Wire-Cell Arrays

Brett Viren

October 24, 2024

Problem Context

Novel developments beginning now/soon:

All are “array-oriented” and benefit from GPU-acceleration.

- Prompt signal processing
- LAr pixel signal processing and simulation
- Porting prototype pattern recognition

Existing WCT data has baggage

Currently, no really solid “array” data type in WCT.

- We have Eigen but it lacks GPU support and many algorithms.
- We have flexible but non-array and complex data types, (`IFrame`/`ICluster`/`ITensor` and PC tree).
- What GPU support we have, requires CPU/GPU/CPU round trips and data transformations.

Basic solution concept

Mimic patterns in the successful Python packages: Numpy, cupy, JAX, PyTorch

- Whole-array operations allowing vectorizing and other optimizations.
- Transparent choice of execution on CPU or GPU and other heterogeneous patterns.
- Pass arrays without CPU/GPU/CPU round-trip.

While fitting into some WCT-specific needs

- Thread-safe exchange of data between WCT components.
- Efficient transfer to/from existing Torch-based inference (DNNROI).

And beautify some WCT ugliness

- Provide a simpler and general `IData` model than currently exists.
- Better alternative to some inconsistent and under-performing hand-wrought code.

Two “obvious” candidates

LibTorch “tensors”

- Good: WCT already uses it for DNNROI and `IDFT`, hugely popular with large community, generally good performance.
- Bad: Python-first, C++ second, truly horrible documentation, non-trivial install, not `const`-correct.

ArrayFire “arrays”

- Good: Fairly popular, C++-first, `const`-correct, excellent documentation.
- Bad: Somewhat poor performance in certain critical benchmarks.

Performance analysis

- WCT `OmnibusSigProc` was profiled to identify the major bottlenecks.
  - Convolution via FFT, median, sort and general array arithmetic.
- An apples-to-apples benchmark application was developed
  - Targets: ArrayFire (AF), LibTorch (LT) and Eigen/FFTW/std::algorithms (EI).
  - Abstract, templated core benchmark algs with generic templated array mini-API.
  - Each target specializes the mini-API.
- Configurable to explore 3 axes: array type, device and test.
- Times are in units of CPU-core or GPU millisecond
  - Average over many runs that span 1-10 seconds total.
- Results use random arrays of size 1024x8192.
  - Sizes relevant to LArTPC frames give consistent results.

Arithmetic

dev	tech	time
GPU	AF	0.14
GPU	LT	0.36
CPU	AF	33.40
CPU	LT	11.91
CPU	EI	18.78

This test simply applies add, multiply and accumulate.

FFT convolution

dev	tech	time	notes
GPU	AF	2.20	
GPU	AF	2.02	fwd is r2c
GPU	LT	0.89	
CPU	AF	667.74	
CPU	AF	414.30	fwd is r2c
CPU	LT	158.24	
CPU	EI	489.61	fftw3
CPU	EI	309.31	fftw3 fwd is r2c

Time includes forward DFT, multiplication in Fourier domain with pre-transformed kernel and return to interval domain with inverse DFT.

- The “r2c” label indicates explicit application of real-to-complex optimization for forward DFT.

Median and sort

median	GPU	AF	1.30	
median	GPU	LT	0.18	
median	CPU	AF	1109.09	
median	CPU	LT	76.73	
median	CPU	EI	90.98	`std::nth_element`
sort	GPU	AF	1.33	
sort	GPU	LT	1.55	
sort	CPU	AF	1119.37	
sort	CPU	LT	383.22	
sort	CPU	EI	317.24	`std::sort`

- Very likely that AF is implementing `median` with a `sort` instead of the partitioned algorithm.
- AF sort is surprisingly bad on CPU.

Problems with the solutions and possible mitigation

- AF DFT performance
- AF Median performance
- LT `const`-incorrectness
- LT lack of documentation

AF DFT performance

Problem: naive AF DFT is 3x slower than LT. LT/CPU beats FFTW (“impossible”).

- Likely LT is auto detecting when r2c and c2r optimization can be applied while AF is not.
- Manual r2c optimization added to AF and EI/FFTW benchmark and 3x goes to 2x.
  - Likely c2r would close the gap further and FFTW would (better!) match LT.
- AF/GPU did not speed up as much as expected with r2c.
  - Remaining 1ms or so difference is perhaps negligible

Mitigation

Adopting AF implies the need for WCT to effectively extend AF’s API to include DFT functions that reap r2c and c2r.

- r2c and c2r also require additional functions to handle Hermitian-symmetry.

Const-correctness issue

Problem: `torch::Tensor` is essentially `shared_ptr<T>`, T is never `const`

This is legal: `const torch::Tensor ct; ct.zero_();`

- The `zero_()` call mutates the "const" tensor.

Mitigation: develop `const`-correct facade class

- Never expose "bare" `torch::Tensor` and instead defined a "wrapper" or "facade".
- Replicate required subset of torch API - non-trivial effort.

Mitigation: always copy

A hypothetical `torch::Tensor IArray::array()` could always return a copy.

- Caller can mutate if they wish, copies add up (consider a large `FanOut`).

May also include a non-copy version and rely on trust:

- `torch::Tensor IArray::array_you_promise_not_to_mutate()`

Const-correcting LibTorch

WCT can define an “array facade” class which provides a subset of the `torch::Tensor` API.

- Need never expose the “bare” `torch::Tensor` type to the user.
- Can enforce `const`-correctness.
- Required API is large, requiring substantial effort to replicate.
- Could introduce some performance depredations.