Design considerations for math libraries

Jeff Hammond (NVIDIA)

Context

- My primary experience with FFTs is via MiniDFT benchmarking 2016-2017.
- I've worked on NWChem continuously since 2006, with a focus on tensor contractions (i.e. not FFTs).
- The higher-level, distributed structure of all math libraries are roughly the same.

Bad Things

- Synchronization
- Unnecessary data movement
- Synchronization
- Load-imbalance
- Synchronization
- Assuming all 50+ cores should be used for computation.
- Synchronization

GPUs improve the situation quite a bit because asynchronous execution is normative, unlike CPU execution.

Bottom-up tensor contractions

Z(d,b,a,c) = X(b,**f**,a,**e**) * Y(d,**e**,c,**f**) // Einstein notation

X2(a,b,e,f) = X(b,f,a,e) // Transpose $O(N^4)$

Y2(e,f,c,d) = Y(d,e,c,f) // Transpose $O(N^4)$

Z2(a,b,c,d) = X2(a,b,e,f) * Y2(e,f,c,d) // Matrix multiplication $O(N^6)$

Z(d,b,a,c) = Z2(a,b,c,d) // Transpose $O(N^4)$

https://github.com/jeffhammond/spaghetty/blob/main/papers/old/spaghetty_chapter_v1.pdf

Bottom-up distributed tensor contractions

```
FORALL a,b,c,d,e,f:                          // index sets group in tiles
    Fetch X(b,f,a,e)                         // Communication O(N^4)
    X2(a,b,e,f) = X(b,f,a,e)                 // Transpose O(N^4)
    Fetch Y(d,e,c,f)                         // Communication O(N^4)
    Y2(e,f,c,d) = Y(d,e,c,f)                 // Transpose O(N^4)
    Z2(a,b,c,d) = X2(a,b,e,f) * Y2(e,f,c,d)  // Matrix multiplication O(N^6)
    Z(d,b,a,c) = Z2(a,b,c,d)                 // Transpose O(N^4)
    Update Z(d,b,a,c)                        // Communication O(N^4)
```

```
FORALL a,b,c,d,e,f:                              // index sets group in tiles
    NB Fetch X(b,f,a,e)                          // Communication $O(N^4)$
    NB Fetch Y(d,e,c,f)                          // Communication $O(N^4)$
    Wait on X, Y
    X2(a,b,e,f) = X(b,f,a,e)                     // Transpose $O(N^4)$
    Y2(e,f,c,d) = Y(d,e,c,f)                     // Transpose $O(N^4)$
    Z2(a,b,c,d) = X2(a,b,e,f) * Y2(e,f,c,d)      // Matrix multiplication $O(N^6)$
    Wait for Z to be available
    Z(d,b,a,c) = Z2(a,b,c,d)                     // Transpose $O(N^4)$
    NB Update Z(d,b,a,c)                         // Communication $O(N^4)$
```

Composing math and communication

- Everything should be asynchronous but not tedious to manage
  - Explicit events/requests are tedious.
  - Completion callbacks are okay, but how far can we recurse? And what’s the MPI situation?
  - CUDA streams work but streams are an implementation tool not the right semantic.
  - CUDA graphs and other graph-based execution models make sense.
  - Explicit dependencies ala OpenMP 4+ are pretty good, but lack an object model.

- There are known gaps with MPI here.
  - Completion callbacks are not supported (as in DCMF/PAMI).
  - Active messages are not supported directly (not that they are useful in FFTs).
  - Driving MPI with a progress thread that manages completions is often reasonable.

https://developer.nvidia.com/blog/cuda-graphs/

Top-down tensor contractions

Z(d,b,a,c) = X(b,**f**,a,**e**) * Y(d,**e**,c,**f**) // Einstein notation

```
Z["dbac"] = X["bfae"]*Y["decf"];    // actual C++ code for CTF
```

1) This is the only known way to build formally optimally implementations.
2) Everyone has to agree on the need for this solution and the notation.
3) This is fully synchronous at the term level and higher-level task parallelism is not supported by this interface.

https://solomonik.cs.illinois.edu/talks/molssi-monterey-may-2017.pdf

Suggestions for FFT people

- Higher-level interfaces will do a much better job adapting to novel memory hierarchies.
- Come up with the high level library interface that makes everyone happy, but make it asynchronous like a nonblocking collective.
  - Apparently, this is hard because use cases pretty wildly across quantum and non-quantum use cases.
  - Nonblocking != asynchronous. Must provide resources to drive background progress.
- Build FFT kernel components that fit into asynchronous tasking systems that allow composition of computation, communication, allocation, etc.
  - No asynchronous tasking system for Fortran except OpenMP 4.5 (tasking is CPU-only).