Bifrost
Easy High-Throughput Computing
github.com/ledatelescope/bifrost

Miles Cranmer (Harvard/McGill), with:
Ben Barsdell (NVIDIA), Danny Price (Berkeley), Hugh
Garsden (Harvard), Gregory Taylor (UNM), Jayce Dowell
(UNM), Frank Schinzel (NRAO), Lincoln Greenhill (Harvard)
The Problem:

Every 4 years, an astronomer is killed by inefficient pipeline development
The Problem:

Can take 1 year for a team to develop a high-throughput pipeline
• Say 5 new terrestrial telescopes each year
• Say 4 astronomers work on pipelines for these

\[(20 \text{ astronomer-years/year})/(80 \text{ years life exp.})\]  
\[\approx 1 \text{ astronomer killed every 4 years!}\]
Solution: **Bifrost**  
*A Pipeline Processing Framework*

*Bifrost saves lives™* *(well... it saves time)*
What is a “High-Throughput Pipeline”? 

• "High-throughput” 
  • 10-40+ Gbps per node 

• Pipeline: 
  • chain of processing elements working on a continuous stream of data 

“Data transfer”  “Processing element”
Why is this difficult?

• Each step works at their own pace
• Astronomy – can’t just scale up hardware
  • Need maximal efficiency
• Huge data flow on CPU & GPU
• Have to deal with continuous data flow
Bifrost pre-cursor: PSRDADA

- Warp-speed fast, but C API looks like this:

```c
int example_dada_client_writer_open (dada_client_t * client);
int64_t example_dada_client_write (dada_client_t * client, void* data,
uint64_t data_size);
int64_t example_dada_client_writer_write_block (dada_client_t *
client, void* data, uint64_t data_size, uint64_t block_id);
int example_dada_client_writer_close (dada_client_t * client, uint64_t
bytes_written);
typedef struct {
    dada_hdu_t * hdu;
multilog_t * log; /* logging interface
char * header_file; // file containing DADA header
char * obs_header; // file containing DADA header
char header_written; // flag for header I/O
} example_client_writer_t;
void usage()
{
    fprintf (stdout,
        "example_dada_client_writer [options] header\n"
        "-k key   hexadecimal shared memory key [default: %x]\n"
        "header  DADA header file contain obs metadata\n"
        "DADA_DEFAULT_BLOCK_KEY);"
    
    /* Function that opens the data transfer target */
    example_dada_client_writer_open (dada_client_t * client){
        assert (client != 0);
        example_client_writer_t * ctx = (example_client_writer_t *)
client>
ctx; assert (ctx != 0);

multilog (ctx->log, LOG_ERR, "could not allocate memory\n");
return (EXIT_FAILURE);
}
// read the ASCII DADA header from the file
if (fileread (ctx->header_file, ctx->obs_header,
DADA_DEFAULT_HEADER_SIZE) < 0) {
    free (ctx->obs_header);
    multilog (ctx->log, LOG_ERR, "could not read ASCII header from
%\n", ctx->header_file);
    return (EXIT_FAILURE);
}
ctx->header_written = 0;

/* Transfer header/data to data block */
int64_t example_dada_client_writer_write (dada_client_t * client,
void* data, uint64_t data_size){
    assert (client != 0);
    example_client_writer_t * ctx = (example_client_writer_t *)
client>
context;
    assert (ctx != 0);
    if (!ctx->header_written) {
        // write the obs_header to the header block
        uint64_t header_size = ipbuf_get_bufsz (ctx->hdu->header_block);
        char * header = ipbuf_get_next_write (ctx->hdu->header_block);
        memcpy (header, ctx->obs_header, header_size);
        // flag the header block for this "observation" as filled
        if (ipbuf_mark_filled (ctx->hdu->header_block, header_size) == 0) {
            ctx->header_written = 1;
            example_client_writer_t * ctx = (example_client_writer_t *)
client>
context;
            assert (ctx != 0);
        }
        ctx->header_written = 0;
    }
    else {    // write data to the data_size bytes to the data_block
        memset (data, 0, data_size);
        return data_size;
    }
    /*! Transfer data to data block, 1 block only */
    int64_t example_dada_client_writer_write_block (dada_client_t *
client, void* data, uint64_t data_size, uint64_t block_id){
        assert (client != 0);
        example_client_writer_t * ctx = (example_client_writer_t *)
client>
context;
        assert (ctx != 0);
        // write 1 block of data
        memset (data, 0, data_size);
        return data_size;
    }
    /* Function that closes socket */
    int example_dada_client_writer_close (dada_client_t * client, uint64_t
bytes_written){
        assert (client != 0);
        example_client_writer_t * ctx = (example_client_writer_t *)
client>
context;
        assert (ctx != 0);
```
Radio astronomy pipelines need:

• Maximal efficiency
• High-throughput
• Long deployments

What about productivity?

Arranging this should be simple!

Why does it need to be immensely complicated?
Rings and Blocks
Exhibit A

• Want to do:
  file read -> GPU STFT -> file write

• What comes most naturally?
  • Functions applied to results of other functions...
  • So... make that the API
data = bf.blocks.read_wav(['file1.wav'], gulp_nframe=4096)
chunks = bf.views.split_axis(data, 'time', 256)
fft = bf.blocks.fft(chunks, axes='time_split', square=True)
fft = bf.blocks.transpose(fft, ['time', 'pol', 'freq'])
quantized = bf.blocks.quantize(fft, 'i8')
bf.blocks.write_sigproc(quantized)
Create a block object which reads in data at a certain rate

data = bf.blocks.read_wav(['file1.wav'], gulp_nframe=4096)
chunks = bf.views.split_axis(data, 'time', 256)

Modify block, chunk up the time series
Implicitly pass ring buffer within block to input ring of next block

```python
ffted = bf.blocks.fft(chunks, axes='time_split', square=True)
ffted = bf.blocks.transpose(ffted, ['time', 'pol', 'freq'])
```

Move around axes with labels

```python
quantized = bf.blocks.quantize(ffted, 'i8')
bf.blocks.write_sigproc(quantized)
```

Convert data type and write to disk
pipeline.run()

(start threads)
What did we lose?
  Some overhead to Python

What did we save?
  Our sanity, our time, etc.
Exhibit B: Alice & Bob

Two astronomers want to collaborate on an app. But...

- **Bob** writes a dedispersion code in C/CUDA, **Alice** writes a harmonic sum code in NumPy/PyCUDA
- **Bob** outputs volume units with barn-megaparsecs, **Alice** wants cubic furlongs
- Both use **different axes**

... They also don’t talk to each other...
How do we make this unfortunate collaboration *painless*?

*modularity, modularity, modularity, modularity*
Modularity = Blocks and Metadata

• Blocks are black box algorithms, input/output through rings
• Ring headers describe data
  • Units
  • Axes
  • GPU/CPU
  • ...

• Blocks can’t see each other; fit together seamlessly
Exhibit C: Target of Opportunity (ToO)

10 Gb/s telescope backend – must insert algorithms in pipeline for ToO observation – need to do it in 20 minutes!

Need to:
1. Average data
2. Matrix multiply
3. Element-wise square

And it all has to be on the **GPU**!
Bifrost = Pipeline Framework, Block Library

Accumulate:

```python
blocks.accumulate(data, 10)
```

Matrix multiply (with a constant):

```python
blocks.matmul(a, const)
```

CUDA JIT compiler (inside a user-defined block):

```python
bf.map("c = b*b", b=b, c=c)
```
Bifrost: Deployed on LWA SV (telescope)

- 34 MHz live stream at LWA TV Channel 2:
  - phys.unm.edu/~lwa/lwatv2.html
  - (Or, Google: “LWA TV”, click first link, go to channel 2)
Questions?

• GitHub: /ledatelescope/bifrost
• Paper in prep, to be submitted to JAI

• LWA 2 live stream: phys.unm.edu/~lwa/lwatv2.html