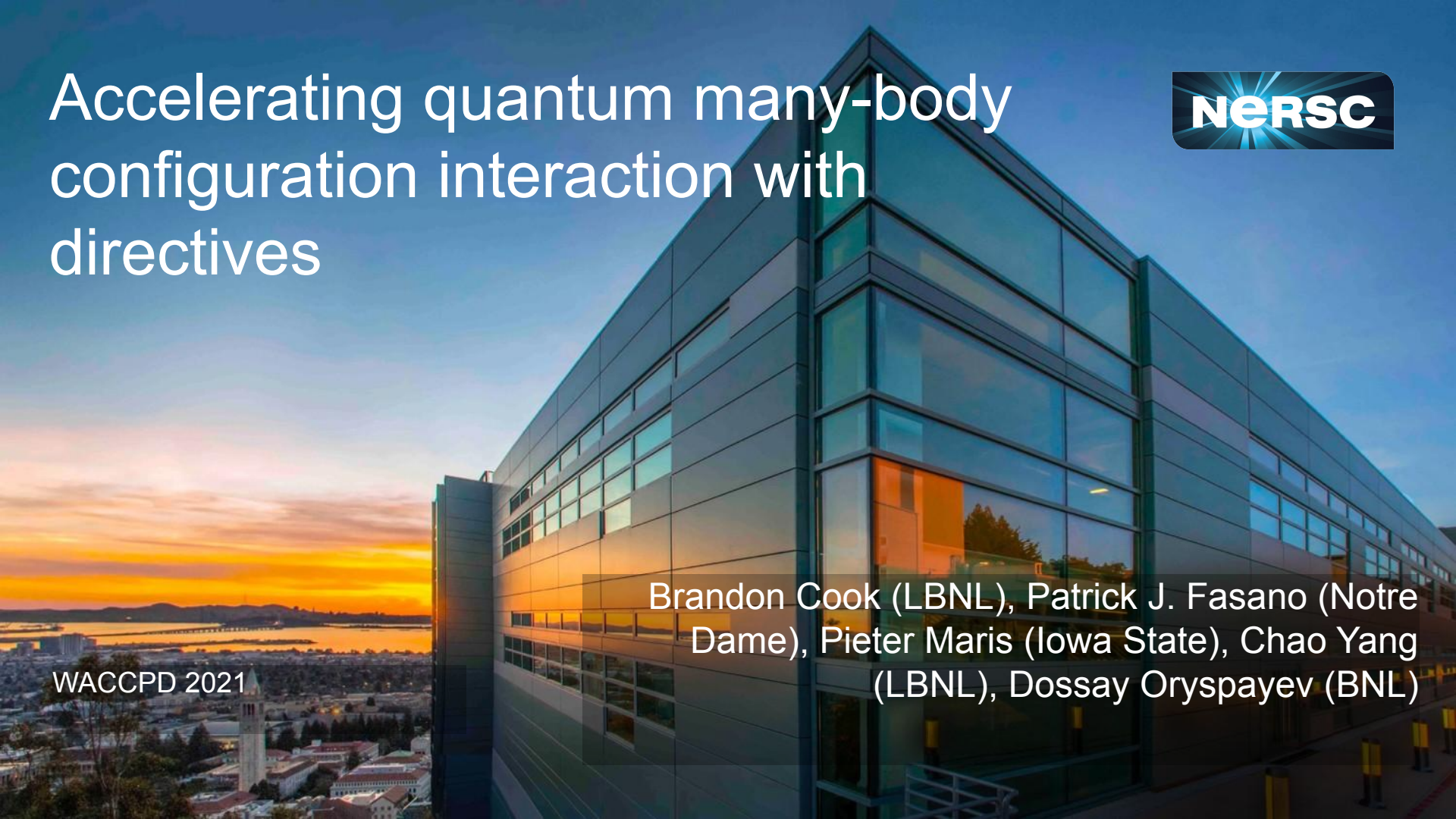
Accelerating quantum many-body configuration interaction with directives



Brandon Cook (LBNL), Patrick J. Fasano (Notre Dame), Pieter Maris (Iowa State), Chao Yang (LBNL), Dossay Oryspayev (BNL)

WACCPD 2021

In this talk

- Introduction to Many Fermion Dynamics (nuclear) Configuration Interaction code
- NESAP, Perlmutter and goals for MFDn GPU porting effort
- Target platforms
- GPU acceleration of key kernels using directives
  - Hierarchical counting nonzero matrix tiles and elements
  - Conversion of counts to offsets
  - Computing and storing nonzero matrix elements
  - Calculation of physical observables (array reductions)

all code available at https://gitlab.com/NERSC/nersc-proxies/mfdn-kernels

Many Fermion Dynamics - nuclear (MFDn)

- Configuration Interaction (CI) for nuclear structure
  - Realistic nucleon-nucleon and three-nucleon forces
- Fortran 90
  - platform independent
  - hybrid MPI + OpenMP
- Production application with 10+ years of development
  - historically targeting multicore CPU platforms such as Jaguar (OLCF), Mira, Theta KNL (ALCF), Edison, Cori KNL (NERSC)
- Currently in use at multiple DOE centers
  - *add* support for GPUs

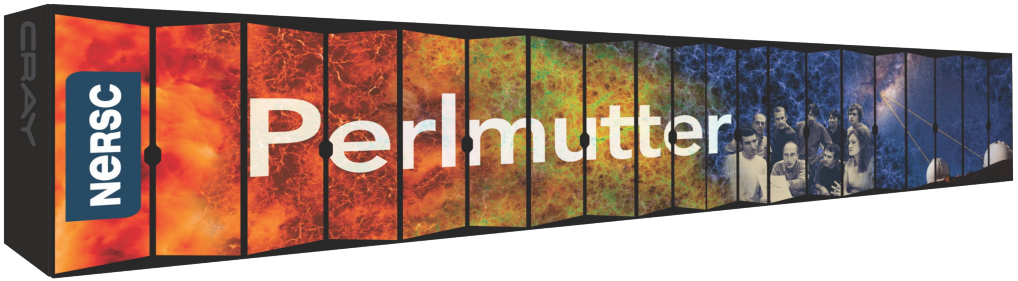
Optimization constraints

- Enable efficient use of GPUs
- Retain portability
  - multiple vendors of GPUs
  - continue support for CPUs
- Productivity
  - Total rewrite in e.g. C++ not feasible
  - Avoid code duplication (as much as possible) or other changes that impact maintainability
- Efficient use of memory
  - Science drivers are to simulate largest problems possible
  - Optimizations must not increase memory footprint

Target platforms

System	Location	CPU	GPU
Cori	NERSC	Intel KNL	none
Theta	ALCF	Intel KNL	none
Perlmutter	**NERSC**	**AMD Milan**	**NVIDIA A100**
Frontier*	OLCF	AMD	AMD
Aurora*	ALCF	Intel	Intel
NERSC-10*	NERSC	?	?

* = long term goals



- Phase 1
  - 1,536 nodes with 1 AMD “Milan” CPU + 4 NVIDIA A100 GPUs
  - 256 GB CPU + 160 GB GPU memory per node
- NESAP application readiness program
  - OpenACC was selected ~2 years ago at start of NESAP as OpenMP support for this GPU was not mature

Test platforms

System	Location	CPU	GPU
Cori GPU	NERSC	Intel Skylake	NVIDIA V100
Cori DGX	NERSC	AMD Rome	NVIDIA A100
Spock	OLCF	AMD Rome	AMD MI100

System	Compiler	_OPENACC	_OPENMP
Cori GPU	NVIDIA HPCSDK 21.7	201711 (2.6)	202011 (5.1)
Cori DGX	NVIDIA HPCSDK 21.7	201711 (2.6)	202011 (5.1)
Spock	HPE CCE 12.0.1	201306 (2.0)	201511 (4.5)

MFDn structure

1. Determine sparsity
   a. number and location of nonzero matrix tiles
   b. number and location of nonzero elements in tiles
2. Calculate matrix elements
3. Compute N lowest eigenvalue/ eigenvector pairs [1]
4. Calculate physical observables from eigenvectors

[1] P. Maris et al. Accelerating an Iterative Eigensolver for Nuclear Structure Configuration Interaction Calculations on GPUs using OpenACC (arXiv:2109.00485)

Many-body state representations

- Many-body basis states are composed of antisymmetrized products of Single Particle (SP) states
- Many-body states can be represented in two ways
  - BIN(ϕ_i) = ...0010010000...0001001..
    - each bit corresponds to an SP state which is either occupied or not
    - for n nucleons n bits are set
    - memory proportional to number of SP states
  - $\phi_i = \{s_1, s_2, ...s_n\}$
    - set of integers storing which SP states are occupied
    - positive-definite and ordered
    - memory proportional to number of nucleons

Sparsity determination

Two many-body states (with two-body forces) interact and the matrix element is nonzero only if 0, 2, or 4 single particle states are differently-occupied.

- bit representation only
- int representation only
- truncated bit representation + int representation

Hybrid bit + integer set representation

```
!$acc parallel loop
do i = 1, n
   c = 0
   !$acc loop reduction(+:c)
   do j = 1, n
      d = popcnt(ieor(bitrep1(i), bitrep2(j)))
      if (d > 4) cycle
      d = count_difference(mbstate1(:,i), np, mbstate2(:,j), np)
      if (d <= 4) c = c + 1
   end do
   counts(i) = c
end do
numnnz = sum(counts)
```

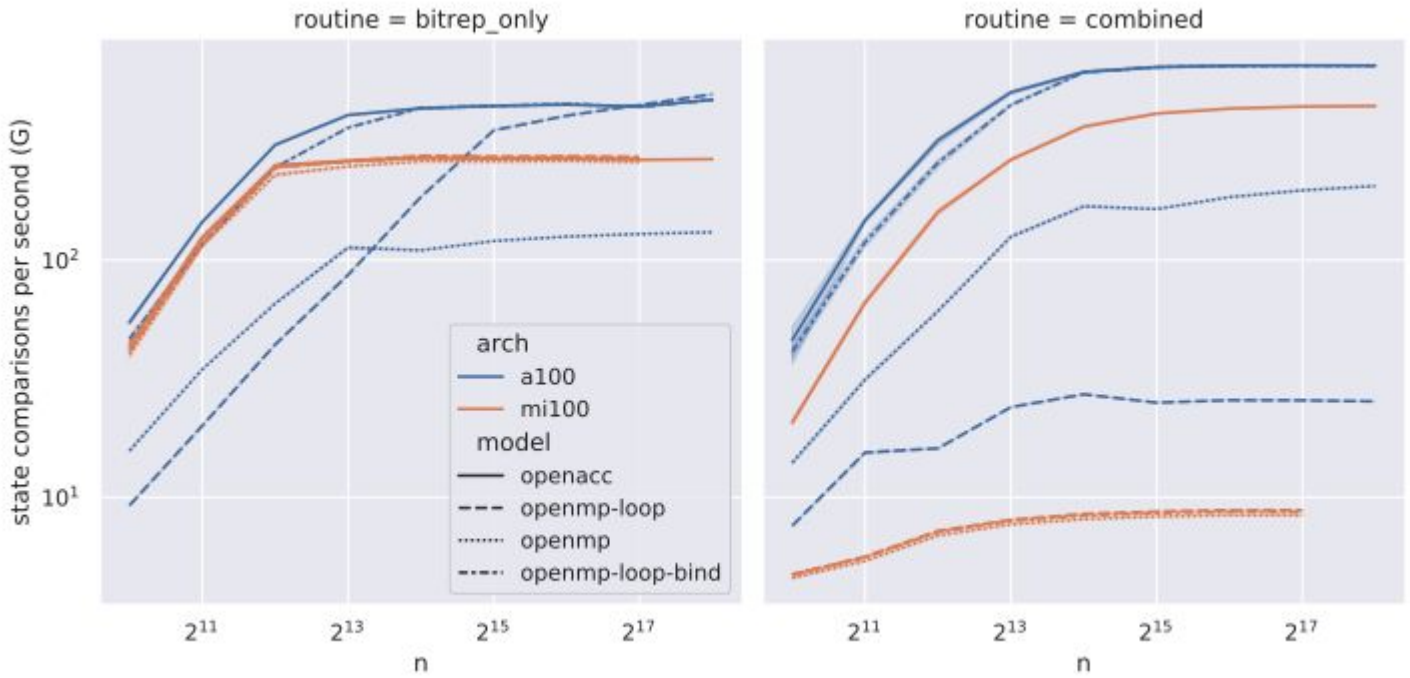
Level 1 directives

```
!$acc parallel loop
!$omp target teams distribute private(d)
!$omp target teams loop private(d)
!$omp target teams loop bind(teams) private(d)
```

Level 2 directives

```
!$acc loop reduction(+:c)
!$omp parallel do reduction(+:c) private(d)
!$omp loop reduction(+:c) private(d)
!$omp loop bind(parallel) reduction(+:c) private(d)
```

bit rep and hybrid non-zero counting performance



Takeaways

- Check the compiler diagnostic output!
  - Even with “simple” loops
- Function/ subroutine calls in parallel loops should receive extra attention
  - You could end up running serial code on the GPU
- OpenACC loop achieved best performance
  - OpenMP loop can be competitive with bind hints if the compiler support is available
- OpenMP target teams distribute parallel do potentially involves overhead compared to OpenMP loop

Prefix sum / scan

$$y_{i+1} = \sum_{j=0}^{i} x_j = y_i + x_i$$

- Common primitive in many algorithms
- OpenMP spec includes a scan clause for reductions
  - but no compiler supports it for offload!
- Available in C++ for specific platforms through many means, e.g. Kokkos, libc++, Thrust, CUB
  - mixing languages not acceptable for maintainability and portability

Prefix sum in MFDn

- In MFDn needed to convert counts to offsets
  - key transformation needed to use a single large shared array vs many smaller private arrays
- Since the offsets can be computed once and reused we just need to avoid a data transfer
  - `!$acc serial` may be enough for some small problems

Filling shared arrays

```
!$acc parallel loop
do i = 1, n
   indx(i) = offset(i)
   !$acc loop device_type(host) seq
   do j = 1, m
      if (mod(j,p) == 0) then
         !$acc atomic capture
         indx(i) = indx(i) + 1
         k = indx(i)
         !$acc end atomic
         arr(k) = j
      end if
   end do
end do
```

Two levels of parallelism

outer level

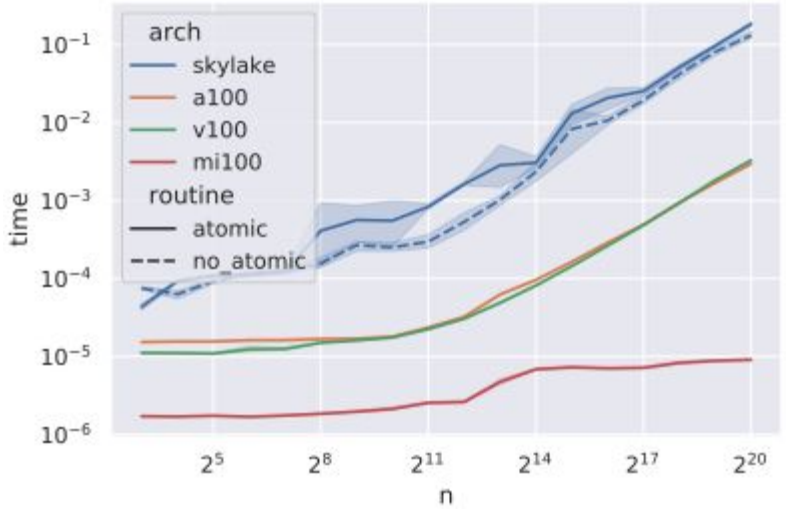
- enough work for CPUs
- no data conflicts

inner level

- order does not matter for correctness
- serial on CPUs for efficient cache use
- parallel on GPUs for parallelism
  - data conflicts -> use of atomics

Architectural specialization

- Even with no conflicts, use of atomics on CPUs introduces overhead
- OpenMP or OpenACC only without additional preprocessor?
  - `!$acc device_type`
  - `!$omp metadirective`



Architectural specialization: OpenACC

```
!$acc parallel loop
do i = 1, n
   indx(i) = offset(i)
   !$acc loop device_type(host) seq
   do j = 1, m
      if (mod(j,p) == 0) then
         !$acc atomic capture
         indx(i) = indx(i) + 1
         k = indx(i)
         !$acc end atomic
         arr(k) = j
      end if
   end do
end do
```

clauses after `device_type(<type>)` only apply to devices of `<type>`

but,

not available for `!$acc atomic`

Architectural specialization: OpenMP

```
!$omp metadirective when(target_device={kind(gpu)}: target teams distribute) &
!$omp& default(parallel do private(k))
do i = 1, n
   indx(i) = offset(i)
   !$omp metadirective when(device={kind(gpu)}: parallel do private(k))
   do j = 1, m
      if (mod(j,p) == 0) then
         !$omp begin metadirective when(device={kind(gpu)}: atomic capture)
         indx(i) = indx(i) + 1
         k = indx(i)
         !$omp end metadirective
         arr(k) = j
      end if
   end do
end do
```

possible according to the specification, but no compiler support yet

Filling shared arrays - takeaways

- multi-architecture code with directives not currently possible without some of:
  - preprocessor
  - runtime API calls
  - code duplication
- `!$omp metadirective` is a promising solution, but compiler support not yet available
- Future versions of OpenACC may also improve in this area

Array reductions

$$a_k = \sum_{ij} x_i \, (O_k)_{ij} \, y_j$$

- In MFDn requirement comes from computing expectation value of operators corresponding to physically observable quantities
- Support has been in specifications for some time
  - but implementation in compilers has lagged (difficult to implement generically with good performance)
- Initial support for OpenMP available in NVIDIA and HPE compilers

3 implementations

$$a_k = \sum_{ij} x_i \, (O_k)_{ij} \, y_j$$

array reduction

```
!$acc parallel loop collapse(2)
reduction(+:a)
do i = 1, n
   do j = 1, n
      do k = 1, m
         a(k) = a(k) + x(k,i) * y(k,j)
      end do
   end do
end do
```

atomic operations

```
!$acc parallel loop collapse(3)
do i = 1, n
   do j = 1, n
      do k = 1, m
         !$acc atomic
         a(k) = a(k) + x(k,i) * y(k,j)
         !$acc end atomic
      end do
   end do
end do
```

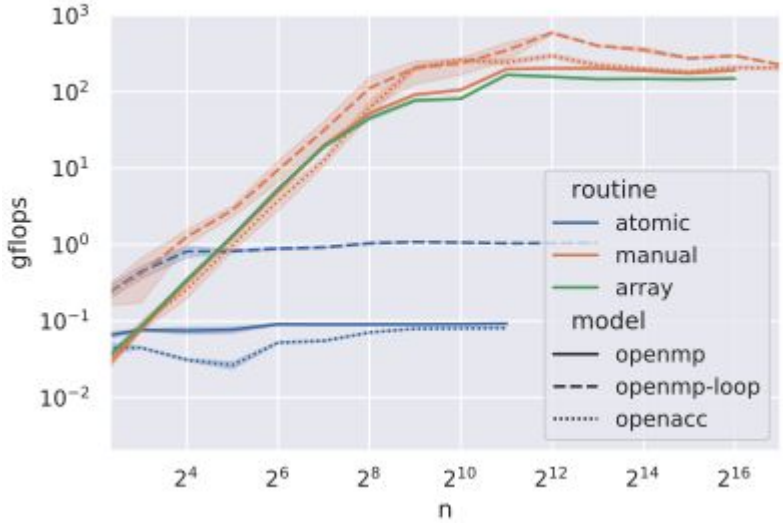
code generation via fypp templates for each array size required

```
#:def CSV(x,n)
${",".join(f"{x}{i}" for i in range(1, n+1))}$
#:enddef CSV

#:for num_elements in range(2, max_elements+1)
  subroutine reduction${num_elements}$(x, y, a, n, dt)
    integer, parameter :: m = ${num_elements}$
    integer, intent(in) :: n
    real(sp), dimension(m, n), intent(in) :: x, y
    real(sp), intent(out) :: a(m)
    integer :: i,j
    real(dp) :: t0
    real(dp), intent(out) :: dt
#:for i in range(1, num_elements+1)
    real(sp) :: a${i}$
#:endfor
    !$acc data present(x,y)
    t0 = wtime()
#:for i in range(1, num_elements+1)
    a${i}$ = a(${i}$)
#:endfor
    !$acc parallel loop collapse(2) &
    !$acc reduction(+:${CSV("a",num_elements}$)
    do i = 1, n
       do j = 1, n
#:for i in range(1, num_elements+1)
          a${i}$ = a${i}$ + x(${i}$,i) * y(${i}$,j)
#:endfor
       end do
    end do
    !$acc end parallel
#:for i in range(1, num_elements+1)
    a(${i}$) = a${i}$
#:endfor
    dt = wtime() - t0
    !$acc end data
  end subroutine reduction${num_elements}$
#:endfor
```

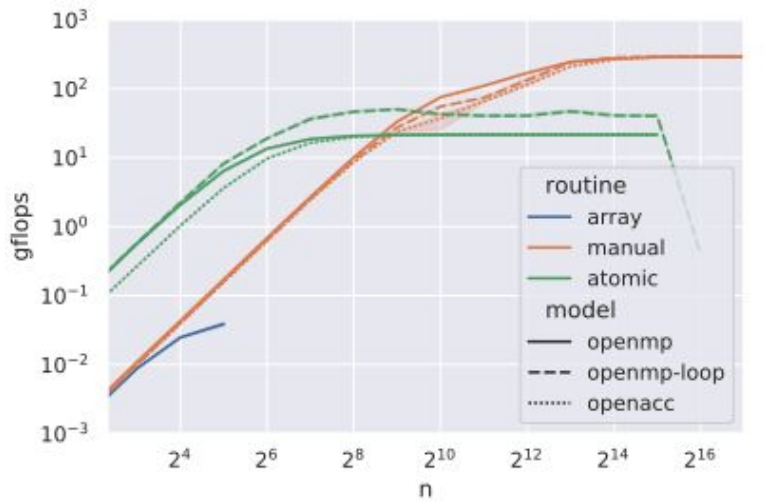
```
subroutine reduction3(x, y, a, n, dt)
  integer, parameter :: m = 3
  integer, intent(in) :: n
  real(sp), dimension(m, n), intent(in) :: x, y
  real(sp), intent(out) :: a(m)
  integer :: i,j
  real(dp) :: t0
  real(dp), intent(out) :: dt
  real(sp) :: a1
  real(sp) :: a2
  real(sp) :: a3
  !$acc data present(x,y)
  t0 = wtime()
  a1 = a(1)
  a2 = a(2)
  a3 = a(3)
  !$acc parallel loop collapse(2) &
  !$acc reduction(+:a1,a2,a3)
  do i = 1, n
     do j = 1, n
        a1 = a1 + x(1,i) * y(1,j)
        a2 = a2 + x(2,i) * y(2,j)
        a3 = a3 + x(3,i) * y(3,j)
     end do
  end do
  !$acc end parallel
  a(1) = a1
  a(2) = a2
  a(3) = a3
  dt = wtime() - t0
  !$acc end data
end subroutine reduction3
```

Performance on CPU (Skylake, nvfortran)



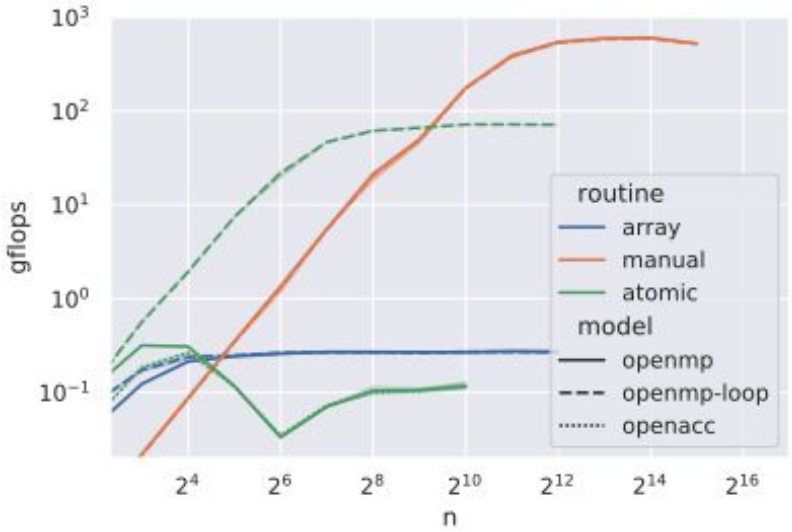
- array size 64
- atomics with many conflicts are *very* slow
- array reduction with OpenMP works well
- manual generation with OpenMP loop best

Performance on GPU (NVIDIA A100, nvfortran)



- array size 64
- atomics with many conflicts are best
  - in some cases
- array reduction with OpenMP works well
  - until you hit a bug with some array sizes
- manual generation with any model highest peak performance

Performance on GPU (AMD MI100, HPE CCE)



- array size 64
- atomics are best in some ranges
  - but only with OpenMP-loop
- array reduction with OpenMP works
  - but slow, no parallel code
- manual generation with OpenMP and OpenMP-loop works
  - OpenACC crashes the compiler

Array reductions takeaways

- atomics are best in some ranges on GPUs
  - with OpenMP and sensitive to directive choice
- array reduction with OpenMP “works”
  - but performance is not portable
- manual generation via templates with OpenMP-loop overall best solution
  - but! adding a templating engine to your build may not be a good idea: long compile times, maintenance, additional dependency

Conclusions

- MFDn is now enabled for CPUs *and* GPUs
- multiple thread private arrays -> 1 big shared array with offsets
  - Counts -> Offsets -> Fill
- For the kernels examined in this work OpenACC provides the best performance
- OpenMP *with support for latest features by compilers* is promising candidate for single source performance portability

Acknowledgements

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

This work is supported by the U.S. Department of Energy (DOE) under Award Nos. DE-FG02-95ER40934 and DE-SC0018223 (SciDAC/NUCLEI), and by the DOE Office of Science, Office of Workforce Development for Teachers and Scientists, Office of Science Graduate Student Research (SCGSR) program (administered by the Oak Ridge Institute for Science and Education (ORISE), managed by ORAU under contract number DE-SC0014664).