BLADE
Allen Telescope Array GPU accelerated beamformer backend
Talk Summary

- Brief Allen Telescope Array history and capabilities.
- On-site hardware pipeline description.
- Comparison between FPGA and GPU for Digital Signal Processing.
- Short interferometry and beamforming theory explanation.
- Introduction to BLADE and its architecture.
- Heterogeneous memory management (Blade::Memory).
- Generic work-unit (Blade::Module).
- Compute command buffer (Blade::Pipeline).
- Execution (Blade::Runner, Blade::Plan).
- Production pipeline example.
- Extra performance tips.
Allen Telescope Array
Hat Creek, CA

- Radio-telescope located in northern California was built from 2002 to 2007.
- Designed to conduct astronomical observations and search for extraterrestrial intelligence (SETI).
- Named after Microsoft co-founder Paul Allen, who provided significant funding for the project.
- Recently refurbished for improved reliability and sensitivity. Thanks to generous private donations.
Allen Telescope Array
Hat Creek, CA

Composed of 42 randomly distributed dishes.
Allen Telescope Array
Offset Gregorian Dish

- Each of the 42 antennas has 20 feet (6.2 meters) in diameter.
- Produce ~1.5 GHz of bandwidth for each polarization (~3.0 GHz in total).
- The entire telescope equated to ~60 GHz or ~1 Tbps at 8 bits per sample.
- Connected to the DSP Room via RF over fiber.
- Ultra wide band reception from 900 MHz to 12 GHz.
Allen Telescope Array
Log-periodic Feed

• Ultra wide band capabilities are made possible by the log-periodic feed-horn.

• Cryocooled to improve the sensitivity of higher frequencies.

• New Antonio feed is surrounded by a glass dome to help maintain the vacuum and shield it from moisture.
Allen Telescope Array
Compute Cluster

- Compute cluster composed of 8 dual-socket EPYC servers with two NVIDIA RTX 3090 class graphics cards.
- Most of the computing is performed by the GPU.
- Two Mellanox Connect-X 5 dual 100G.
- Two PCIe x16 cards for NVMe cache.
Allen Telescope Array
RF Front-end (RFSoC)
Allen Telescope Array
Antenna Data Stream

• Each antenna is similar to a very large Software Defined Radio (RTL-SDR, HackRF, USRP, etc).

• It produces a stream of complex numbers that represent the radio spectrum in the time-domain.

• Each complex sample is composed of two numbers representing the real (I) and imaginary (Q) parts.

• Visualization can be done in the frequency-domain using a Fast Fourier Transform (FFT).

• Examples are beamformed recordings made by the ATA of the satellite JPSS-1 (NOAA-20) and a random LTE Downlink.
Allen Telescope Array
Scientific Contributions

Fast Radio Burst Discovered by SETI Institute’s Allen Telescope Array
Jun 3, 2021

The Astronomer’s Telegram
@astronomerstel

(NEW) ATel 15735: Bright radio bursts from the active FRB 20220912A detected with the Allen Telescope Array... astronomersteam.org/?read=15735
Allen Telescope Array

Beamforming

• Signal processing technique used in sensor arrays (e.g., antennas or microphones) to focus on signals from a specific direction.

• Involves applying different weights and phases to the signals received by each element in the array.

• The weighted signals are then summed together to create a single output signal.

• The weights and phases are chosen to enhance signals from the desired direction (main lobe) and suppress signals from other directions (side lobes).

• Beamforming is used in applications such as wireless communication, radar, sonar, and audio processing to improve signal quality and spatial selectivity.

\[ y(t) = \sum_{n=1}^{N} w_n \cdot x_n(t) \]
GPU vs FPGA for radio-astronomy

**FPGA**

*Pros:*
- Low latency with deterministic timing.
- More energy efficient (MHz/W).

*Cons:*
- Development Complexity.
- Scarce developer availability.
- Takes time to reprogram.
- Expensive.

**GPU**

*Pros:*
- Off-the-shelf availability.
- Larger developer availability.
- Average development complexity.
- Reprogrammed easily.
- Cheaper.

*Cons:*
- Bad latency. Bad timing.
- Less energy efficient (MHz/W).
BLADE
Breakthrough Listen Accelerated DSP Engine

• Responsible for most of the Digital Signal Processing of the ATA.
• Currently processing data incoming from 20 antennas with more soon!
• Each antenna represents ~3.0 GHz of bandwidth in 8 bits samples.
• Equates to an aggregated ~960 Gbps in 16 instances (~60 Gbps/instance).
• Currently implements 8 processing modules (beamforming, channelization, etc).
• Design rules followed:
  • Performant while hackable.
  • Not afraid to use new tech (C++20, JIT Kernels, etc).
  • Low number of dependencies.
BLADE
Beamforming & Stuff™

Analog Cluster
EACH
Xilinx RFSoC FPGA
12 GHz 8-bits ADCs
200G Networking

Digital Signal Processing Cluster

HPGUPPI
CPU-based

BLADE
GPU-based

HPGUPPI
CPU-based

On-Site DSP Room

EACH
2x AMD EPYC
2x NVIDIA A5000 (24 GB VRAM)
2x NVIDIA Mellanox Dual 100G NIC
2x 8TB NVMe Cache
Each module represents a compute operation (cast, beamforming, channelization, polarization, etc).

A sequence of modules is contained inside a pipeline. It’s also responsible to interface with the host device and hold staging buffers.

Runner will hold one or more (ideally two) pipelines described as workers. The runner will asynchronously schedule the execution optimizing for maximum parallelization.
Heterogeneous Memory Management

- Device-independent interface.
- Avoid RAW pointer handling.
- Automatic allocation lifecycle.
- Multidimensional array support.
- Ability to provide custom accessors.
Heterogeneous Memory Management
First Iteration: std::span

```cpp
{
    float* _buffer;
    size_t _size = 42;
    cudaMallocManaged(&_buffer, sizeof(float) * _size);
    std::span<float> vector(_buffer, _size);
}
```

```cpp
constexpr size_type size_bytes() const noexcept;
constexpr size_type size() const noexcept;
constexpr pointer data() const noexcept;
...```

Heterogeneous Memory Management

• Device-independent interface.
✓ Avoid RAW pointer handling.
• Automatic allocation lifecycle.
• Multidimensional array support.
• Ability to provide custom accessors.
class Vector {
    public:
        explicit Vector(const size_t& size) {
            float* _buffer;
            cudaMallocManaged(&_buffer, size);
            _holder = std::span<float>(_buffer, size);
        }

        ~Vector() {
            if (!_holder.empty()) {
                cudaFree(_holder.data());
            }
        }

    private:
        std::span<float> _holder;
};
Heterogeneous Memory Management

• Device-independent interface.
✓ Avoid RAW pointer handling.
✓ Automatic allocation lifecycle.
• Multidimensional array support.
✓ Ability to provide custom accessors.
Heterogeneous Memory Management

Third Iteration: class Vector<enum Device>;

```cpp
enum class Device : uint8_t {
    CPU = 1 << 0,
    CUDA = 1 << 1,
    METAL = 1 << 2,
    VULKAN = 1 << 3,
};

template<Device DeviceId, typename Type>
class Vector {
    public:
        explicit Vector(const size_t& size) {
            if constexpr (DeviceId == Device::CUDA) {
                ...
            }
            if constexpr (DeviceId == Device::CPU) {
                ...
            }
        }
};

Vector<Device::CUDA> cudaVec(42);
Vector<Device::CPU> cpuVec(42);
```
Heterogeneous Memory Management
Third-ish Iteration: class Copy(Vector<DstD>, Vector<SrcD>);

```cpp
template<Device DstDevId, Device SrcDevId, typename Type, typename Dims>
static const Result Copy(Vector<DstDevId, Type, Dims>& dst, 
 const Vector<SrcDevId, Type, Dims>& src, 
 const cudaMemcpyKind& kind) {
    if (dst.size() != src.size()) { ... }
    if (dst.shape() != src.shape()) { ... }

cudaMemcpyAsync(dst.data(), src.data(), src.size_bytes(), kind, stream);
    ... 
}
```

```cpp
Vector<Device::CUDA> cudaArr;
Vector<Device::CPU> cpuArr;
Copy(cudaArr, cpuArr);
```
Heterogeneous Memory Management

✓ Device-independent interface.
✓ Avoid RAW pointer handling.
✓ Automatic allocation lifecycle.
  • Multidimensional array support.
✓ Ability to provide custom accessors.
Heterogeneous Memory Management

Forth Iteration: class Vector : public Shape;

template<U64 Dimensions>
class Shape {
public:
    using Type = std::array<U64, Dimensions>;
    ...
    const U64 size() const { ... }
    const U64 offset(const Type& index) const { ... }
    const U64 dimensions() const { ... }
}

Vector<Device::CUDA, F32, 2> arr({4, 4});
arr.size();      // 16
arr.offset({1, 1})  // 6
arr[{1, 1}] = 42.0;  // C++20
arr[1, 1] = 42.0;    // C++23

template<Device DeviceId, typename Type, typename Dimensions>
class Vector : public Shape<Dimensions> {
...
    DataType& operator[](const typename Shape::Type& shape) {
        return _data[_shape.offset(shape)];
    }
}
Heterogeneous Memory Management

✓ Device-independent interface.
✓ Avoid RAW pointer handling.
✓ Automatic allocation lifecycle.
✓ Multidimensional array support.
✓ Ability to provide custom accessors.

✧ Zero-allocation profiling.
✧ Permutations.
Heterogeneous Memory Management

Bonus: Zero-allocation Profiling

```c++
explicit Vector(...) : ... {
    if (Memory::Profiler::IsCapturing()) {
        if (DeviceId == Device::CPU) {
            Memory::Profiler::RegisterCpuAllocation(size_bytes());
        } else if (DeviceId == Device::CUDA) {
            Memory::Profiler::RegisterCudaAllocation(size_bytes());
        }
        return;
    }
    ...
}
```

Memory::Profiler::StartCapture();
Vector<Device::CUDA, F32, 4> arr({4, 4});
Vector<Device::CPU, F32, 4> arr({4, 4});
...
Memory::Profiler::StopCapture();
Memory::Profiler::PrintCapture();

==== Memory Profile Capture ====
--- CUDA -----------------------
Allocated: 15.2 GB
Deallocated: 15.2 GB
Tensors Allocated: 5
Tensors Deallocated: 5
--- CPU ------------------------
Allocated: 1.2 GB
Deallocated: 1.2 GB
Tensors Allocated: 2
Tensors Deallocated: 2
Heterogeneous Memory Management
Bonus: Permutation/Transposition

How to do a transposition or permutation:

• Don't.
• If you really really need to: Use indexing for an inline transposition inside the compute kernel.
• If you don't have control over the next step: No alternative other than performing a discrete transposition by copying data.
Module

The Generic Work-Unit of Blade

- Generic module configuration for telescopes.
- Accept constant input buffers of N types.
- Produce constant output buffers of N types.
- Hold compute kernels.
Module

Generic Configuration

template<typename IT, typename OT>
class BLADE_API Generic : public Module {
public:
  // Configuration
  struct Config {
    BOOL enableIncoherentBeam = false;
    BOOL enableIncoherentBeamSqrt = false;

    U64 blockSize = 512;
  };

  constexpr Config& getConfig() const;
...
template<typename IT, typename OT>
class BLADE_API Generic : public Module {
public:

... // Input
struct Input {
    const ArrayTensor<Device::CUDA, IT>& buf;
    const PhasorTensor<Device::CUDA, IT>& phasors;
};
constexpr const ArrayTensor<Device::CUDA, IT>& getInputBuffer() const;
constexpr const PhasorTensor<Device::CUDA, IT>& getInputPhasors() const;

// Output
struct Output {
    ArrayTensor<Device::CUDA, OT> buf;
};
constexpr const ArrayTensor<Device::CUDA, OT>& getOutputBuffer() const;
...}


Module Instantiation

- Allocations made inside the constructor.
- Setup and forget method.

```cpp
template<typename IT, typename OT>
class BLADE_API Generic : public Module {
public:
    // Configuration
    ...
    // Input
    ...
    // Output
    ...
    explicit Generic(const typename Generic<IT, OT>::Config& config,
                     const typename Generic<IT, OT>::Input& input,
                     const cudaStream_t& stream);
    const Result process(const cudaStream_t& stream);
};
```

```cpp
template<typename IT, typename OT>
template<
    template<typename IT, typename OT> class BLADE_API ATA
>

```

```cpp
// Configure kernels.
BL_CHECK_THROW(this->createKernel(
    // Kernel name.
    "main",
    // Kernel function key.
    "ATA",
    // Kernel grid & block sizes.
    ...
    // Kernel templates.
    ...
);

// Allocate output buffers.
this->output.buf = ArrayTensor<Device::CUDA, OT>(getOutputShape());

// Print configuration values.
...
```

```cpp
template class BLADE_API ATA<CF32, CF32>;
```
Module
Just-in-Time Kernel Compilation

- Runtime compilation of the CUDA kernel.
- Possible to use templates to improve kernel performance.
- Faster instantiation of a module with any configuration parameter.
- Uses NVRTC (NVIDIA Runtime Compiler) wrapped by Jitify.
- Equates to smaller binary sizes.

```
template<uint64_t NBEAMS, uint64_t NANTS, uint64_t NCHANS, 
          uint64_t NTIME, uint64_t NPOLS, uint64_t TBLOCK, 
          bool EnableIncoherentBeam, bool EnableIncoherentBeamSqrt>
__global__ void ATA(const cuFloatComplex* input, 
                     const cuFloatComplex* phasor, 
                     cuFloatComplex* out) {
...
}
```
Module
Beamformer

- First to be developed.
- Combines the signal received by individual antennas into a single larger disk.
- Accomplished by delaying the signal slightly using a phasor.
- Supports electronic steering and multiple beam generation.
Module
High-Resolution Spectrogram

- Based on cuFFT.
- Received data is pre-channelized down to 500 kHz in resolution.
- The ideal channelization resolution is around 1 Hz for SETI search.
- The module further channelizes the data using FFT.
- Total of 1.5 million bins FFT of a time-domain vector for each polarization.
- Translates into a 3 million bins (3 GHz) FFT done every second.

And many more...
Pipeline

Compute Command Buffer

- Composed of a collection of modules.
- Holds host to device staging buffers.
- Provides synchronization methods.
-Executes CUDA kernels provided by modules inside CUDA graph for improved performance.

```cpp
class BLADE_API Pipeline {
public:
    const Result synchronize();
    bool isSynchronized();
    const Result compute();
    template<typename Block> void connect(3);
};
```
Pipeline
Module Initialization

• If necessary, the initial buffer is allocated.
• Modules are connected to the pipeline inside the constructor.
• Each module is initialized immediately and allocates its memory buffers.
• The output buffer of a module can be accessed by the next using the getter methods.
• Since the allocation was already made, the next module can calculate the necessary resources.

```cpp
// Allocating pipeline buffers.
this->input = ArrayTensor<Device::CUDA, CI8>(config.inputShape);

// Instantiating input cast from I8 to CF32.
this->connect(inputCast, {}, {
    .buf = this->input,
});

// Instantiating pre-beamformer channelizer.
this->connect(channelizer, {
    .rate = config.preBeamformerChannelizerRate,
}, {
    .buf = inputCast->getOutputBuffer(),
});

// Instantiating polarizer module.
this->connect(polarizer, {
    .mode = Polarizer::Mode::XY2LR,
}, {
    .buf = channelizer->getOutputBuffer(),
});
```

Pipeline
Transfer Descriptors

- Transfer methods are defined inside the Pipeline class.
- Copy data from the host to device.
- Copy operations are also asynchronous.
- Calling orchestrated by Blade::Runner.

```cpp
template<typename OT>
const Result ModeB<OT>::transferIn(const Tensor<Device::CPU, F64>& blockJulianDate,
                                   const Tensor<Device::CPU, F64>& blockDut1,
                                   const ArrayTensor<Device::CPU, CI8>& input,
                                   const cudaStream_t& stream) {

  // Copy input to staging buffers.
  BL_CHECK(Memory::Copy(this->blockJulianDate, blockJulianDate));
  BL_CHECK(Memory::Copy(this->blockDut1, blockDut1));
  BL_CHECK(Memory::Copy(this->input, input, stream));

  return Result::SUCCESS;
}
```
Pipeline
CUDA Graph Execution

- Modules execution cached inside CUDA Graph.
- Results in improved CPU usage and faster GPU computing.
- Automatic optimization is completely transparent to the user and developer.
Concurrent Execution

*Blade::Runner<>*, *Blade::Plan<>*

- The Runner provides an asynchronous execution queue for Pipeline.
- It supports multiple in-flight executions. Similar to a worker pool.
- The Plan offloads some boilerplate from the user.

```cpp
template<class Pipeline>
class BLADE_API Runner {
public:
  explicit Runner(const U64& numberOfWorkers,
                  const typename Pipeline::Config& config);
  bool enqueue(const std::function<const U64(Pipeline&)>& jobFunc);
  bool dequeue(U64* id);
};
```

```cpp
class BLADE_API Plan {
  template<class T> static void Available(const std::unique_ptr<Runner<T>>& runner);
  static void Dequeue(auto& runner, U64* id);
  template<class Pipeline> static void Compute(Pipeline& pipeline);
  template<typename... Args> static void TransferIn(auto& pipeline, Args... transfers);
};
```
Concurrent Execution
Asynchronous Worker Pool

Blade
Compute
Job 1

Blade
I/O Transfer (PCIe)
Job 1

Blade
Compute
Job 2

Blade
I/O Transfer (PCIe)
Job 2

Blade
Compute
Job 0

Blade
I/O Transfer (PCIe)
Job 0

Blade
Compute
Job 1

Blade
I/O Transfer (PCIe)
Job 1

Blade
Compute
Job 2

Blade
I/O Transfer (PCIe)
Job 2

Blade
Compute
Job 3

Blade
I/O Transfer (PCIe)
Job 3

Blade
Compute
Job 4

Blade
I/O Transfer (PCIe)
Job 4

Resource Utilization

Compute

50%

I/O Transfer (PCIe)

50%

Compute

~100%

I/O Transfer (PCIe)

~100%
Concurrent Execution
Enqueueing Async Job

while (...) {
    ...
    runner->enqueue([&](auto & worker) {
        // Check if runner has free slot.
        Plan::Available(runner);

        // Convert C pointers to Blade::Vector.
        auto input = ArrayTensor<Device::CPU, CI8>(input_ptr, worker.getInputBuffer().shape());
        auto output = ArrayTensor<Device::CPU, CF32>(output_ptr, worker.getOutputBuffer().shape());

        // Transfer input data from CPU memory to the worker.
        Plan::TransferIn(worker, julianDate, dummyDut1, input);

        // Compute block.
        Plan::Compute(worker);

        // Transfer output data from the worker to the CPU memory.
        Plan::TransferOut(output, worker.getOutputBuffer(), worker);

        return id;
    });
}
Mode B
Beamforming
Detection & Integration

Mode H
High-Spectral Resolution
Channelization (~1 Hz/bin)
Blade Mode B + RAWSPEC

**Real-Time Pipeline**

ADC/FPGA/NIC Stack → Blade Beamforming → NVMe

**Mode B + RAWSPEC**
- Write: 3.2 GB/s
- Read: 3.2 GB/s

**New**

ADC/FPGA/NIC Stack → Blade Beamforming → RAWSPEC → NVMe

**Sample Observation**
- Array Tensor: [A: 20, F: 192, T: 8192, P: 2]
- Integration: 4 samples
- Detection: Stokes-I (F32)

Blade Mode BH

**Real-Time Pipeline**

ADC/FPGA/NIC Stack → Blade Beamforming → HI-RES Channelization → NVMe

**Mode BH**
- Write: 0.4 GB/s
- Read: N/A

**NEW**

ADC/FPGA/NIC Stack → Blade Beamforming → HI-RES Channelization → NVMe

**NVMe I/O**
- Write: 0.4 GB/s

**NVMe I/O**
- Write: 3.6 GB/s
- Read: 3.2 GB/s

**NVMe I/O**
- Write: 0.4 GB/s
- Read: N/A
Takeaways

• Mind the target audience.
• Sometimes the optimization is not obvious.
• No transpositions (use memory views instead).
• Try to hide optimizations behind abstractions.
• Allen Telescope Array is open for visitors!
Thanks for listening!
https://github.com/luigifcruz/blade

Contact me!
https://luigi.ltd/contact/

Comments or questions?

art by jgs