OAK RIDGE National Laboratory

NuCCOR on Crusher

Justin Gage Lietz, Associate Research Staff

Oak Ridge National Laboratory

Algorithms and Performance Analysis Group

December 9, 2022

ORNL is managed by UT-Battelle LLC for the US Department of Energy

U.S. DEPARTMENT OF ENERGY

NUCCOR Overview

- Nuclear Coupled Cluster Oak Ridge
- Goal: Calculate properties of atomic nuclei from “first principles”
- How: Approximately solve the Schrödinger using the coupled cluster method
- What: Mix of old and new Fortran with small pieces of CUDA/HIP.

NuCCOR Overview

- NuCCOR is one of eight Center for Advanced Application Readiness (CAAR) projects selected to prepare the app prior to Frontier.
- CAAR began at the end of 2019, so much of the porting for NuCCOR happened pre-Crusher.
- Porting in two parts:
  - HIPification of CUDA code
  - Development of the NTCL (Nuclear Tensor Contraction Library)

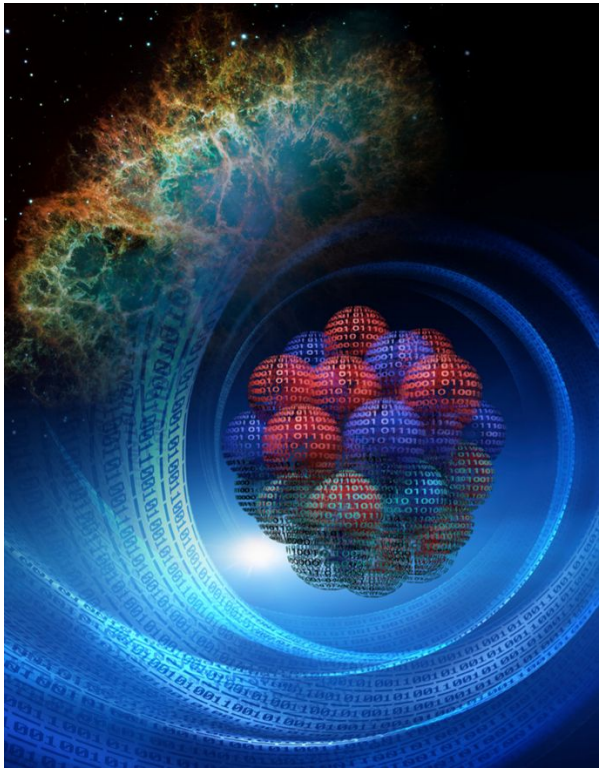
OAK RIDGE National Laboratory

Part I: Scientific Motivations

U.S. DEPARTMENT OF ENERGY

Nuclear Structure

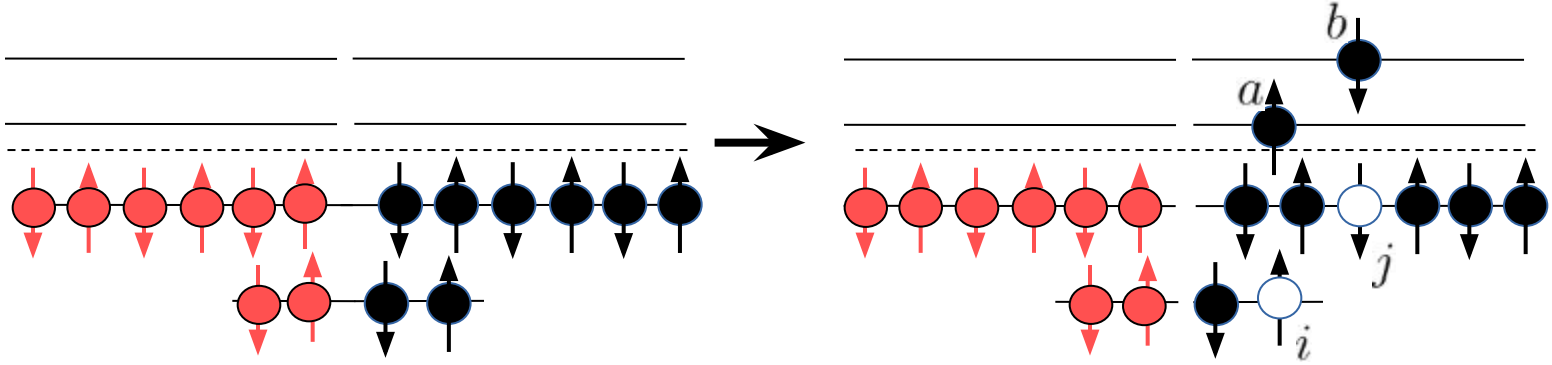
- Input: Particles and Interactions
- Output: Properties of nuclei (binding energies, radii,decay rates)
- Uses:
  - Fundamental symmetries (neutrinoless double beta decay)
  - Supernovae
  - Neutron stars



Coupled Cluster Tensors

$$\hat{T}_2 = \frac{1}{4} \sum_{abij} \langle ab|t_2|ij\rangle \; N[a_a^\dagger a_b^\dagger a_j a_i]$$

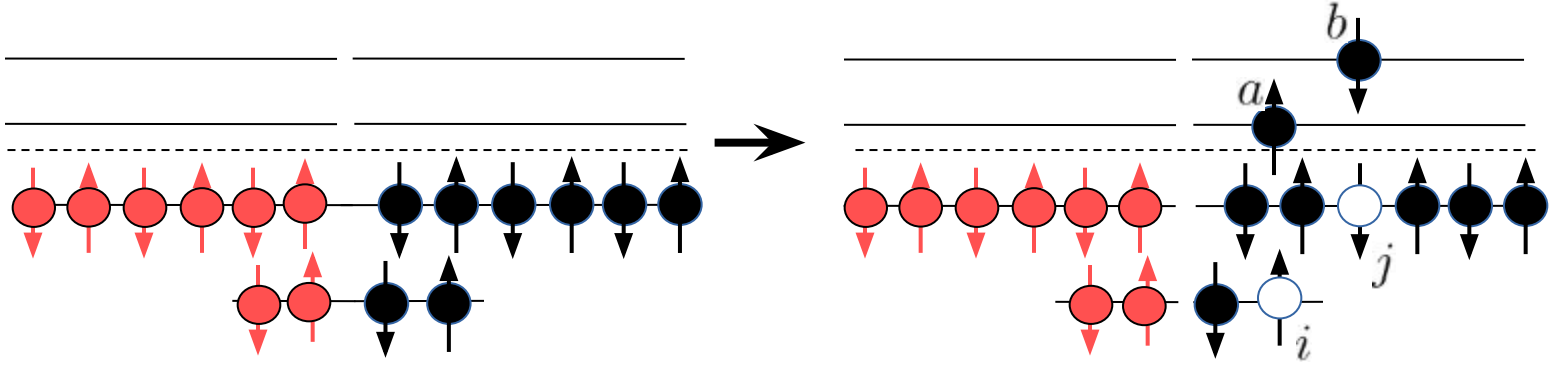
$$|\Phi_0\rangle \rightarrow \hat{T}_2 \, |\Phi_0\rangle$$



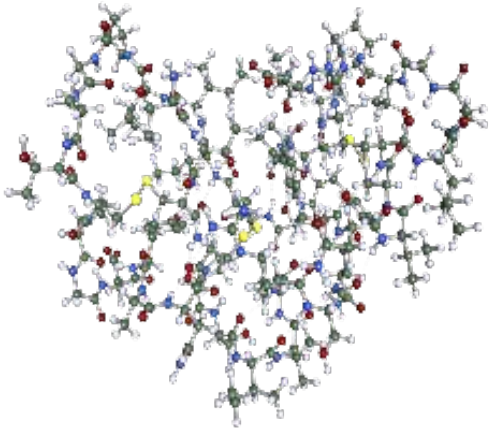
Coupled Cluster Tensors

$$\hat{T}_2 = \frac{1}{4} \sum_{abij} \langle ab|t_2|ij\rangle \; N[a_a^\dagger a_b^\dagger a_j a_i]$$

$$|\Phi_0\rangle \rightarrow \hat{T}_2 \, |\Phi_0\rangle$$

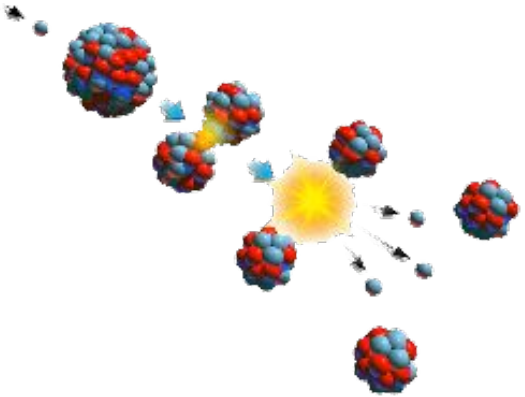


Tensors arise in multiple domains



Quantum Chemistry

Astrophysics



Nuclear Physics

What is a tensor?

It is different things to different people!
To someone researching gravity:

$$G_{\mu\nu} + \Lambda g_{\mu\nu} = \frac{8\pi G}{c^4} T_{\mu\nu}$$

To someone researching quantum computing:

$$|\Psi\rangle_{AB} = |\psi\rangle_A \otimes |\phi\rangle_B$$

To someone researching machine learning or data science:

$$T = \text{array}([[1,2,3],[4,5,6],[7,8,9]],\\ [[10,11,12],[13,14,15],[16,17,18]])$$

These will all simply be called “tensors”.

Tensor Contractions: How we manipulate the data

$$\langle ab|t_2|ij\rangle + = \frac{1}{4}\sum_{klcd} \langle kl|v|cd\rangle \langle cd|t_2|ij\rangle \langle ab|t_2|kl\rangle$$

Indices → Basis states (size n)
Tensors with 4 indices → Memory scales as O(n^4)
Looping over 8 indices → Computational scales as O(n^8)

Polynomial Scaling is Still Big!

$$\langle ab|t_2|ij\rangle += \frac{1}{4}\sum_{klcd}\langle kl|v|cd\rangle\langle cd|t_2|ij\rangle\langle ab|t_2|kl\rangle$$

$$\langle abc|t|ijk\rangle += \frac{1}{8}\sum_{efg}\langle abc|v|efg\rangle\langle efg|t|ijk\rangle$$

Some target systems on Frontier will use n~10^5 basis states.
Order 4 tensors: n^4~10^20 ~ 100 Exabytes FP64
Order 6 tensors: n^6~10^30 ~ 10^12 Exabytes FP64

OAK RIDGE National Laboratory

Part II: Optimizing Tensor Contractions In Coupled Cluster

ORNL is managed by UT-Battelle LLC for the US Department of Energy

U.S. DEPARTMENT OF ENERGY

Overcoming Challenges

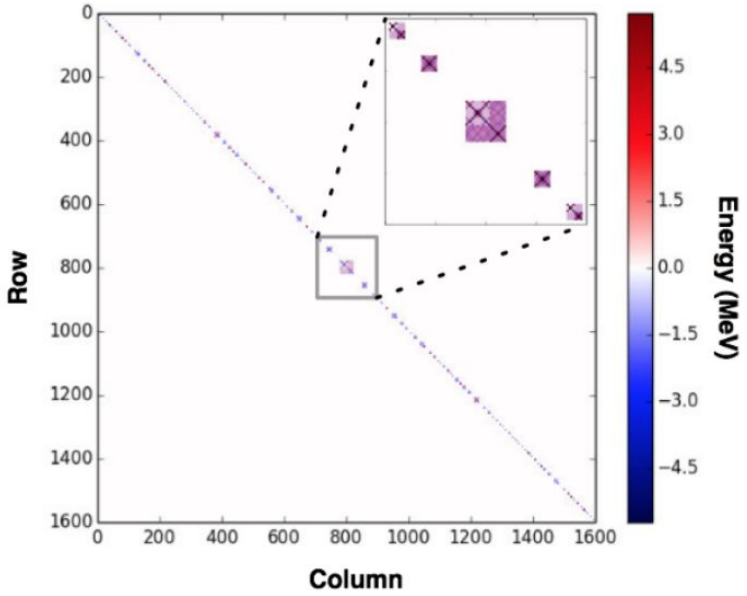
$$\langle ab|t_2|ij\rangle + = \frac{1}{4}\sum_{klcd} \langle kl|v|cd\rangle \langle cd|t_2|ij\rangle \langle ab|t_2|kl\rangle$$

One path to efficient tensor contractions:

1) Store an intermediate tensor to lower computational complexity
2) Use our knowledge of the physical system to make a custom sparse tensor format
3) Index manipulations for efficient contractions
4) Design algorithms and data structures to leverage leadership class supercomputers

Physical Symmetries

$$\vec{P}_a + \vec{P}_b \neq \vec{P}_c + \vec{P}_d \implies \langle ab|v|cd\rangle = 0$$



*a*b* → rows
*c*d* → cols

Conserved quantities on the diagonal.

8TB → 8GB!

Despite this, we still need many more basis states to converge interesting physical systems!

Aligning Indices

$$\langle kl|W|ij\rangle = \sum_{cd} \langle kl|v|cd\rangle \langle cd|t_2|ij\rangle$$

Definition of matrix multiplication:

$$C_{ij} = \sum_{k} A_{ik} B_{kj}$$

$kl \rightarrow I, \quad ij \rightarrow J, \quad cd \rightarrow K$

$$\langle I|W|J\rangle = \sum_{K} \langle I|v|K\rangle \langle K|t_2|J\rangle$$

Permuting to align indices

$$\langle kb|x|cj\rangle += \frac{1}{2}\sum_{dl}\langle kl|v|cd\rangle\,\langle db|t|lj\rangle$$

Generate V by permuting/combining *v* as {k,l,c,d} → {k*c,d*l}
Generate T by permuting/combining *t* as {d,b,l,j} → {d*l,b*j}
Perform gemm by combining indices
This results in a new tensor:

$$\langle kc|X|bj\rangle += \frac{1}{2}\sum_{dl}\langle kc|V|dl\rangle\,\langle dl|T|bj\rangle$$

So lastly we generate x by permuting/separating *X* as {k*c,b*j} → {k,b,c,j}

This is referred to as
TTGT (Transpose-Transpose-GEMM-Transpose) or
PPGP (Permute-Permute-GEMM-Permute).

Permuting to align indices

$$\langle kb|x|cj\rangle += \frac{1}{2}\sum_{dl}\langle kl|v|cd\rangle\,\langle db|t|lj\rangle$$

Generate V by permuting/combining *v* as {k,l,c,d} → {k*c,d*l}
Generate T by permuting/combining *t* as {d,b,l,j} → {d*l,b*j}
Perform gemm by combining indices
This results in a new tensor:

$$\langle kc|X|bj\rangle += \frac{1}{2}\sum_{dl}\langle kc|V|dl\rangle\,\langle dl|T|bj\rangle$$

So lastly we generate x by permuting/separating *X* as {k*c,b*j} → {k,b,c,j}

This is referred to as
TTGT (Transpose-Transpose-GEMM-Transpose) or
PPGP (Permute-Permute-GEMM-Permute).

OAK RIDGE National Laboratory

Part III: Fortran and Porting

U.S. DEPARTMENT OF ENERGY

Fortran + GPU

There are numerous approaches to running on GPUs

- Our strategy: Focus on working gnu implementation
- Use libraries where you can
- Massage equations into a form that can leverage library calls
- For non-library calls, we went with the most "manual" approach of:
  - CUDA kernel
  - C kernel launch wrapper
  - Call the kernel launch from Fortran through a C binding

Simple CUDA Kernel Example:

```
__global__ void simple_transpose_kernel(const double *src, double *dst, int sRows, int sCols){
  // src[row][col] = src[i][j]
  // j/cols/x go from left to right
  // i/rows/y go from top to bottom
  int start_row = blockIdx.y * TILE_DIM + threadIdx.y;
  int row;
  int col = blockIdx.x * TILE_DIM + threadIdx.x;

  for(int iter = 0; iter < TILE_DIM; iter += BLOCK_ROWS){
    row = iter + start_row;
    if(row < sRows && col < sCols){
      dst[row*sCols + col] = src[col*sRows + row];
    }
  }
}
```

Porting to AMD

- Crusher quick-start guide in OLCF docs is quite nice!
- HIPify scripts work quite well! Takes care of most of find/replace work.
- The primary manual work for us: 32 threads/warp vs 64 threads/warp
  - warpSize built-in variable instead of hard-coded sizes.
  - Watch out for tiled algorithms. 32x32 = 1024, while 64x64 = 4096 can spill
- Once these changes were made, we were pretty much good to go. Calls to rocBLAS or other libraries all look similar. Look out for custom struct definitions.

Simple HIP Kernel Example (it's the same!):

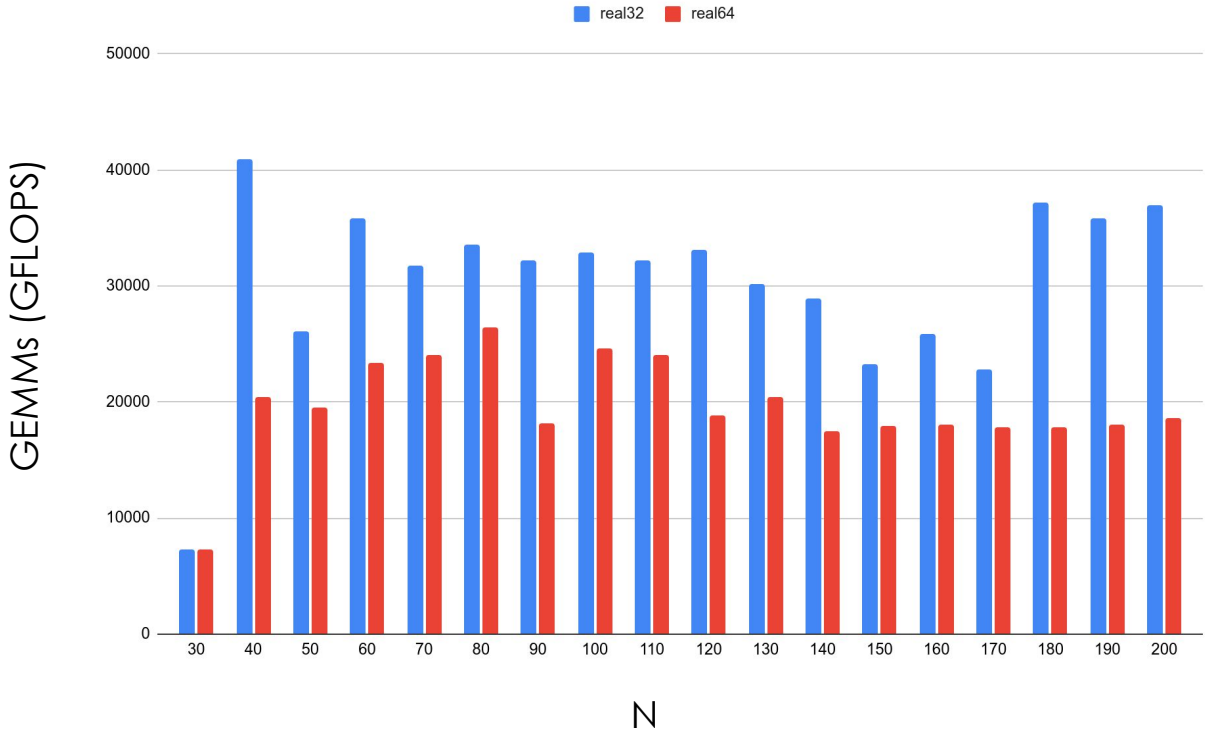
```
__global__ void simple_transpose_kernel(const double *src, double *dst, int sRows, int sCols){
  // src[row][col] = src[i][j]
  // j/cols/x go from left to right
  // i/rows/y go from top to bottom
  int start_row = blockIdx.y * TILE_DIM + threadIdx.y;
  int row;
  int col = blockIdx.x * TILE_DIM + threadIdx.x;

  for(int iter = 0; iter < TILE_DIM; iter += BLOCK_ROWS){
    row = iter + start_row;
    if(row < sRows && col < sCols){
      dst[row*sCols + col] = src[col*sRows + row];
    }
  }
}
```

Crusher Performance

-Single MI250x GCD

-C(p,q,r,s) = A(p,q,t,u)*B(t,u,r,s)

-All indices run from {1,...,N}

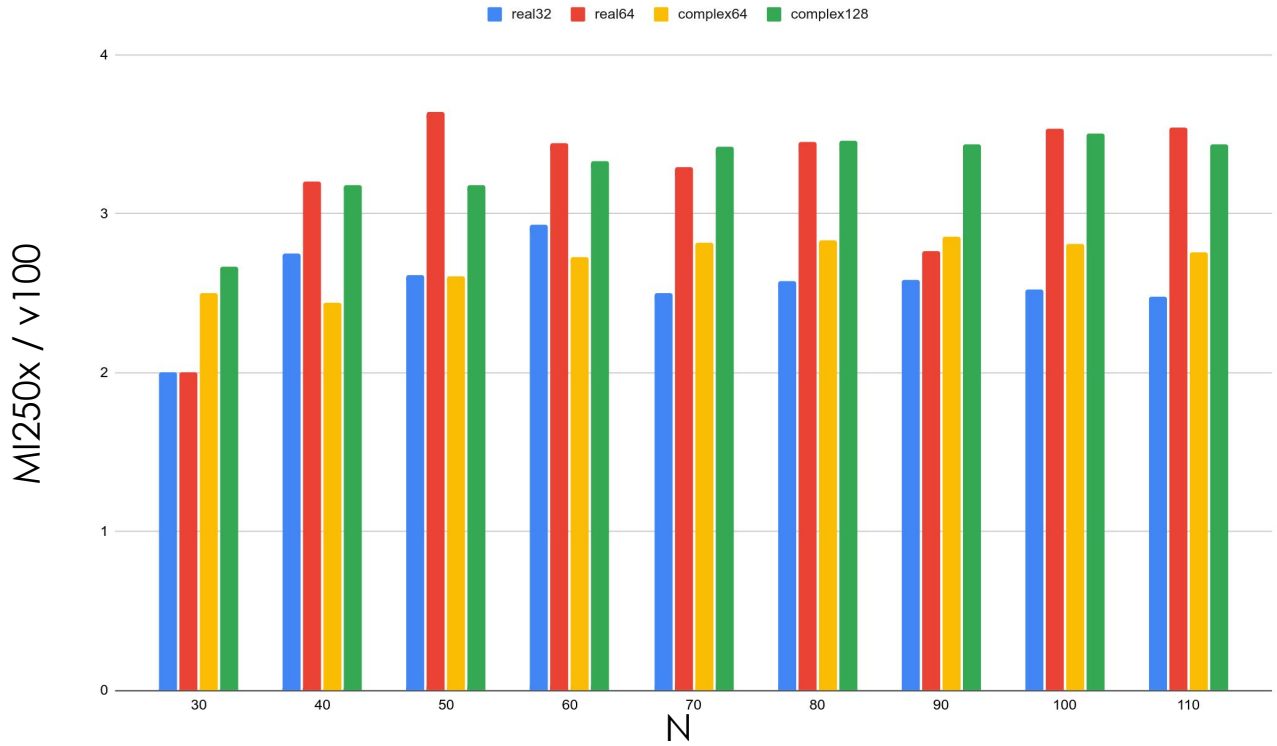


Crusher vs Summit performance

-Single MI250x GCD / v100

-C(p,q,r,s) = A(u,q,t,p)*B(s,u,r,t)

-All indices run from {1,...,N}

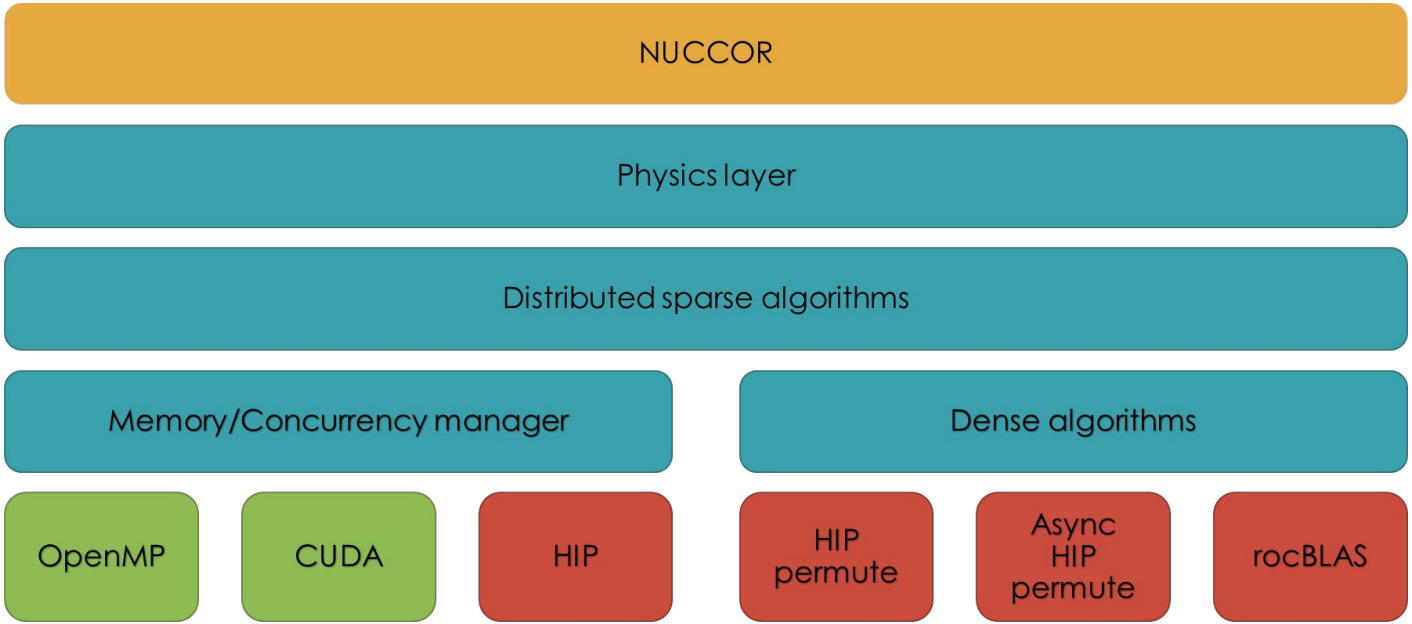


OAK RIDGE National Laboratory

Part IV: NTCL Design

U.S. DEPARTMENT OF ENERGY

NuCCOR stack



The NTCL

What is the NTCL?
The Nuclear Tensor Contraction Library.

Why is the NTCL?
Porting code can be hard! Accelerate things for domain scientists.

Why “nuclear”?
It’s general, but we can implement our domain specific knowledge too.

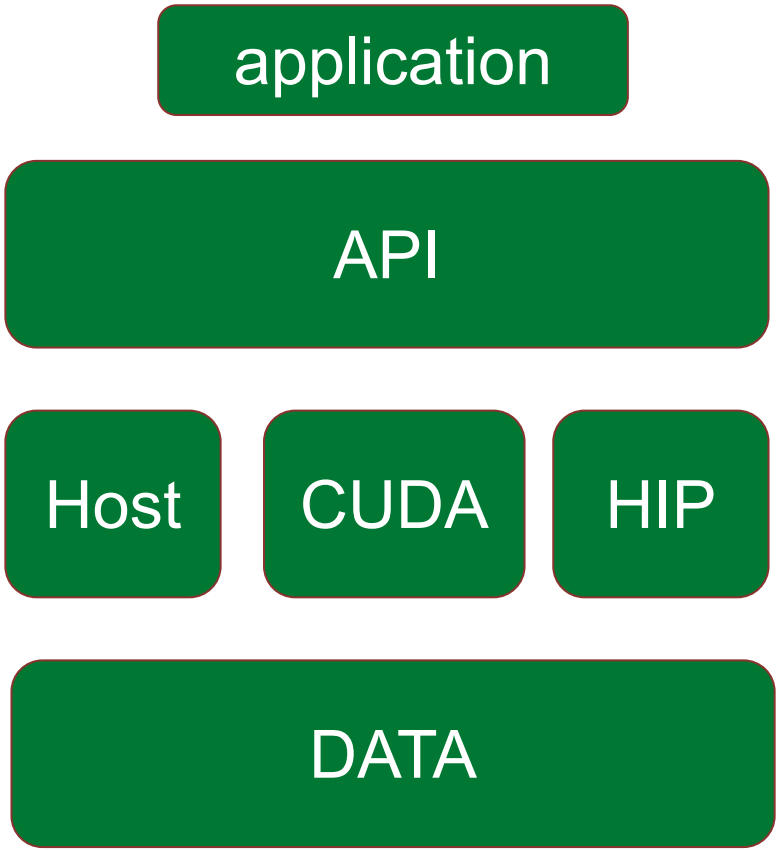
Ideas:

- Maximize code reuse and flexibility.
- Minimize dependencies and only compile what is necessary.
- These are implemented through object oriented interfaces and design patterns.
- Interfaces give portability to domain science part of code, but require a separate plugin for each implementation.

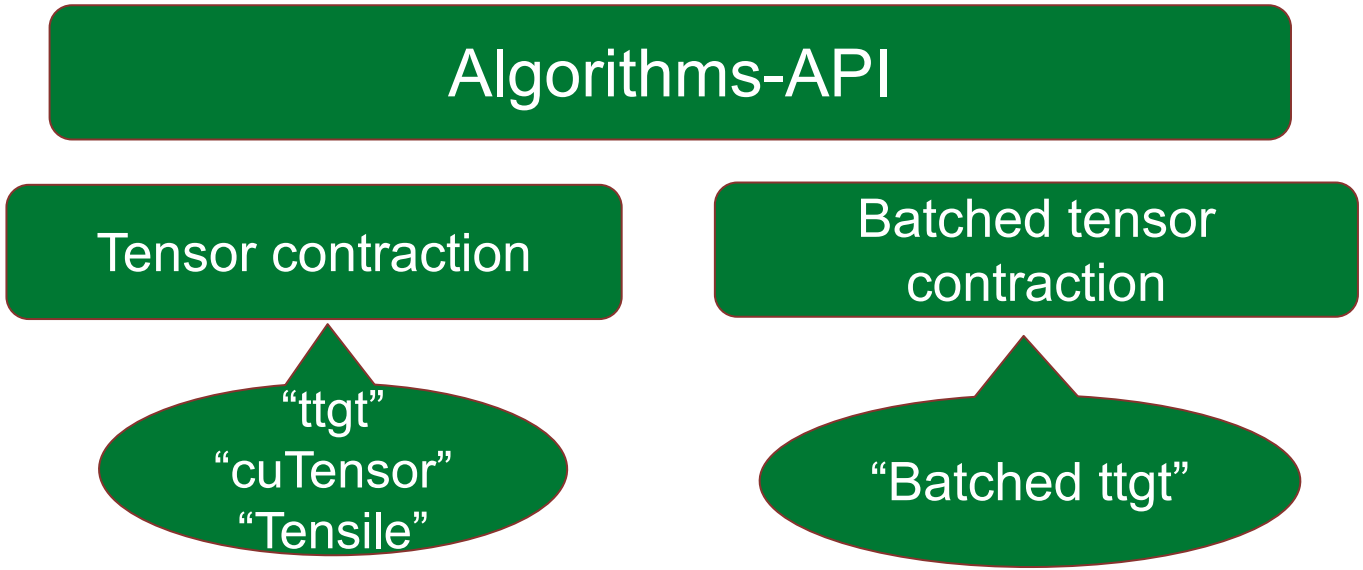
How to use the NTCL

```
class(tensor), allocatable :: ta, tb, tc
real(real64), dimension(5,3,7,2) :: c
real(real64), dimension(11,3,13,2) :: a
real(real64), dimension(5,13,7,11) ::b

class(tensor_contraction), allocatable :: tensor_contractor

call algorithms_initialize()

call random_number(a)
call random_number(b)
c = 0.0

call allocate_and_copy_tensor(ta, a, "device")
call allocate_and_copy_tensor(tb, b, "device")
call allocate_and_copy_tensor(tc, c, "device")

call tensor_contraction_factory%create(tensor_contractor, "c(a,b,i,j)=a(k,b,c,j)*b(a,c,i,k)", "buffered_ttgt")

call tensor_contractor%contract(tc, ta, tb)
```

NTCL: Design



NTCL: Design


Algorithms-API
Tensor contraction
Batched tensor contraction
“ttgt”
“cuTensor”
“Tensile”
“Batched ttgt”

A case from NuCCOR

$$\langle abc|t|ijk\rangle += \sum_d \langle kd|v|ab\rangle \langle dc|t|ij\rangle$$

Generate V by permuting/combining *v* as {k,d,a,b} → {a*b*k,d}
Generate T by combining *t* (RHS) as {d,c,i,j} → {d,c*i*j}
Perform gemm by combining indices
This results in a new tensor:

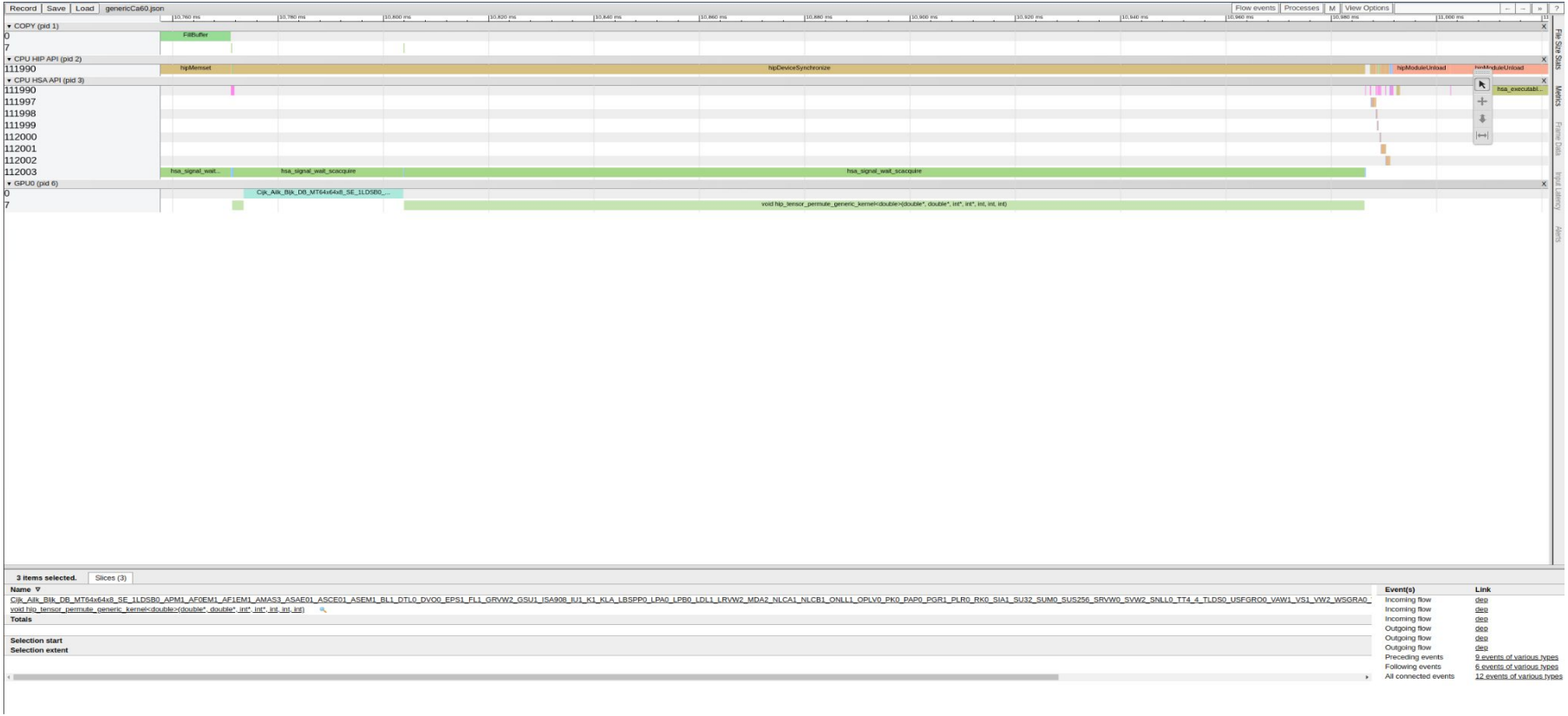
$$\langle abk|T|cij\rangle += \sum_d \langle abk|V|d\rangle \langle d|t|cij\rangle$$

So lastly we generate t (LHS) by permuting *T* as {a*b*k,c*i*j} → {a,b,c,i,j,k}

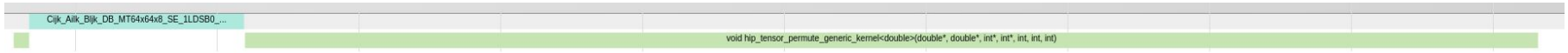
Notice:

1) t on RHS does not need to permute (combining indices costs nothing)
2) This operation is like a higher order outer product. Taking lower order tensors to make a higher order tensor.

ROCM Profiler using Chrome Tracing



ROCM Profiler for TTGT/PPGP



From command line profiler:
hip_tensor_permute_generic_kernel… = 86%
Cijk_Ailk_Bljk_DB_MT64x64x8 = 14%

From our timing tools:
Flop_rate: 727 GFLOPS

```
      bytes_allocated:          14.143867 GiB
      allocation_time:           0.119000 s
allocation_bandwidth:         118.856025 GiB/s
          bytes_read:           0.088585 GiB
       bytes_written:          14.055282 GiB
         flops_count:         169.782117 GFLOP
             tc_time:           0.233500 s
      read_bandwidth:           0.379380 GiB/s
     write_bandwidth:          60.193926 GiB/s
           flop_rate:         727.118275 GFLOPS
```

ROCM Profiler for Tailored Permute Kernel

From command line profiler:
permute = 54%
MatMul = 46%

From our timing tools:
Flop_rate: 1968 GFLOPS

```
      bytes_allocated:          14.143867 GiB
      allocation_time:           0.124000 s
allocation_bandwidth:         114.063443 GiB/s
         bytes_read:            0.088585 GiB
      bytes_written:           14.055282 GiB
        flops_count:          169.782117 GFLOP
            tc_time:           0.086500 s
     read_bandwidth:            1.024108 GiB/s
    write_bandwidth:          162.488805 GiB/s
         flop_rate:          1962.799042 GFLOPS
```

Many algorithms for each interface

Different physics targets have different tensor structure.
Most of our calculations need to perform the same tensor contraction millions of times (with different extents each time).
Some systems have large blocks that need to be split over many GPU's.
Some systems have tiny blocks that need to be aggregated into batched calls.

We want to target them separately!

Small dgemms mini-app (a few million small dgemms):

host blas time	10.54 seconds
time of cublasDgemm loop:	129.65 seconds
time of magmablas_dgemm_vbatched:	0.009452 seconds

What are the tensor needs in other domains?
Quantum chemistry? Data science? Quantum Computing?
Quite varied!

OAK RIDGE National Laboratory

Part V: Conclusion

NTCL: Coupled Cluster Case

1. Allocation

```
call allocate_and_create_tensor(v_pppp, [n, n, n, n], "real64", "device")
call allocate_and_create_tensor(v_hhhh, [n, n, n, n], "real64", "device")
call allocate_and_create_tensor(t2, [n, n, n, n], "real64", "device")
call allocate_and_create_tensor(hbar, [n, n, n, n], "real64", "device")
call allocate_and_create_tensor(f_pp, [n, n], "real64", "device")
call allocate_and_create_tensor(f_hh, [n, n], "real64", "device")
```

2. Build diagrams for contraction

```
call tensor_contraction_factory%create(diagrams(1)%t, &
          "v(a,b,i,j)=x(b,c)*t(a,c,i,j)", "loops")
call tensor_contraction_factory%create(diagrams(2)%t, &
          "v(a,b,i,j)=x(a,b,i,k)*t(k,j)", "loops")
call tensor_contraction_factory%create(diagrams(3)%t, &
          "v(a,b,i,j)=x(a,b,c,d)*t(c,d,i,j)", "loops")
call tensor_contraction_factory%create(diagrams(4)%t, &
          "v(a,b,i,j)=x(a,b,k,l)*t(k,l,i,j)", "loops")
```

3. contract

```
do iter =  1, niter
    call diagrams(1)%t%contract(hbar, f_pp, t2, scalar(1.0d0), scalar(1.0d0))
    call diagrams(2)%t%contract(hbar, t2, f_hh, scalar(1.0d0), scalar(1.0d0))
    call diagrams(3)%t%contract(hbar, v_pppp, t2, scalar(0.5d0), scalar(1.0d0))
    call diagrams(4)%t%contract(hbar, t2, v_hhhh, scalar(0.5d0), scalar(1.0d0))
```

NuCCOR experience takeaways

- Porting NuCCOR to Crusher was pretty straight-forward.
- We took portability to the next level and made every algorithm an interface.
- Flexibility to add new hardware capabilities and algorithms without affecting the domain science application.
- Small changes to an input file to swap hardware.
- Small changes to an input file to swap algorithms.
- We moved most of this functionality to a new library, but these concepts could be applied to any application
- Our code is now MUCH more flexible, which has other benefits on top of hardware portability.

OAK RIDGE National Laboratory

Thank you for your time!

Feel free to ask questions now or at lietzjg@ornl.gov

ORNL is managed by UT-Battelle LLC for the US Department of Energy

U.S. DEPARTMENT OF ENERGY