Co-dfns 2016: GPU Performance, Workflow, and Usability

Aaron W. Hsu [arcfide@sacrideo.us](mailto:arcfide@sacrideo.us)

Indiana University, USA

Dyalog 2016, Glasgow, Scotland

# Recap from 2015

Last year the compiler was producing GPU code for select benchmarks and was ready to be put to the test against some more discerning users.

The compiler was also producing fused code and we had begun the work of supporting a wider variety of data types and operators.

Working towards a solid workflow for getting your code into Co-dfns and improving usability.

# Lanes and Threads and Warps, Oh My!

Most of 2016 has been about getting the performance and capability of the Co-dfns compiler significantly improved across a wider range of user-relevant benchmarks on the GPU. The compiler is now producing fast code for a much wider range of primitives and benchmarks.

Fast GPU implementations and efficient compilation of user-contributed benchmarks have been a key focus of this year.

GPU execution of Co-dfns programs is proving to be a solid source of increased performance for data-oriented processing.

## Performance Tipping Point

In our work with various primitives and benchmarks, a trend has emerged regarding the “tipping point” of performance for functional APL code on the GPU. We find that GPUs generally start to approach and exceed CPU performance numbers when we deal with greater than 1MB of data at a time on the GPU. Less than this and the GPUs tend to not have enough data or parallelism to keep busy.

This has interesting implications for bitvectors. Since bitvectors efficiently store data, it is very difficult to get “super” performance with certain bitvector operations. Generally the input sizes must be larger to see competitive numbers with GPUs and bitvectors. Some primitives are very fast with bitvectors, while others are merely 0.2× or the like faster than Dyalog as data sizes grow.

# Big New Shiny Things

Here are some major highlights of shiny new things this year:

* A caching API
* Vastly improved performance of certain operators
* Wider array of Runtime Primitives
* Improved mixed function code generation
* Better GPU implementations of primitives
* New performance benchmarks including user-submitted
* DWA Integration

## DWA Integration

User Story #1: An APL neophyte emails me about my compiler and starts to try it out. But he can’t get DWA and he’s running on a Mac!

Solution #1: Co-dfns now embeds its own MicroDWA into the code base directly, so that you don’t require your own DWA, and thanks to some courage from this user, we now have experimental Mac OS X support (don’t bring OS X to the Workshop)!

## Caching API

User Story #2: User has complicated code that the compiler can’t yet fully compile, but the hot spots occur in more than one place in the code.

Solution #2: Cache Data to the GPU to incrementally migrate code to the compiler.

Sometimes you want to integrate your code in piecemeal, but this isn’t convenient if you have a tight inner loop that needs accelerating but copying overheads prevent you from optimizing this piece.

Of course, the eventual solution is to just move more and more things to the GPU and make more and more things run faster and faster on Co-dfns. In the meantime, however, we have implemented a Caching API that allows you to create and store GPU allocated arrays and pass them to Co-dfns compiled functions. This avoids any copy and conversion overhead. Once you are done invoking various functions on the array values, you can convert the data back out at the end and copy it off the GPU.

This allows you to see intermediate speedups in your code as you continue to migrate pieces of the code to Co-dfns.

This also helps in cases where the compiler produces fast code for an inner loop, but the out loop uses a feature that isn’t presently supported in the compiler.

User Story #3: User has nice APL code that requires lots of crunching over bitvectors, doubles, and uses inner product and reduce heavily but also has a lot of mixed primitives over bitvectors.

Solution #3: Expand out the capabilities of Inner Product and Reduction to use more sophisticated algorithms and map onto the GPU much more efficiently, and tear out your hair trying to make bitvectors as fast as possible on the GPU.

## High-performance Operators

### Outer Product

Outer product will now execute primitive functions on the GPU with minimal overhead. Unfortunately, the execution of user-defined functions still means that we must execute potentially a single kernel per call to the user-defined function.

### Each

The Each operater has been recently refactored and revamped to work over more data types and to use the new code generators, which enables better and more specific code to be generated for primitive cases. It still has the same limitations as Outer Product regarding user-defined functions at the moment.

### Inner Product/Reduction

Reduction over matrices is fast. Likewise, with Inner Product in some cases we are able to see 300× and 400× speedups over the Interpreter.

Reduction (First Axis) over matrices has proven to be very fast, while reduction over standard vectors with certain primitive operands that Dyalog has highly optimized are often either break even or see a 2× speedup or slowdown depending on data size and type.

We’re taking advantage of the Row by Row algorithms that Roger Hui often presents as the basis for our Inner Product implementation. So far it’s serving well.

## Mixed/Scalar Primitive Improvements

We have spent a lot of time tweaking and optimizing the performance of certain critical mixed primitives that play key roles in certain benchmarks. Often this means handling things like bitvectors over non-power of 2 argument sizes better than the Interpreter.

User Story #4: We’ve got some eager hackers at universities and lurking about that want to implement high-performance cryptographic libraries that are much easier to audit and read than C/C++-based implementations, but they need a good language and the right primitives.

Solution #4: Make sure things like cryptographic randomness are readily and easily accessible in performant and reliable ways that don’t require lots of corner cases.

In a deviation from the Interpreter, we’ve implemented Roll (?) using more cryptographically appealing random number generation. We are looking at introducing large scale pseudo-random number generation for the GPU, but there has not been a large demand for it. On the other hand, we have a small team at some Universities that are highly interested in using Co-dfns to re-implement high-performance cryptographic libraries (such as OpenSSL or GnuPG) for which having a real random number generator as a core part of the language is a big boon.

A new code generation framework is gradually taking over the old, which enables us to make more optimizations about certain cases, such as the types and the rank of operations. This framework is designed to be extended with shape and range information in future changes to the compiler analysis passes. This is intended as a means of solving certain performance bottlenecks that make it necessary to use some special code for certain benchmarks to achieve maximum performance.

### Scan

User #5: Needs to implement a vector database or tree-transformation algorithms on the GPU without murdering their cat in the process.

Solution #5: Give them high-performance grouping, sorting, filtering, and scanning primitives.

Scan is extremely important. Actually, scratch that, Scan is Life on the GPU for non-trivial, apparently sequential algorithms.

It is a memory bound operation that is potentially the single most important operation for GPU execution of things like sorting, reduction, filtering, replicating, Key/Group/Multisets, &c.

We’ve put some significant time and effort into this one.

Fortunately, there is some literature out there about Scan primitives on the GPU, but they are mostly targeted towards Prefix Sum, and often don’t deal with higher-ranks (as in, anything not a vector).

To further make things complicated, many of the best-in-class algorithms rely on “Scientific Computing” quality code, which means that it works really well for exactly one machine, and maybe okay for everyone else.

We’ve managed a pretty good, semi-original implementation of Sklansky Scan Networks that are independent of the size of the array for their performance, don’t require excessive auto-tuning, and are general/high-level enough that they can reasonably be expected to work well over a variety of GPUs.

I have learned more about Scan this year than I ever thought was possible, and I’ve still only scratched the surface. Some of this is new research that may be published in future academic venues.

The good news: we now have a pretty good scan and can begin to more seriously tackle some of the operations we’ve had to avoid until now, including things like Key, Grade Up, and Grade Down. We have already begun to make inroads on replicate and, of course, the APL Scan.

Currently we are seeing about a 50 – 60% utilization of the theoretical peak memory bandwidth of the GPUs on which we are testing. This is about 2× slower than the best-in-class Scans out there which can achieve throughput rates equivalent to memcpy.

Peak theoretical memory bandwidth of the CPU we test on (Intel Xeon E5-1620v3 ) is 68 GB/s. Peak bandwidth on the NVIDIA GTX 980 is 224 GB/s and on the NVIDIA Tesla K40c it is 288 GB/s.

This gives us a theoretical performance cap of around 4× over the CPU, and we’re seeing about 2× on with the current implementation.

With Scan performing reasonably well, we now have a host of algorithms we can use to efficiently implement other core primitives that might have been intractable on the GPU without Scan.

Lesson Learned: Sometimes, existing algorithms and libraries simply don’t work, because they make assumptions that you can’t rely on, like that you’re always going to call scan with a nice, neat primitive like +.

## New Benchmarks

How about some new user-contributed benchmarks? Yes, please!

### Genokey

Motivated the creation of the caching API. It has a complex set of logic, not all of which is best executed on the GPU. It also has multiple hot spots distributed around the code in such a way that the compiler cannot currently compile the entire Namespace.

Instead, we use the Caching API to selectively compile the performance hot spots and avoid copy overheads in between these hotspots to move data to and from the GPU.

As a result, the data stays on the GPU until the final result is needed and we are able to see around a 1.5 – 3× performance improvement with minimal changing of the original code.

Many thanks to Gert L Møller for this exciting benchmark (giving us lots of work to do).

### ANN

Neural Networks are basically taking over the world.

APL is particularly well suited to expressing neural network code in a clean, concise way.

However, you need to train your neural networks on a lot of data. Dyalog is very fast, but we’re talking about days and days.

With the compiler, we are able to compile complete neural network code prototypes and see performance speedups of around 20× on relatively standard to small-ish datasets.

Many thanks to Romilly Cocking for this benchmark code.

# Co-dfns Programming Cycle, a.k.a. – how do I use this thing?!

Currently the compiler still requires a closed namespace with functional, non-recursive functions.

**The Process**

1. Identify Hotspots
2. Formulate them as dfns
3. Create a namespace to hold these functions
4. Call the namespace for each hot spot
5. Utilize Caching to avoid copying if required
6. Benchmarking tools can help to identify performance features

A key point is that the caching API will allow you to incrementally convert small parts of your code and gradually make the dfns larger and more encompassing until the point that you no longer need the caching API.

# Usability, Deployment, and Tracking Improvements

We’ve decided to shift the release schedule of the compiler. Chasing a “version 1” isn’t particularly helpful. Instead, we’re focusing on driving the development of the compiler on the basis of actual user programs, since spending significant amounts of effort implementing high-performance edge cases that no one uses doesn’t make much sense.

To work with these, we’ve had a few goals with regards to usability:

1. Easy to track releases
2. Easy to track progress
3. Easy to submit feedback and new benchmarks/user programs

The biggest change is a shift to a fixed release cycle, specifically monthly primary releases.

There is a file LIMITATIONS file that indicates some known limitations and issues with the compiler. This provides a “one stop” location for tracking features and language support in the compiler.

There are three main methods for helping to improve the compiler:

* (Easiest) Submit issues on Github
* (Nicer) Submit a pull request with new tests showing an issue with your code
* (Best) Submit a benchmark program as a pull request to the benchmarks repository showing the code you want to go fast

If you have code publication concerns, you can either submit a stripped benchmark or you can submit benchmarks through email letting us know that they are distribution limited.

# Final Thoughts

We continue to be driven primarily by user provided benchmarks and code. Spending months optimizing one case doesn’t make sense if you won’t ever use it!

At the moment we’re interested in all benchmarks, but especially those that have user-defined functions passed to operators where the computation of each individual call to the user-defined function is small, but there are many calls.

We are committed to a regular release schedule now and make it easy to track the compiler’s progress through GitHub and submit your own benchmarks through pull requests against the Benchmarks repository. If you have feature requests, either submit a pull request with a new test case or file an issue on GitHub.

**User Code Makes a Difference:**

* User #1: Mac OS X support and DWA Integration
* User #2: Caching API
* User #3: Higher performance bitvector algorithms and fast operators
* User #4: Cryptographically viable number generation and behaviors
* User #5: High-performance Scan algorithms

# Thank You!

Don’t forget to attend Thursday’s workshop on Co-dfns to get a hands-on experience with using the compiler with Romilly’s ANN code! You’ll also get the chance to learn how to submit your own benchmarks and see how things run on your machine (if you bring one).

<https://arcfide.github.io/Co-dfns/>

https://github.com/arcfide/Co-dfns-benchmarks

<https://gratipay.com/Co-dfns/>

Special thanks to Dyalog, Ltd. For their continued support of APL research and the Co-dfns project.