GPU programming using C++, Fortran and Python

Jeff Hammond
NVIDIA HPC Software Group

Outline

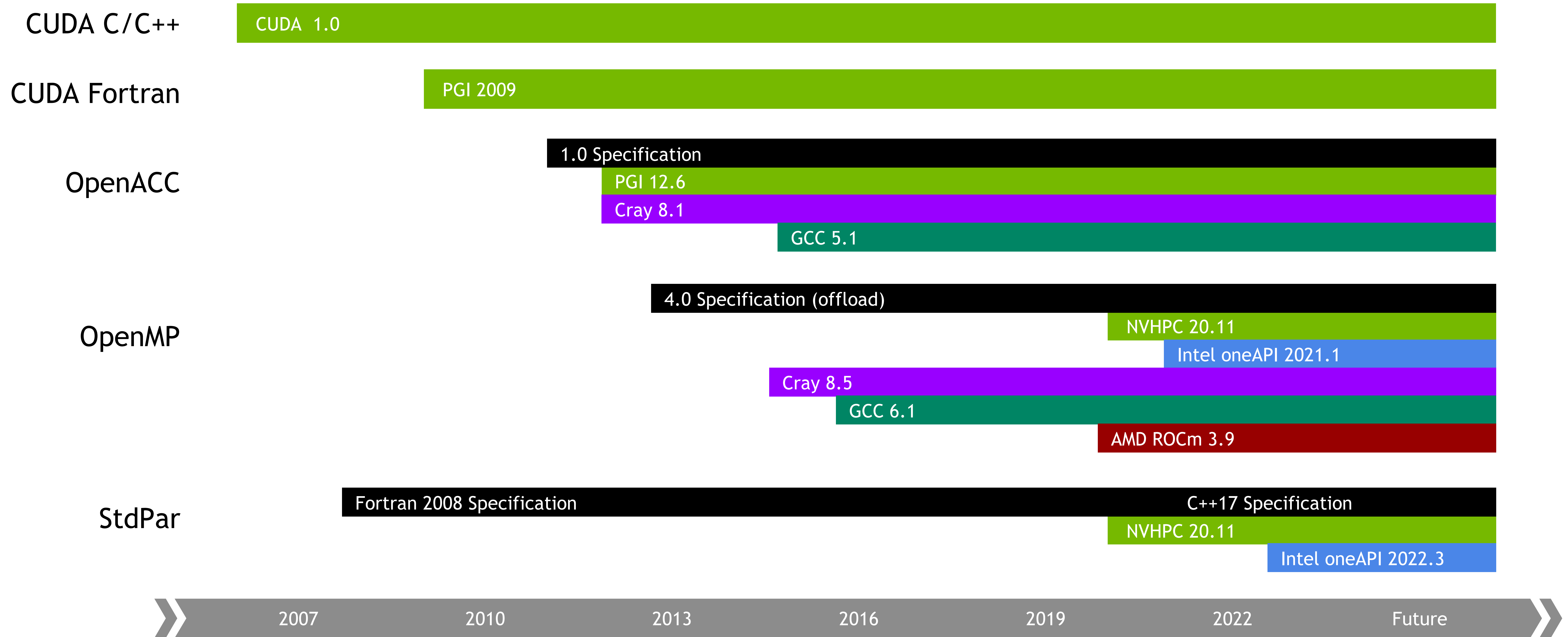
- Overview of GPU programming in C++, Fortran and Python
- Application examples of successes using standard parallelism
- Comparison of models using basic linear algebra operations
- Quantum chemistry results

NVIDIA.

Overview

Programming Models for GPUs

A brief, possibly incomplete, history



Programming the NVIDIA platform *with C++*

CPU, GPU, and Network

ACCELERATED STANDARD MODELS

ISO C++ | OpenACC | OpenMP

PLATFORM SPECIALIZATION

CUDA C/C++

```
using namespace std;
using namespace execution;

auto lambda = [a](auto&& x,
                  auto&& y) {
    return x + a * y;
};

transform(par_unseq,
    begin(Y), end(Y),
    begin(X), begin(Y),
    lambda);
```

```
#pragma acc parallel loop
for (int i=0; i<n; i++) {
    Y[i] += a * X[i];
}
```

```
#pragma omp target &
        teams distribute &
        parallel do &
        simd
for (int i=0; i<n; i++) {
    Y[i] += a * X[i];
}

#pragma omp target teams loop
for (int i=0; i<n; i++) {
    Y[i] += a * X[i];
}
```

```
__global__
template <typename T>
void axpy(int n, T a, T *x, T *y)
{
  int i = blockIdx.x*blockDim.x +
          threadIdx.x;
  if (i < n) y[i] += a*x[i];
}

int main(void) {
  ...
  saxpy<<<(N+255)/256,256>>>(...);
  ...
}
```

ACCELERATION LIBRARIES

CUDA Runtime | CUBLAS | CUTENSOR | CUSOLVER | ... | NVSHMEM

Programming the NVIDIA platform *with Fortran*

CPU, GPU, and Network

ACCELERATED STANDARD MODELS

ISO Fortran | OpenACC | OpenMP

PLATFORM SPECIALIZATION

CUDA Fortran

ISO Fortran

```
do concurrent (j=1:order, &
               i=1:order)
  B(i,j) = A(j,i)
enddo
```

```
B = transpose(A)
```

OpenACC

```
!$acc parallel loop tile(32,32)
do j=1,order
  do i=1,order
     B(i,j) = A(j,i)
  enddo
enddo
```

```
!$acc kernels
do j=1,order
  do i=1,order
     B(i,j) = A(j,i)
  enddo
enddo
!$acc end kernels
```

OpenMP

```
!$omp target teams distribute &
      parallel do simd &
      collapse(2)
do j=1,order
  do i=1,order
    B(i,j) = A(j,i)
  enddo
enddo
```

```
!$omp target teams loop &
      collapse(2)
do j=1,order
  do i=1,order
    B(i,j) = A(j,i)
  enddo
enddo
```

CUDA Fortran

```
BIDX = blockIdx%x-1
BIDY = blockIdx%y-1
TIDX = threadIdx%x
TIDY = threadIdx%y

x = BIDX * TILE + TIDX;
y = BIDY * TILE + TIDY;
do j = 0,TILE-1,block_rows
  SM(TIDX,TIDY+j) = A(x,y+j);
end do

call syncThreads()

x = BIDY * TILE + TIDX;
y = BIDX * TILE + TIDY;
do j = 0,TILE-1,block_rows
  B(x,y+j) = SM(TIDY+j,TIDX)
end do
```

ACCELERATION LIBRARIES

CUDA Runtime | CUBLAS | CUTENSOR | CUSOLVER | ... | NVSHMEM

NVIDIA.

Programming the NVIDIA platform *with Python*

CPU, GPU, and Network

ACCELERATED STANDARD MODELS

NumPy ~ CuPy ~ cuNumeric

```
import cupy as cp

X = cp.arange(0,length,dtype='float64')
Y = cp.arange(0,length,dtype='float64')

# Numpy
Y +=  a * X



# CuPy
saxpy = cp.ElementwiseKernel(
        'float64 a, float64 x, float64 y',
        'y = a * x + y',
        'saxpy_elementwise'
    )

saxpy(a, X, Y)
```

PLATFORM SPECIALIZATION

CUDA Python

```
from cuda import cuda, cudart, nvrtc

C = '''\
  extern "C"
  __global__ void saxpy(int n, double a, double * X, double * Y)
  {
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        if (i < n) {
                Y[i] += a * X[i];
        }
  }
'''

K = KernelHelper(C, int(cuDevice))
S = K.getFunction(b'saxpy')

cuda.cuLaunchKernel(S, blocksPerGrid, 1, 1, threadsPerBlock, 1, 1,
                    0, cuda.CUstream(0), Args, 0))
```

ACCELERATION LIBRARIES

CUDA Runtime | CUBLAS | CUTENSOR | CUSOLVER | ...

NVIDIA.

Application Examples

C++17 parallel algorithms

- 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)
{
#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 ;
}
```

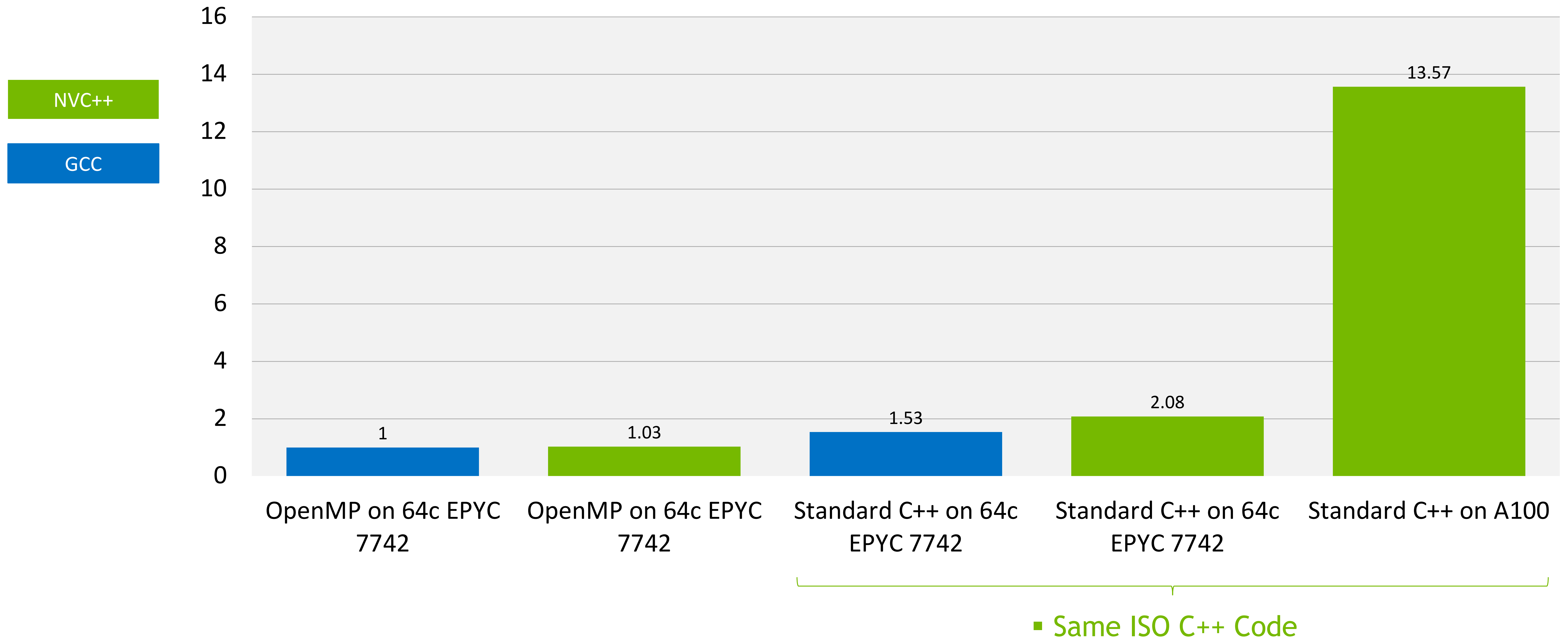
C++ with OpenMP

```cpp
static inline void CalcHydroConstraintForElems(Domain &domain, Index_t le
ngth,
               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



NVIDIA

MiniWeather

Standard Language Parallelism in Climate/Weather Applications

MiniWeather

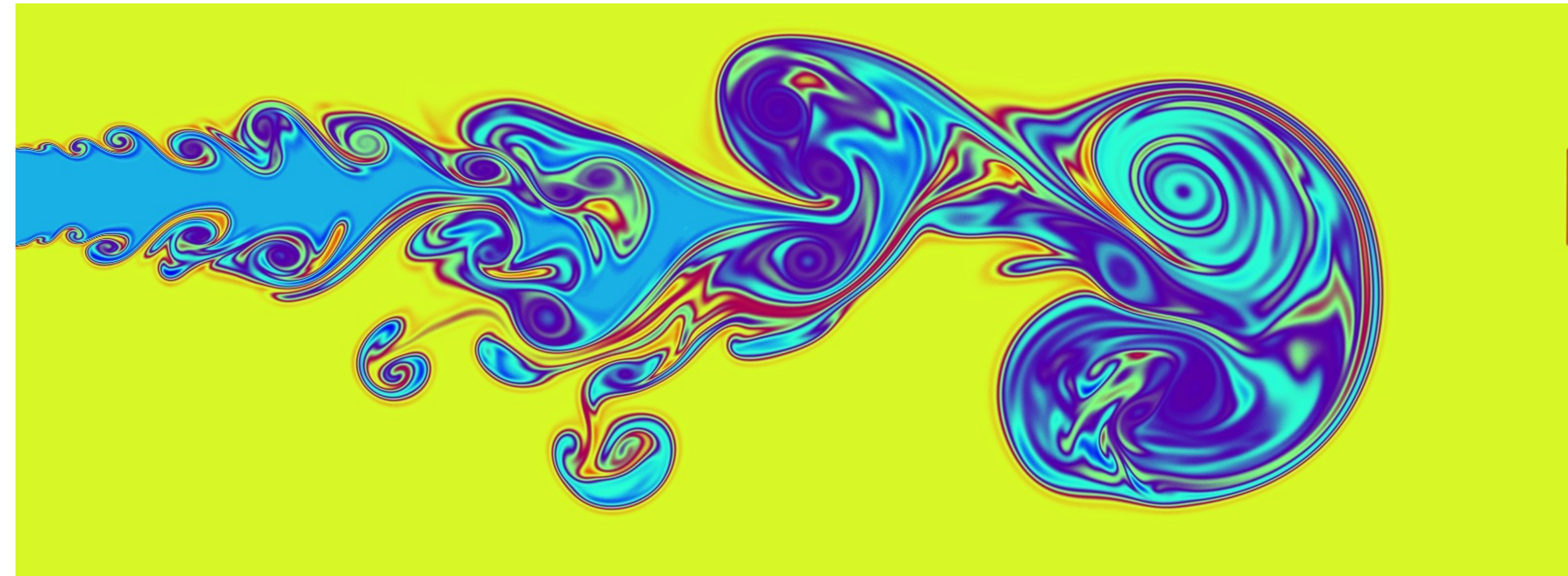
Mini-App written in C++ and Fortran that simulates weather-like fluid flows using Finite Volume and Runge-Kutta methods.

Existing parallelization in MPI, OpenMP, OpenACC, …

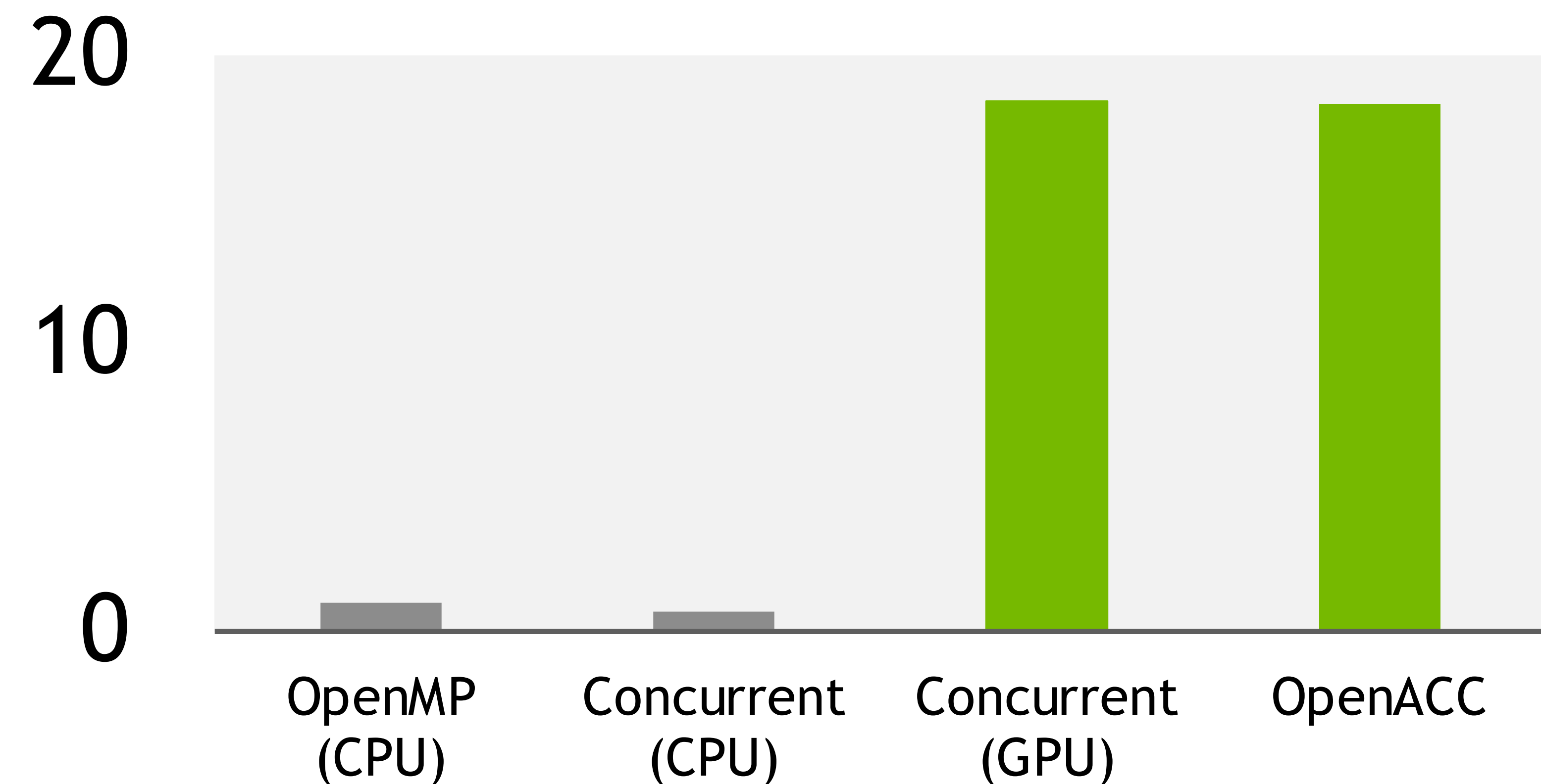
Included in the SPEChpc benchmark suite*

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.

NVIDIA

POT3D: Do Concurrent + Limited OpenACC

POT3D

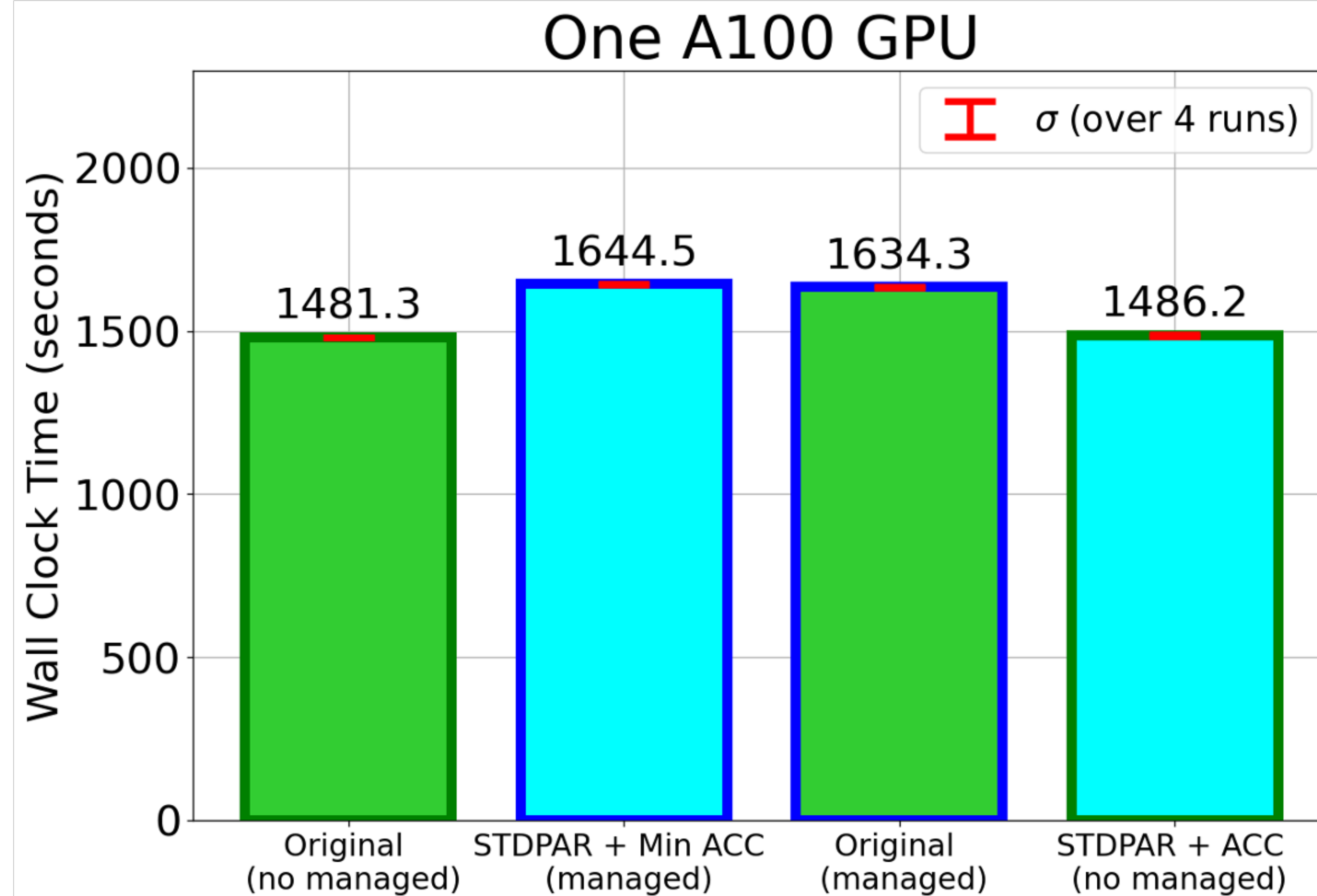
POT3D is a Fortran application for approximating solar coronal magnetic fields.

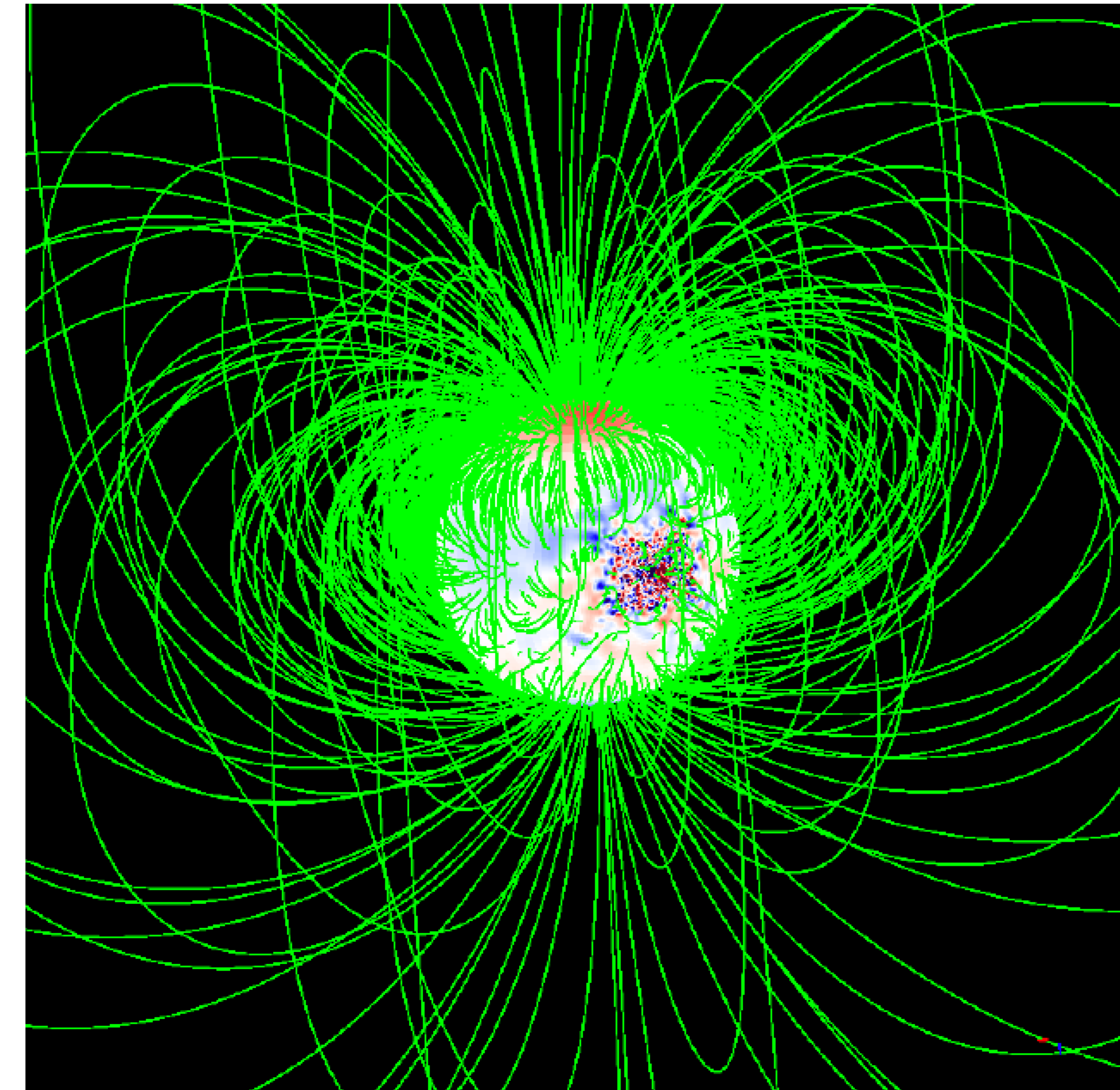
Included in the SPEChpc benchmark suite*

Existing parallelization in MPI & OpenACC

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

https://github.com/predsci/POT3D



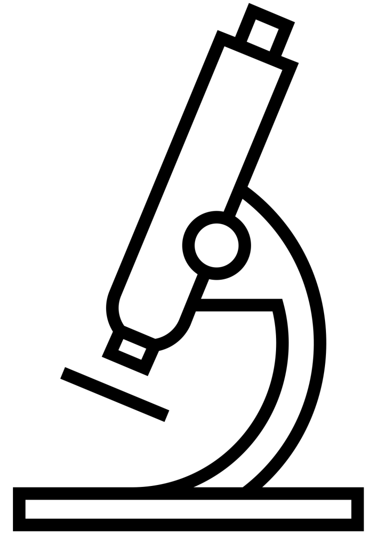


```
!$acc enter data copyin(phi,dr_i)
!$acc enter data create(br)
do concurrent (k=1:np,j=1:nt,i=1:nrm1)
  br(i,j,k)=(phi(i+1,j,k)-phi(i,j,k ))*dr_i(i)
enddo
!$acc exit data delete(phi,dr_i,br)
```

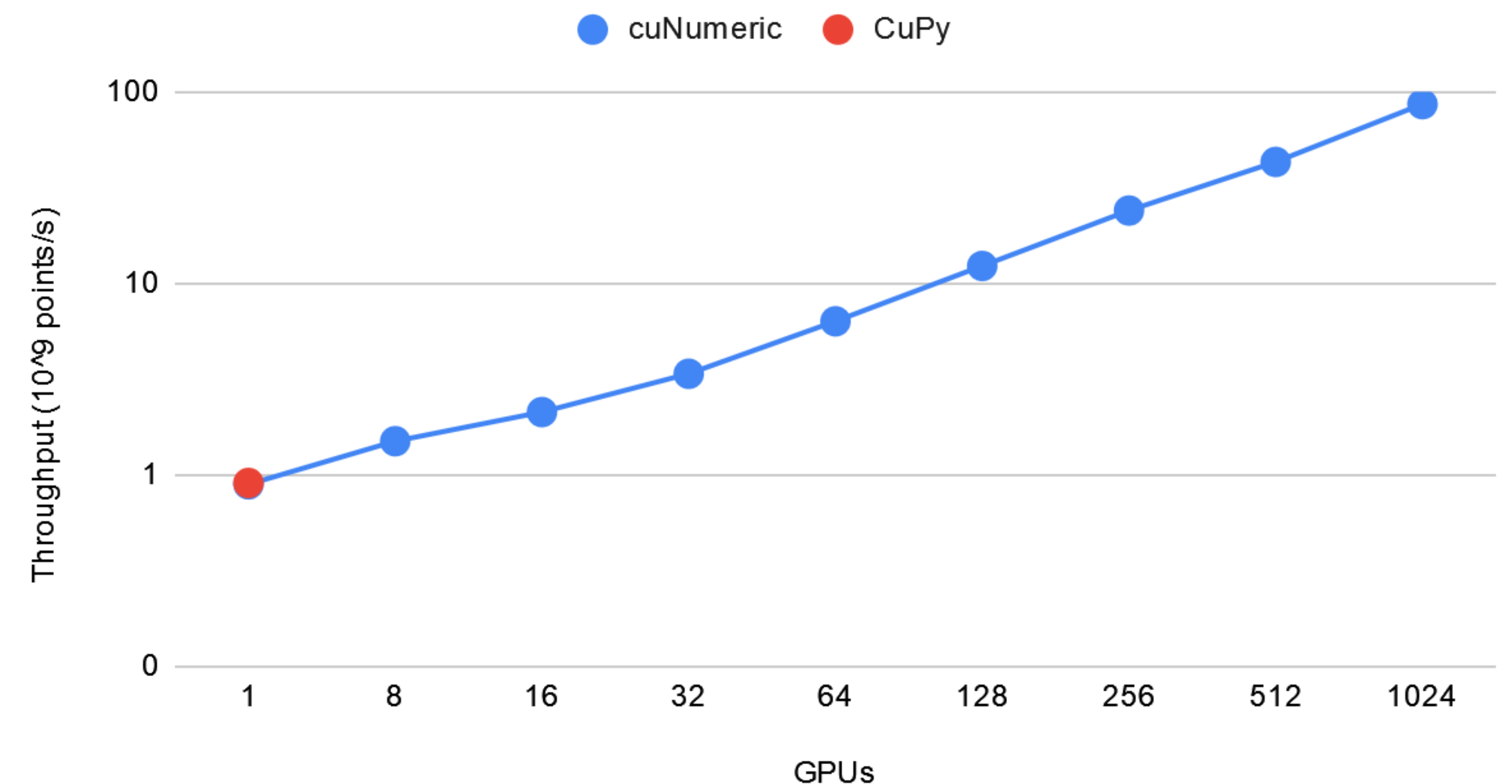
Data courtesy of Predictive Science Inc.

*SPEChpc is a trademark of The Standard Performance Evaluation Corporation

MICROSCOPY WITH RICHARDSON-LUCY DECONVOLUTION



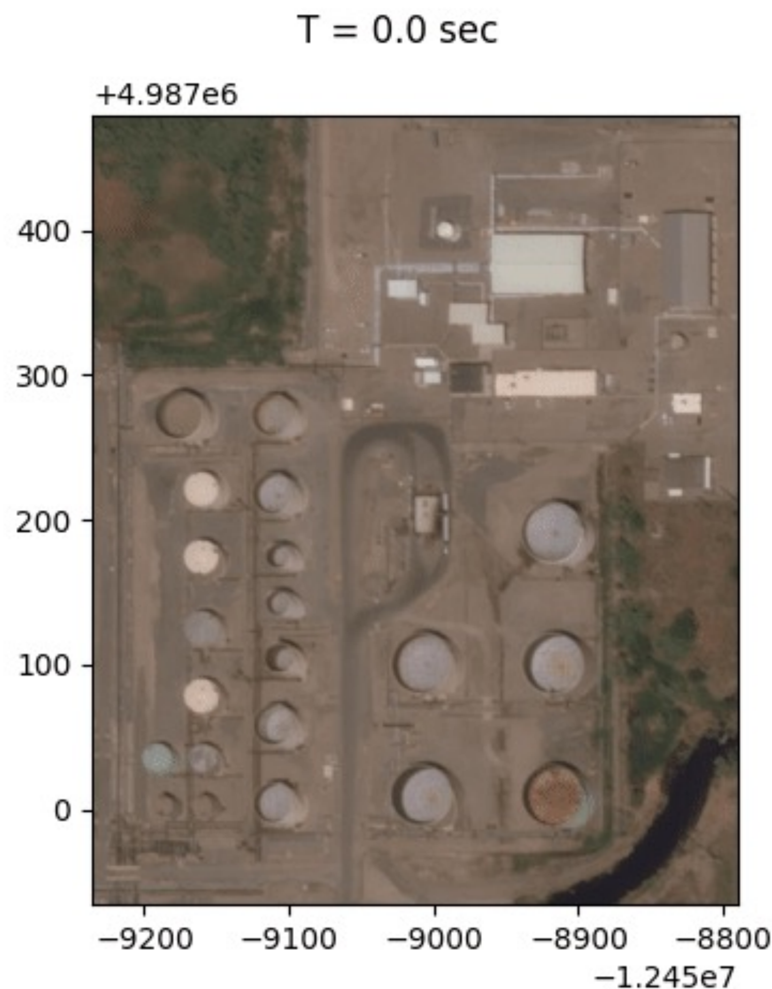
```
def richardson_lucy(image, psf, num_iter=50,
                    clip=True, filter_epsilon=None):
  float_type = _supported_float_type(image.dtype)
  image = image.astype(float_type, copy=False)
  psf = psf.astype(float_type, copy=False)
  im_deconv = np.full(image.shape, 0.5, dtype=float_type)
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



NVIDIA

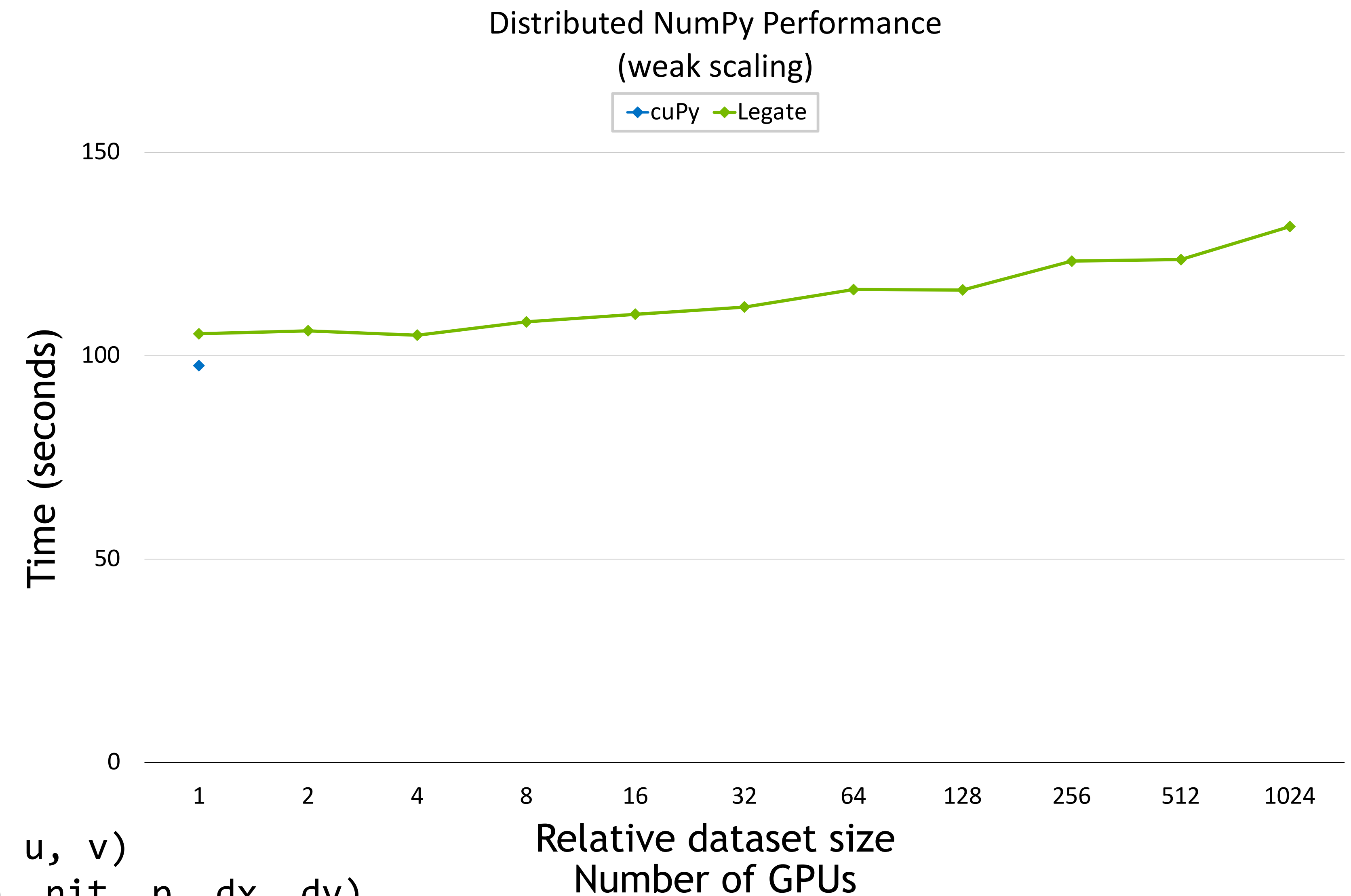
COMPUTATIONAL FLUID DYNAMICS

- CFD codes like:
  - Shallow-Water Equation Solver
- Oil Pipeline Risk Management: Geoclaw-landspill simulations
- Python Libraries: Jupyter, NumPy, SciPy, SymPy, Matplotlib




```
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)

…
```

CFD Python on cuNumeric!



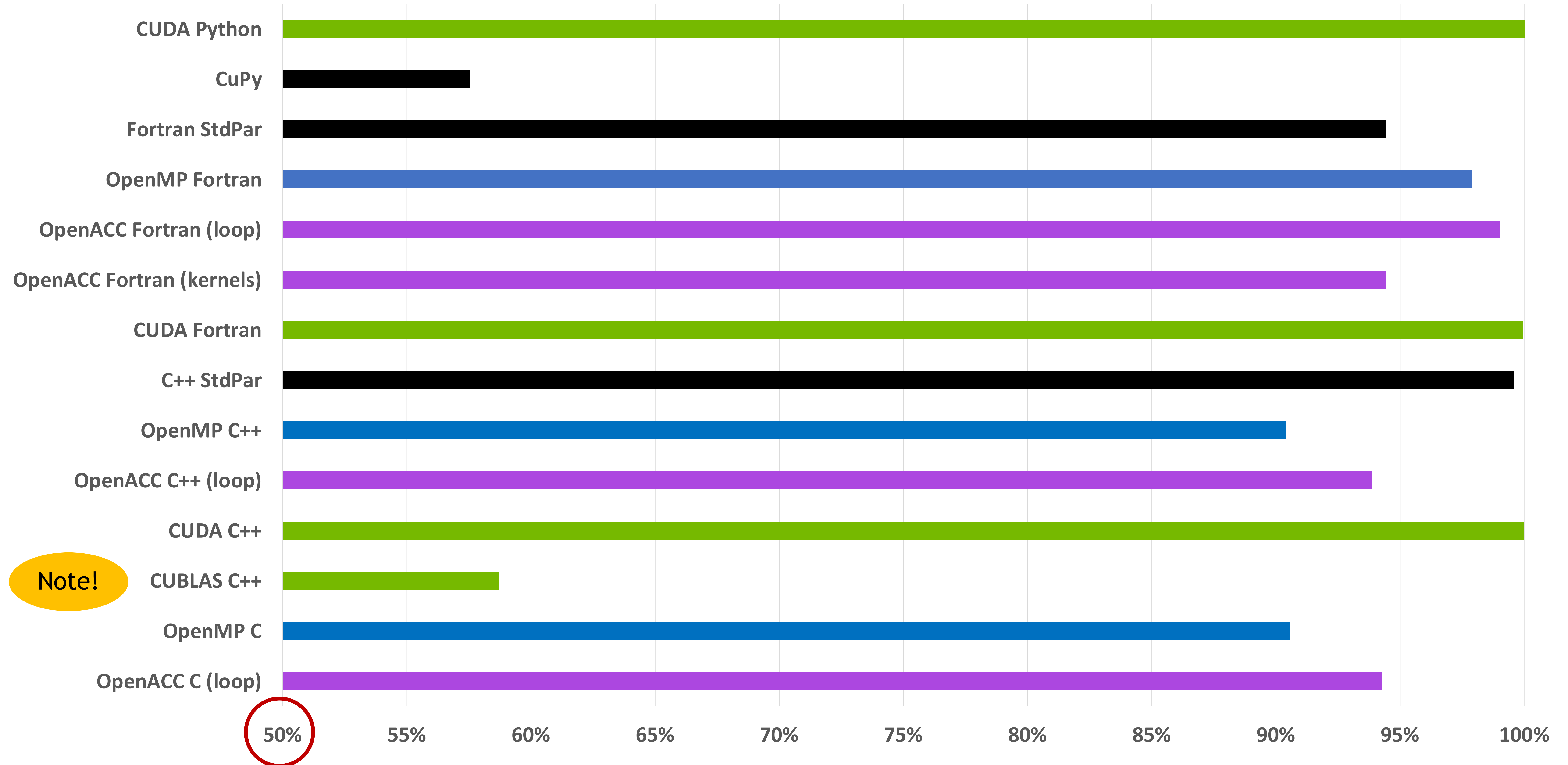
Extracted from "CFD Python" course at https://github.com/barbagroup/CFDPython

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

NVIDIA.

Comparative Analysis

Vector Addition: Z = a * X + Y



NVIDIA

Vector Addition

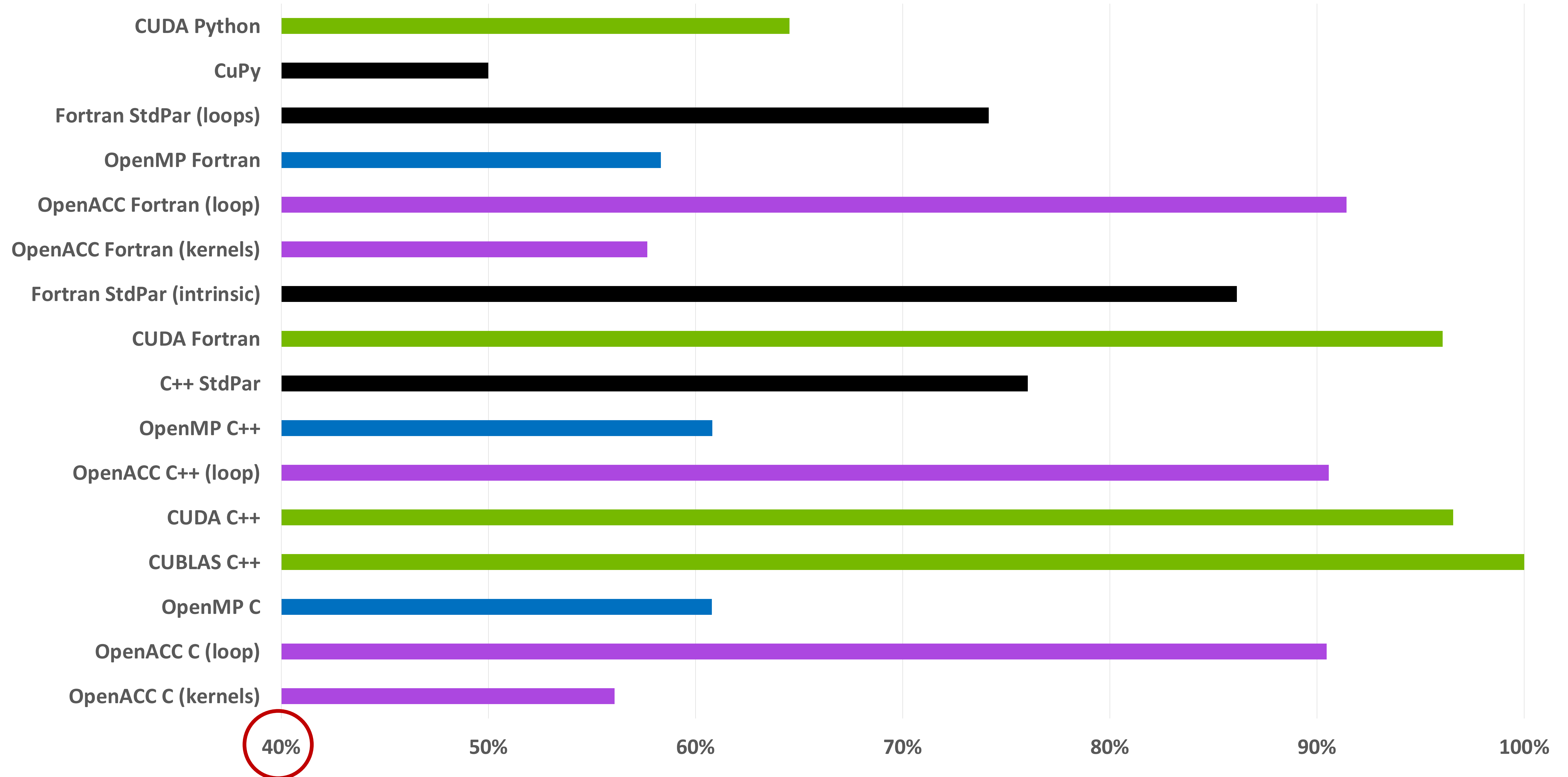
Representative implementations

```
// CUDA C++

__global__
void saxpy(size_t n, T a, T * X, T * Y, T * Z)
{
    auto i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        Z[i] = a * X[i] + Y[i];
    }
}

const int block_size = 256;
dim3 dimBlock(block_size, 1, 1);
dim3 dimGrid(length/block_size, 1, 1);

axpy<<<dimGrid, dimBlock>>>(length, a, X, Y, Z);
```

```
// C++17 standard parallelism

std::transform( std::execution::par_unseq,
                std::begin(X), std::end(X),
                std::begin(Y), std::begin(Z),
                [a](auto&& x, auto&& y) {
                    return a * x + y;
                }
);
```

```
// OpenACC C++

#pragma acc parallel loop
for (size_t i=0; i<length; ++i) {
    Z[i] = a * X[i] + Y[i];
}
```

NVIDIA


Matrix Transpose: B = B + A^T
% of CUBLAS (DGEAM)
CUDA Python
CuPy
Fortran StdPar (loops)
OpenMP Fortran
OpenACC Fortran (loop)
OpenACC Fortran (kernels)
Fortran StdPar (intrinsic)
CUDA Fortran
C++ StdPar
OpenMP C++
OpenACC C++ (loop)
CUDA C++
CUBLAS C++
OpenMP C
OpenACC C (loop)
OpenACC C (kernels)
40%
50%
60%
70%
80%
90%
100%
NVIDIA

Matrix Transpose

Representative implementations

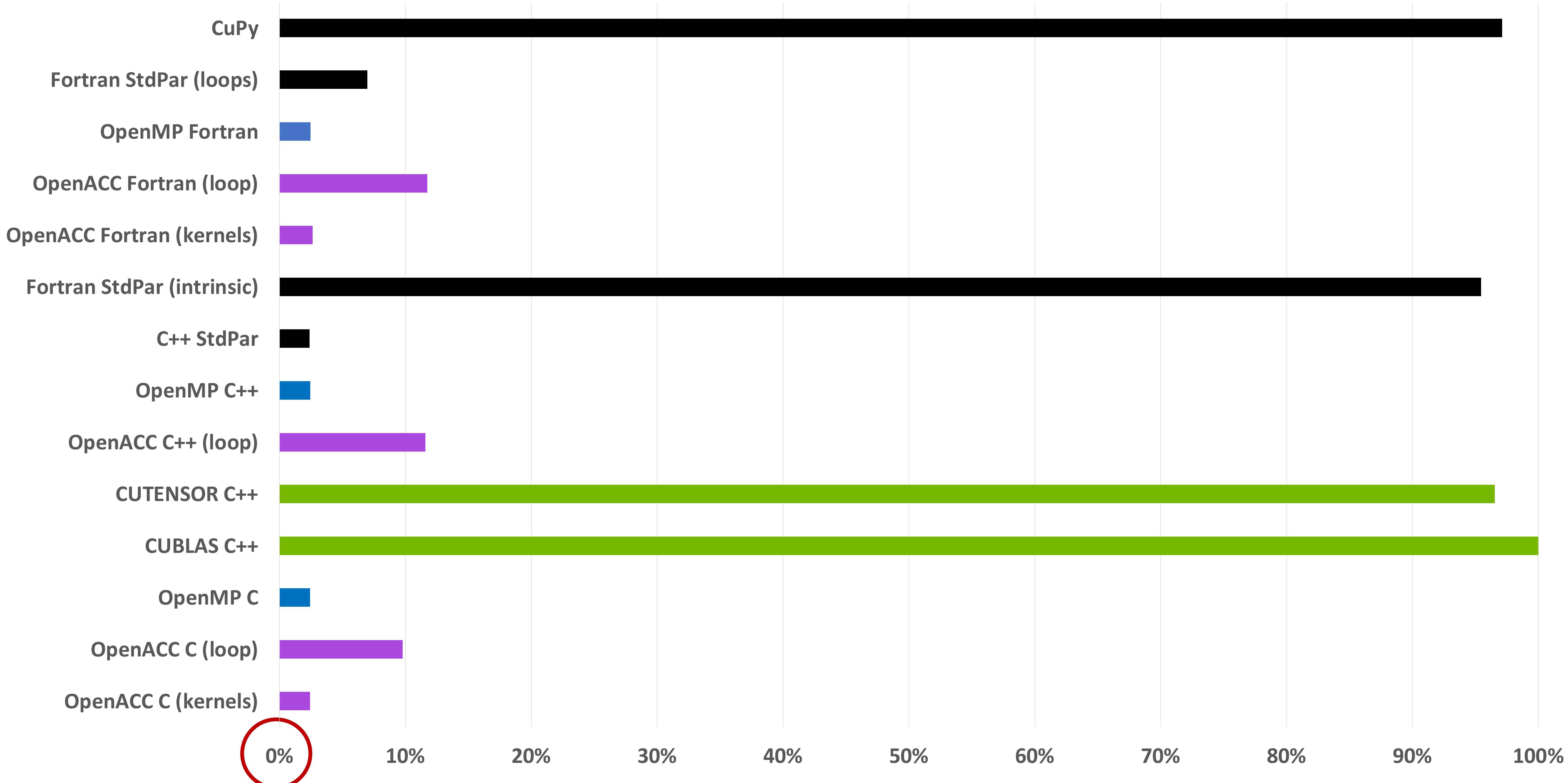
```
! CUDA Fortran

integer(kind=INT32), parameter :: tile_dim = 32
integer(kind=INT32), parameter :: block_rows = 8

attributes(global) subroutine transpose(N, A, B)
  implicit none
  integer(kind=INT32), intent(in), value :: N
  real(kind=REAL64), intent(inout) :: A(N,N)
  real(kind=REAL64), intent(inout) :: B(N,N)
  real(kind=REAL64), shared :: tile(33,32)
  integer :: x, y, j
  x = (blockIdx%x-1) * tile_dim + (threadIdx%x);
  y = (blockIdx%y-1) * tile_dim + (threadIdx%y);
  do j = 0,tile_dim-1,block_rows
    tile(threadIdx%x,threadIdx%y+j) = A(x,y+j);
  end do
  call syncThreads()
  x = (blockIdx%y-1) * tile_dim + (threadIdx%x);
  y = (blockIdx%x-1) * tile_dim + (threadIdx%y);
  do j = 0,tile_dim-1,block_rows
    B(x,y+j) = B(x,y+j) + tile(threadIdx%y+j,threadIdx%x)
  end do
end subroutine transpose
```

```
! Fortran standard parallelism

! Intrinsic version
B = B + transpose(A)

! Loop version
do concurrent (j=1:N, i=1:N)
  B(i,j) = B(i,j) + A(j,i)
enddo
```

```
! OpenACC Fortran

!$acc parallel loop tile(32,32)
do j=1,N
  do i=1,N
    B(i,j) = B(i,j) + A(j,i)
  enddo
enddo
```

NVIDIA.

Matrix Multiplication: C = C + A * B



NVIDIA

Matrix Multiplication

Representative implementations

```
// CUBLAS C/C++

rb = cublasDgemm(handle,
                 CUBLAS_OP_N, CUBLAS_OP_N,
                 N, N, N,
                 &alpha, A, N
                 B, N
                 &one, C, N);
```

```
! Fortran standard parallelism

! Intrinsic version
C = C + matmul(A,B)
```

```
! OpenACC Fortran

!$acc parallel loop tile(32,32)
do j=1,order
  do i=1,order
    do p=1,order
      C(i,j) = C(i,j) + A(i,p) * B(p,j)
    enddo
  enddo
enddo
!$acc end parallel
```

```
! Loop version
do concurrent (j=1:order, i=1:order) local(T)
  T = C(i,j)
  do concurrent (p=1:order) ! Implicit reduction
    T = T + A(i,p) * B(p,j)
   enddo
   C(i,j) = T
enddo
```

NVIDIA.

Quantum Chemistry

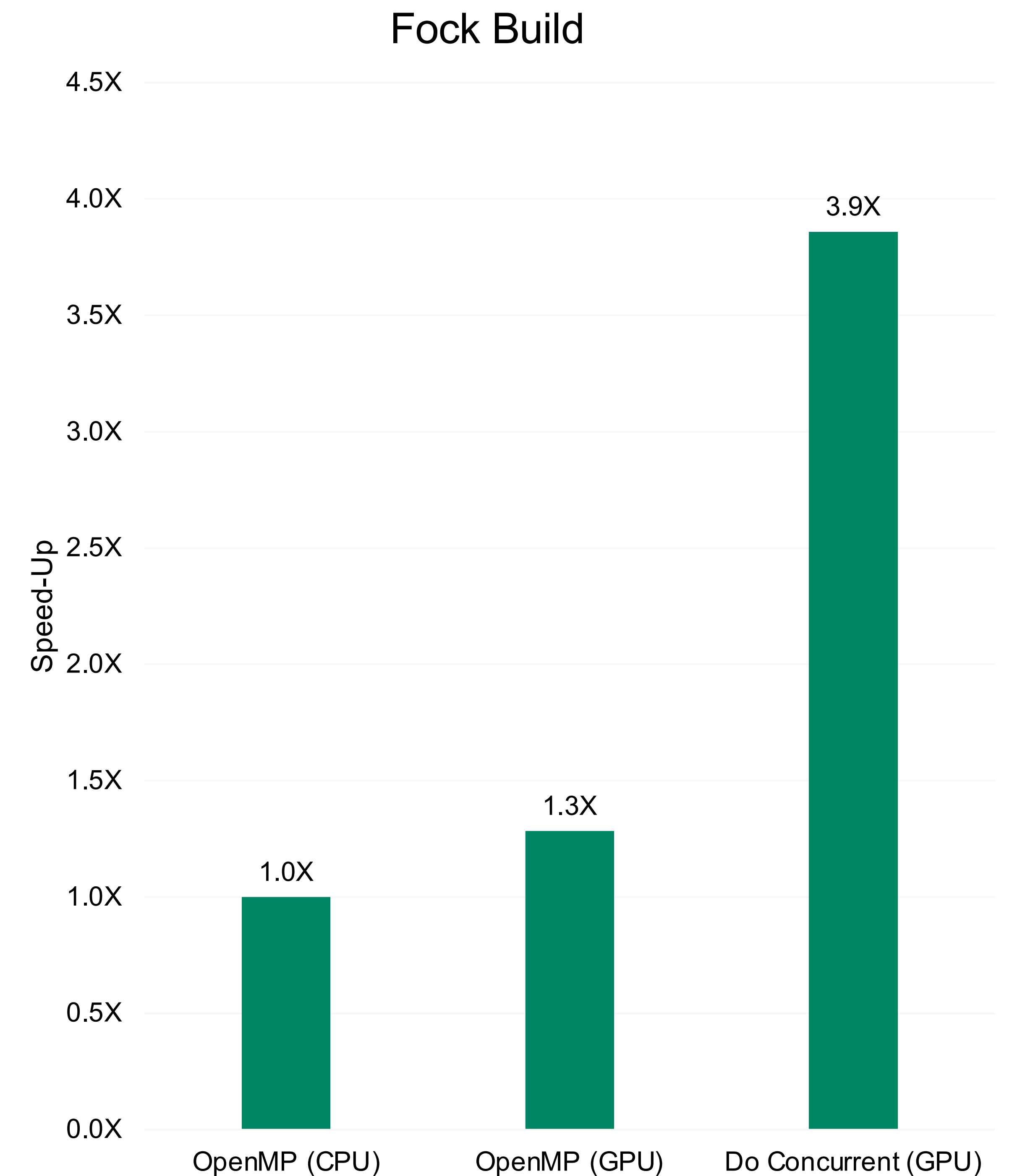
GAMESS

Computational Chemistry with Fortran Do Concurrent

- GAMESS is a popular Quantum Chemistry application.
- More than 40 years of development in Fortran and C
- MPI + OpenMP baseline code
- Hartree-Fock rewritten in Do Concurrent

```
!pre-sorting, screening

!$omp target teams distribute &
             parallel do &
!$omp shared() private()
do iquart = 1, ssdd_quarts
  !recover shell index
  ish=IDX(s_sh)
  jsh=IDX(s_sh)
  ksh=IDX(d_sh)
  lsh=IDX(d_sh)
  !compute ints
  !digest ints
enddo
```

```
!pre-sorting, screening



DO CONCURRENT(iquart=1::ssdd_quarts)&
   SHARED() LOCAL()
  !recover shell index
  ish=IDX(s_sh)
  jsh=IDX(s_sh)
  ksh=IDX(d_sh)
  lsh=IDX(d_sh)
  !compute ints
  !digest ints
enddo
```



nvfortran 22.7, NVIDIA A100 GPU, AMD “Milan” CPU

* Courtesy of Melisa Alkan, Iowa State University. Not yet published.

NVIDIA.

How do we compute CCSD(T) in TCE?

TL;DR lots and lots of tensor contractions

How do we implement CCSD(T)?

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

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

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

Tensor Contraction Strategies

1. Transpose-Transpose-GEMM-Transpose
   - Transposes lead to inefficient and/or unnecessary data movement.
2. **Loops**
   - Impossible to optimize as well as BLAS.
3. Code generators
   - Requires project subcontract for computer scientists forever.
4. **Libraries** (TBLIS, **CUTENSOR**, TAL_SH)
   - Not universally available yet…

```
!!! syntax modified for slide purposes !!!

  integer, intent(in) :: h3d,h2d,h1d
  integer, intent(in) :: p6d,p5d,p4d,p7d
  real, intent(inout) :: T3(h2d,h3d,h1d,p4d,p6d,p5d)
  real, intent(in) :: T2(p7d,p4d,h1d,h2d)
  real, intent(in) :: V2(p7d,h3d,p6d,p5d)
  integer :: h3,h2,h1,p6,p5,p4,p7
  !!! DIRECTIVES GO HERE, IF APPLICABLE
  do p5=1,p5d
   do p6=1,p6d
    do p4=1,p4d
     do h1=1,h1d
      do h3=1,h3d
       do h2=1,h2d
        do p7=1,p7d
         T3(h2,h3,h1,p4,p6,p5) += T2(p7,p4,h1,h2) * V2(p7,h3,p6,p5)
        enddo
  enddo ; enddo ; enddo ; enddo ; enddo ; enddo
end
```

NVIDIA.

```
!!! syntax modified for slide purposes !!!

  integer :: h3d,h2d,h1d,p6d,p5d,p4d,p7d
  integer :: h3,h2,h1,p6,p5,p4,p7
  double precision, intent(inout) :: T3(h2d,h3d,h1d,p4d,p6d,p5d)
  double precision, intent(in) :: V2(p7d,h3d,p6d,p5d)
  double precision, intent(in) :: T2(p7d,p4d,h1d,h2d)
  double precision :: X2(p7d,h2d,h1d,p4d)
  ! transposing inputs improves memory access, hence performance
  do concurrent (h2=1:h2d, h1=1:h1d, p4=1:p4d, p7=1:p7d)
    X2(p7,h2,h1,p4) = T2(p7,p4,h1,h2)
  enddo
  do concurrent (p5=1:p5d, p6=1:p6d, p4=1:p4d, h1=1:h1d, h3=1:h3d, h2=1:h2d)
    do p7=1,p7d ! no reduction support...yet
      T3(h2,h3,h1,p4,p6,p5) += X2(p7,h2,h1,p4) * V2(p7,h3,p6,p5)
    enddo
  enddo
end
```

NVIDIA.

!!! syntax modified for slide purposes !!!

```
integer :: h3d,h2d,h1d,p6d,p5d,p4d,p7d
integer :: h3,h2,h1,p6,p5,p4,p7
double precision, intent(inout) :: T3(h2d,h3d,h1d,p4d,p6d,p5d)
double precision, intent(in) :: V2(p7d,h3d,p6d,p5d)
double precision, intent(in) :: T2(p7d,p4d,h1d,h2d)
double precision :: X2(p7d,h2d,h1d,p4d)
! transposing inputs improves memory access, hence performance
X2 = reshape(t2sub, [p7d,h2d,h1d,p4d], order=[1,4,3,2])
do concurrent (p5=1:p5d, p6=1:p6d, p4=1:p4d, h1=1:h1d, h3=1:h3d, h2=1:h2d)
  do p7=1,p7d
    T3(h2,h3,h1,p4,p6,p5) += X2(p7,h2,h1,p4) * V2(p7,h3,p6,p5)
  enddo
enddo
```

NVIDIA.

```c
// CUTENSOR from C/C++
// (the Fortran API also exists)

// d2_9 = [2][8]
{
    int k = 8;
    int32_t mT3[6]={h2,h3,h1,p4,p6,p5};
    int32_t mT2[4]={p7,p4,h1,h2};
    int32_t mV2[4]={p7,h3,p6,p5};
    alpha[2][k] = 1;
    active[2][k] = true;
    cutensorInitContractionDescriptor(
          &h,
          &dX[2][k], &dT2, mT2, aT2,
          &dV2, mV2, aV2,
          &dT3, mT3, aT3,
          &dT3, mT3, aT3,
          CUTENSOR_R_MIN_64F);
}
```

```fortran
subroutine ref_sd_t_d2_9(h3d,h2d,h1d,
                         p6d,p5d,p4d,p7d,
                         T3,T2,V2)
  integer, intent(in) :: h3d,h2d,h1d
  integer, intent(in) :: p6d,p5d,p4d,p7d
  real, intent(inout) :: T3(h2d,h3d,h1d,p4d,p6d,p5d)
  real, intent(in) :: T2(p7d,p4d,h1d,h2d)
  real, intent(in) :: V2(p7d,h3d,p6d,p5d)
  integer :: h3,h2,h1,p6,p5,p4,p7
  do p5=1,p5d
   do p6=1,p6d
    do p4=1,p4d
     do h1=1,h1d
      do h3=1,h3d
       do h2=1,h2d
        do p7=1,p7d
         T3(h2,h3,h1,p4,p6,p5) += T2(p7,p4,h1,h2)
                                * V2(p7,h3,p6,p5)
        enddo
       enddo
      enddo
     enddo
    enddo
   enddo
  enddo
end
```

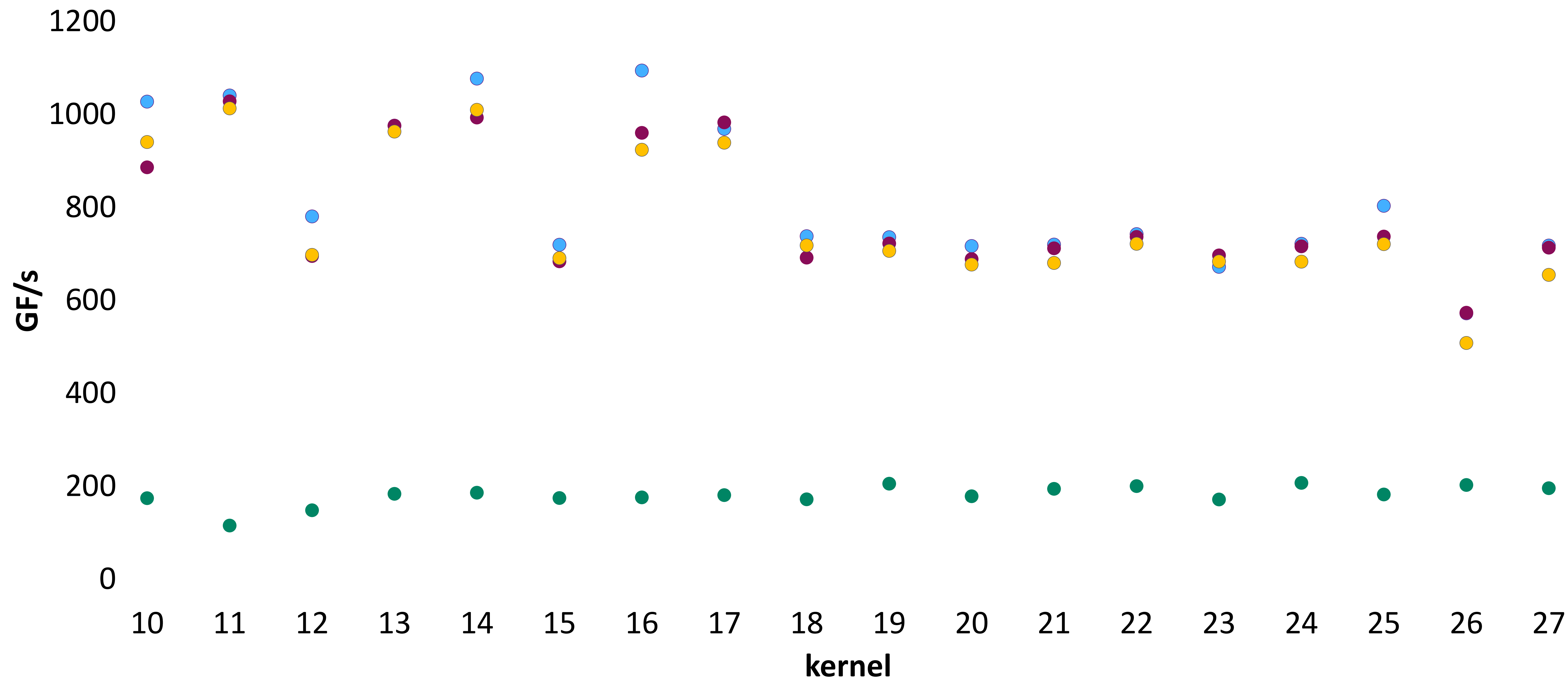
https://github.com/jeffhammond/nwchem-tce-triples-kernels/blob/master/src/cutensor_driver.c

Experiments

- A100 DGX Station
  - CPU: AMD EPYC 7742 (64c)
  - GPU: NVIDIA A100 SXM 80GB
- Compilers and Math Libraries
  - NVHPC 21.7 compilers and OpenBLAS
  - CUTENSOR 1.3.2 (in NVHPC 21.9)
- NTTK (standalone driver for NWChem TCE CCSD(T) kernels)
  - https://github.com/jeffhammond/nwchem-tce-triples-kernels
  - tilesize = 30 keeps the memory footprint under 6 GB
    - Contracted dimension of 30 cannot hit peak flop/s

NVIDIA.

NWChem TCE CCSD(T) kernels

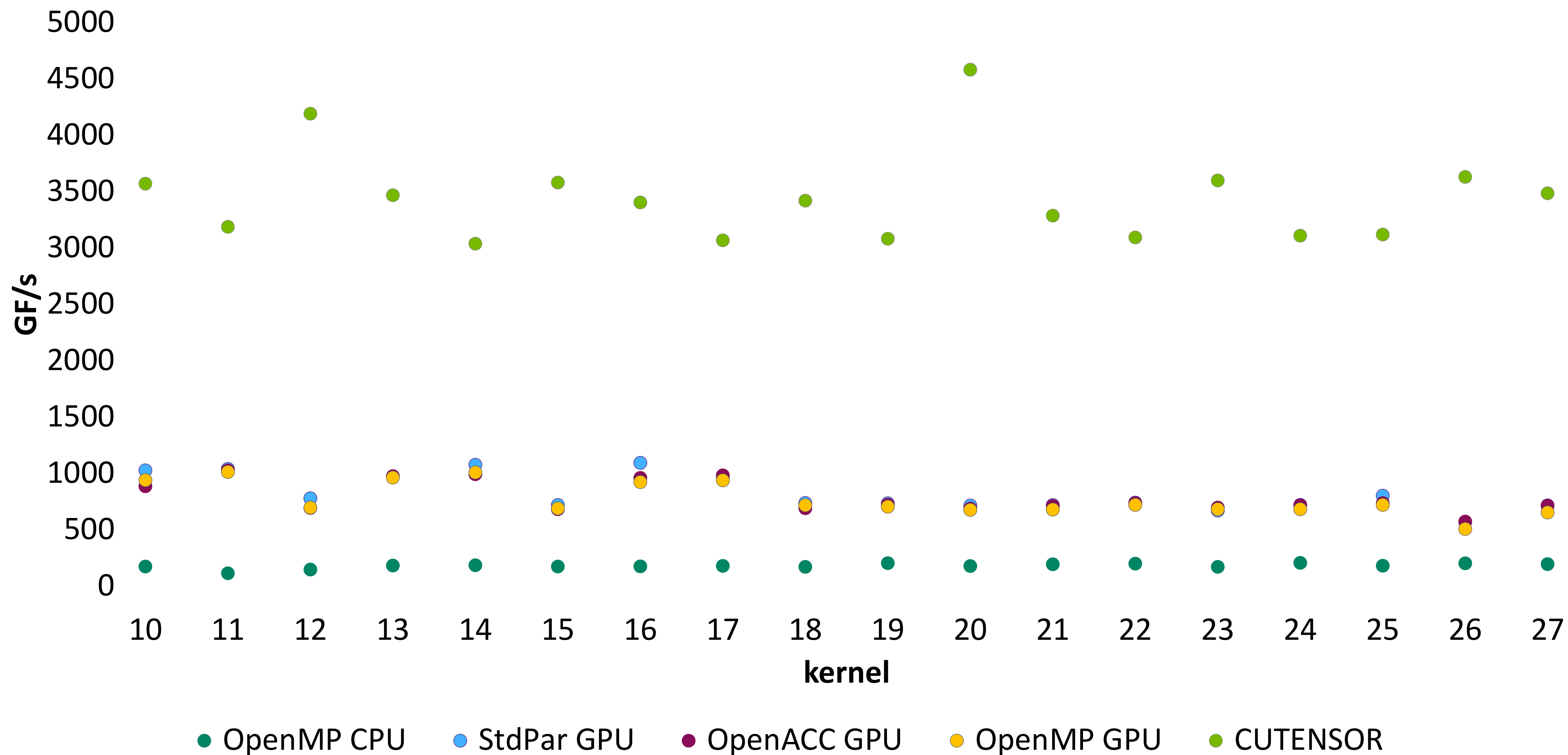


● OpenMP CPU ● StdPar GPU ● OpenACC GPU ● OpenMP GPU

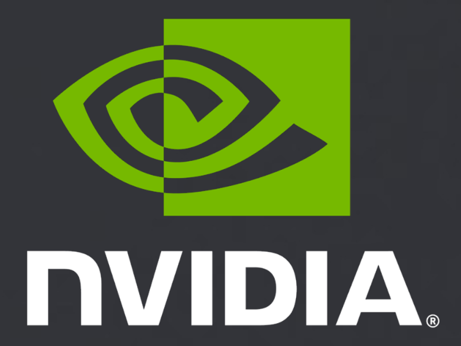
https://github.com/jeffhammond/nwchem-tce-triples-kernels/

NVIDIA

NWChem TCE CCSD(T) kernels



https://github.com/jeffhammond/nwchem-tce-triples-kernels/


NVIDIA