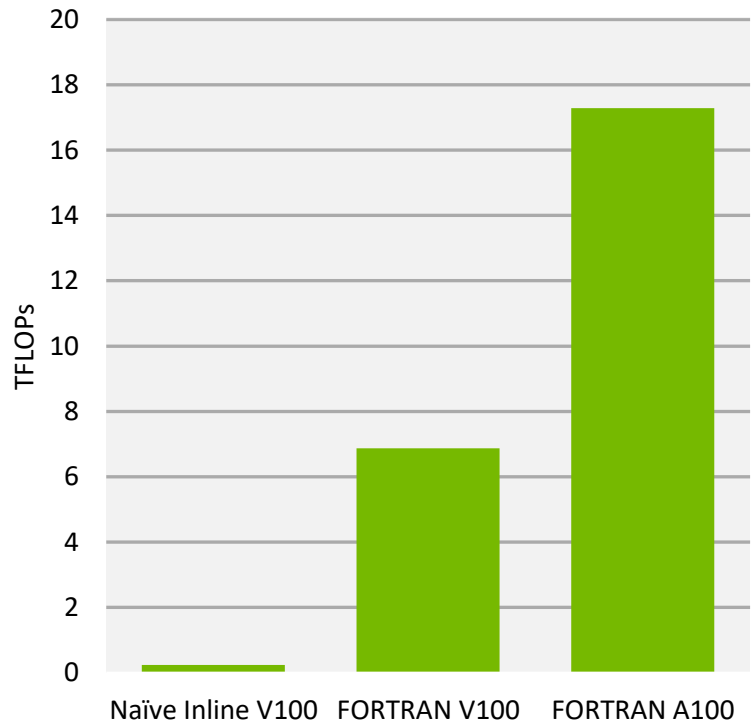
NVFORTRAN Accelerates Fortran Intrinsics with cuTENSOR Backend

```
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



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
```


STANDARD PARALLELISM FOR PYTHON

PRODUCTIVITY & PERFORMANCE

Sequential and Composable Code

```
def cg_solve(A, b, conv_iters):
    x = np.zeros_like(b)
    r = b - A.dot(x)
    p = r
    rsold = r.dot(r)
    converged = False
    max_iters = b.shape[0]

    for i in range(max_iters):
        Ap = A.dot(p)
        alpha = rsold / (p.dot(Ap))
        x = x + alpha * p
        r = r - alpha * Ap
        rsnew = r.dot(r)

        if i % conv_iters == 0 and \
            np.sqrt(rsnew) < 1e-10:
            converged = i
            break

        beta = rsnew / rsold
        p = r + beta * p
        rsold = rsnew
```

- Sequential semantics - no visible parallelism or synchronization
- Name-based global data – no partitioning
- Composable – can combine with other libraries and datatypes

- **This code can run on a supercomputer**

CUNUMERIC

Automatic NumPy Acceleration and Scalability

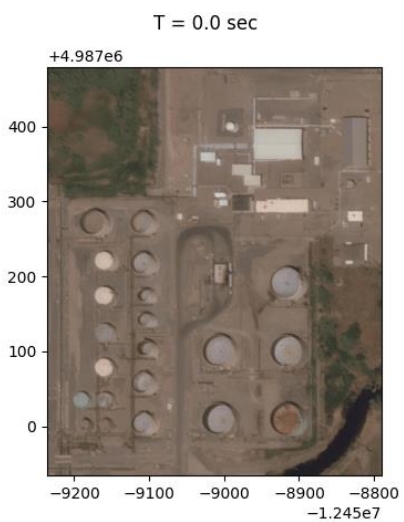
cuNumeric

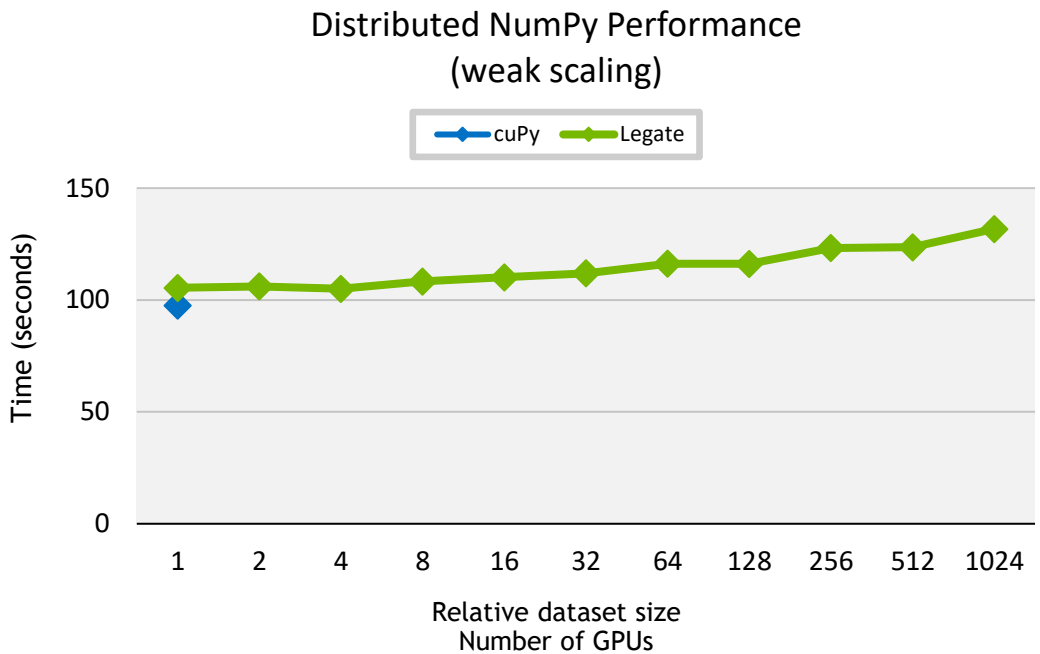
CuNumeric transparently accelerates and scales existing Numpy workloads

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



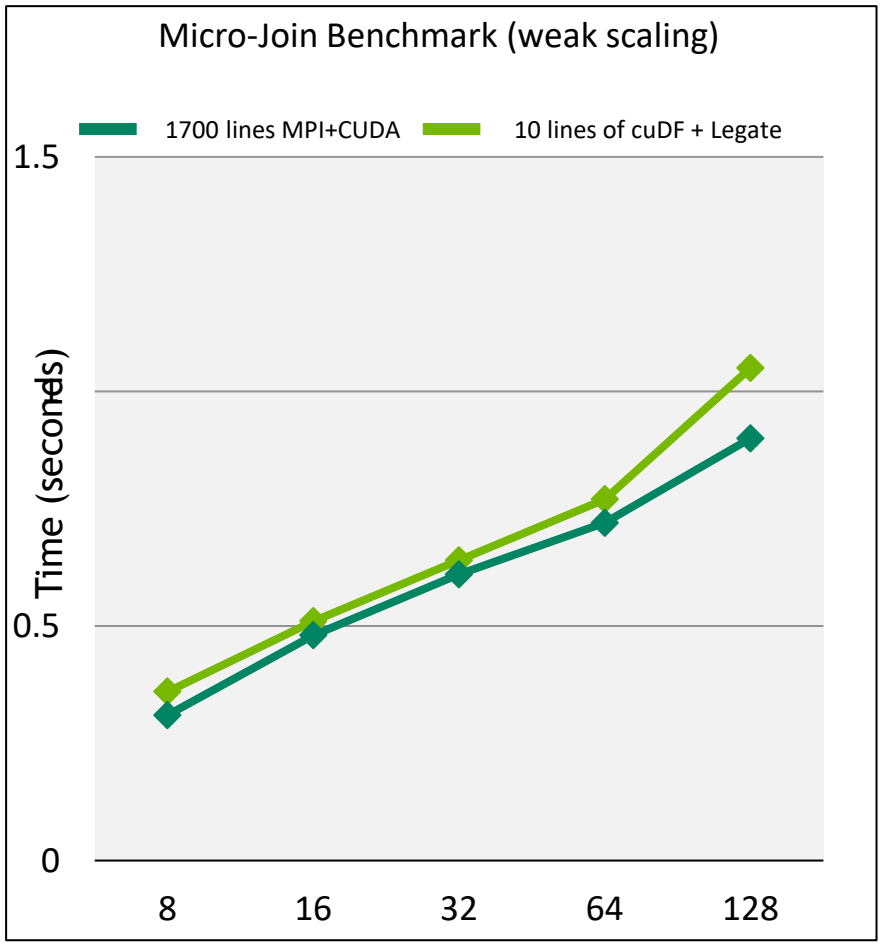



```
for _ in range(iter):
    un = u.copy()

    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

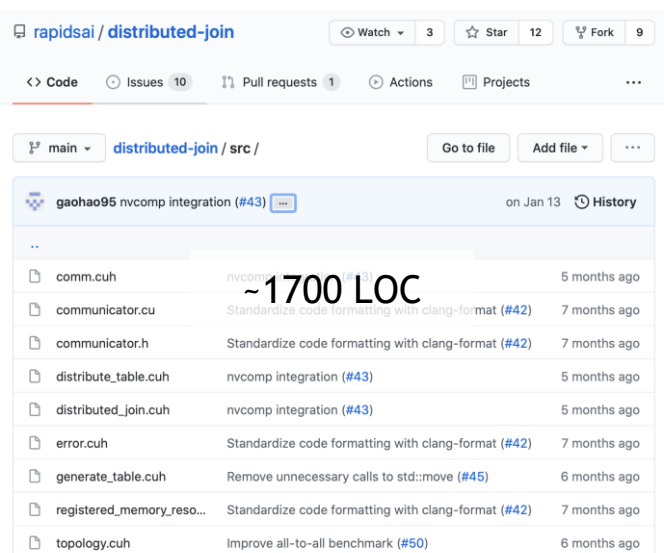
NEAR-MPI PERFORMANCE IN ~10 LINES OF CODE



Machine: DGX SuperPOD with A100-80GB GPUs

```
size = num_rows_per_gpu * num_gpus

key_l = np.arange(size)
val_l = np.random.randn(size)
lhs = pd.DataFrame({ "key": key_l, ”val": val_l })

key_r = key_l // 3 * 3    # selectivity: 0.33
payload_r = np.random.randn(size)
rhs = pd.DataFrame({ "key": key_r, ”val": val_r })

out = lhs.merge(rhs, on="key")
```

vs.



RICHARDSON-LUCY DECONVOLUTION

How to Overcome the Limits of Diffraction with Computation

Richardson-Lucy deconvolution is an iterative algorithm for finding “true” image from a known PSF

NumPy implementation in scikit-image

One minor change: NumPy’s convolve function is 1-D only, but cuNumeric’s convolve supports N-d arrays (using the same interface)

cuNumeric provides implementations of all these NumPy functions

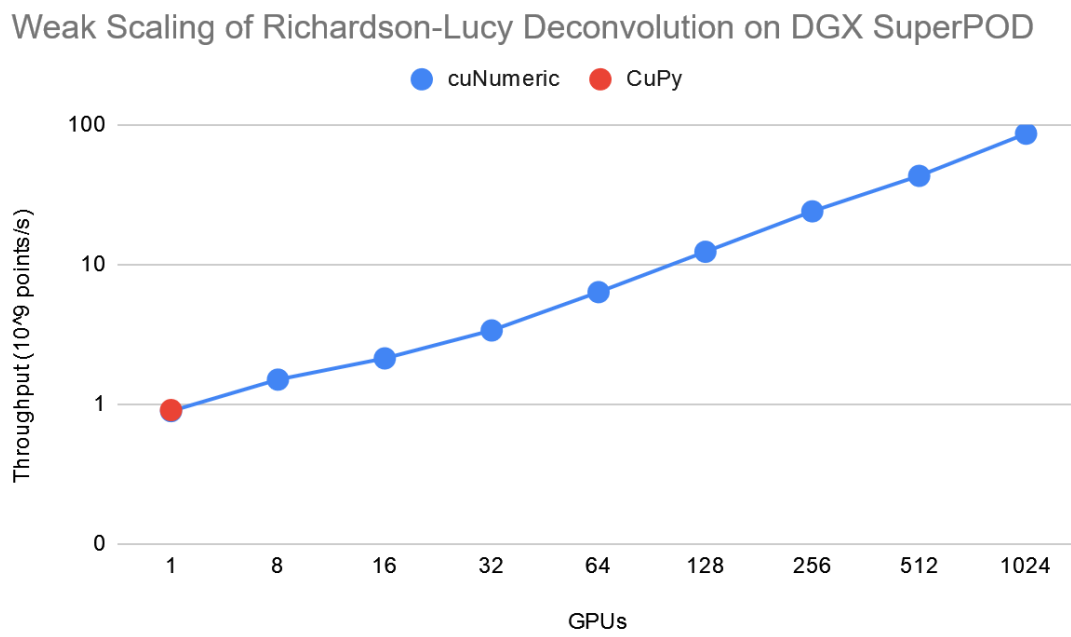
```
def richardson_lucy(image, psf, num_iter=50,
                    clip=True, filter_epsilon=None):
  float_type = _supported_float_type(image.dtype)
  image = image.astype(float_type, copy=False)
  psf = psf.astype(float_type, copy=False)
  im_deconv = np.full(image.shape, 0.5, dtype=float_type)
  psf_mirror = np.flip(psf)

  for _ in range(num_iter):
    conv = convolve(im_deconv, psf, mode='same’)
      if filter_epsilon:
         with np.errstate(invalid='ignore’):
            relative_blur = np.where(conv < filter_epsilon, 0,
                                     image / conv)
      else:
         relative_blur = image / conv
      im_deconv *= convolve(relative_blur, psf_mirror,
                            mode='same’)

   if clip:
      im_deconv[im_deconv > 1] = 1
      im_deconv[im_deconv < -1] = -1

  return im_deconv
```

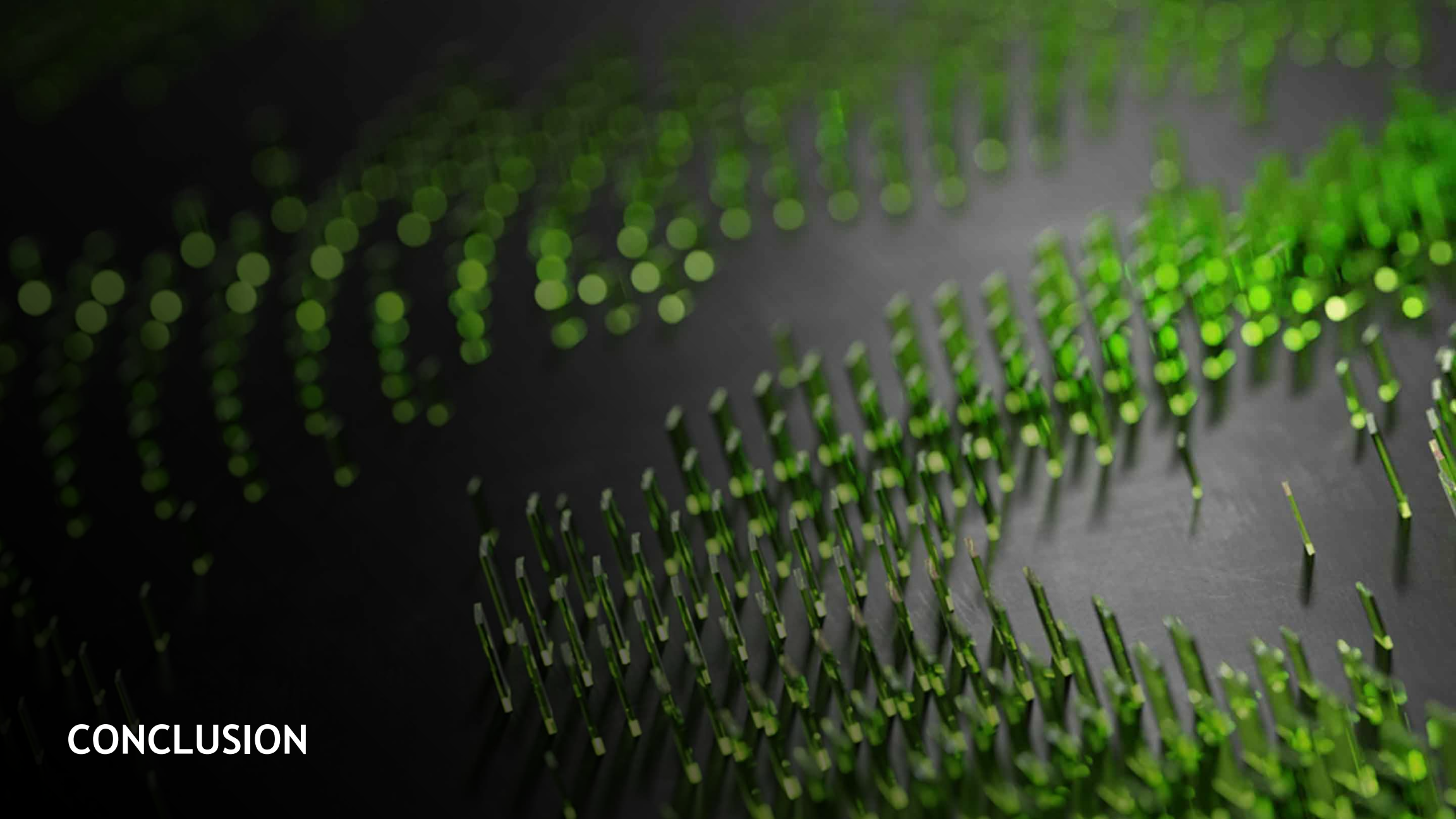
PERFORMANCE RESULTS

Nearly perfect weak-scaling

Experiment: 134M FP64 Pixels/GPU, PSF: 512x256x151, on DGX SuperPOD A100

More GPUs can be used to handle more detailed data coming off increasingly powerful microscopes





CONCLUSION

GTC SPRING 2022 SESSIONS TO REWATCH

For more information on these topics

- No More Porting: Coding for GPUs with Standard C++, Fortran, and Python [S41496]
- A Deep Dive into the Latest HPC Software [S41494]
- 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]

STANDARD PARALLELISM RESOURCES

NVIDIA Developer Blogs

- Developing Accelerated Code with Standard Language Parallelism
- Accelerating Standard C++ with GPUs
- Accelerating Fortran DO CONCURRENT
- Bringing Tensor Cores to Standard Fortran
- Accelerating Python on GPUs with NVC++ and Cython

Legate and cuNumeric Resources

- https://github.com/nv-legate

Open-source codes

- LULESH - https://github.com/LLNL/LULESH
- STLBM - https://gitlab.com/unigehpfs/stlbm
- MiniWeather - https://github.com/mrnorman/miniWeather/
- POT3D - https://github.com/predsci/POT3D

C++ algorithms and execution policy reference

- https://en.cppreference.com/w/cpp/algorithm

NVIDIA HPC Compilers Forum

- https://forums.developer.nvidia.com/c/accelerated-computing/hpc-compilers


NVIDIA
Feel free to reach out
jlarkin@nvidia.com