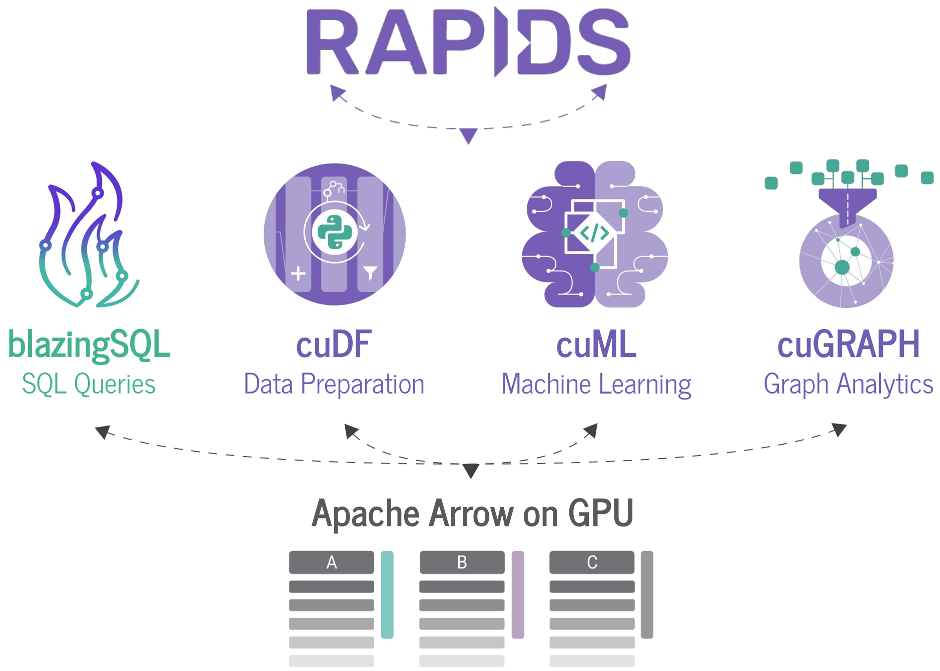
BlazingSQL with UCX a long and winding road


RAPIDS
blazingSQL
SQL Queries
cuDF
Data Preparation
cuML
Machine Learning
cuGRAPH
Graph Analytics
Apache Arrow on GPU
A
B
C

BLAZINGSQL

SQL on DataFrames

BlazingSQL runs SQL queries on cuDF dataframes. Now you can transform the same dataframe with SQL and pandas-like (cuDF) operations at scale.

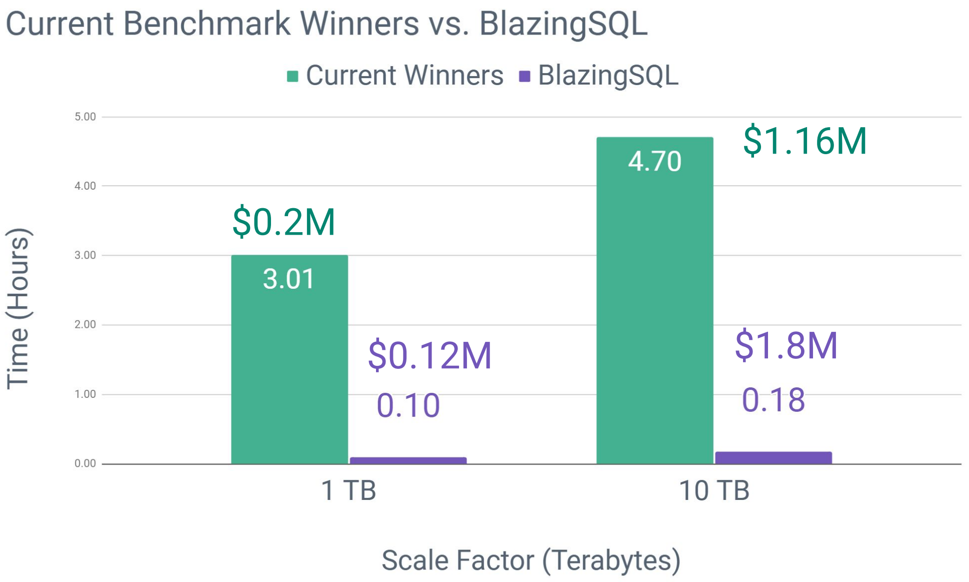
```
df = cudf.read_csv('../data/sample_taxi.csv')
bc.create_table('taxi', df)
```

```
bc.sql('select count(*) from taxi')
```

	COUNT(*)
0	743660

blazingSQL

TPCx-BB



BlazingSQL running TPCx-BB big data benchmark on 30 use cases.
https://medium.com/rapids-ai/relentlessly-improving-performance-d1f7d923ef90

UCX Prequel

- Communication in BlazingSQL a year ago was not very isolated
- Implemented Communication layer in UCP
  - couldn't get IPC to work
  - performance wasn't there for us
- Reworked it all using UCT
  - got IPC to work
    - Still seemed to be making copies though we verified it was using IPC calls
  - performance wasn't there for us
  - lots of bugs due to complex communication code that was mixed up with all the execution logic (due to blazingsql)
  - no ecosystem partners were using UCT directly so we couldn't get much help
- Not enough resources and time to figure this out and we wanted to wait and see if Nvidia's work could help us do this later.

Relational Algebra

SQL
```
select o_custkey, sum(o_totalprice) from orders where o_orderkey < 10 group by o_custkey
```

Calcite Relational Algebra
```
LogicalProject(o_custkey=[$0], EXPR$1=[CASE(=($2, 0), null:DOUBLE, $1)])
  LogicalAggregate(group=[{0}], EXPR$1=[$SUM0($1)], agg#1=[COUNT($1)])
    LogicalProject(o_custkey=[$1], o_totalprice=[$2])
        BindableTableScan(table=[[main, orders]], filters=[[<($0, 10)]], projects=[[0, 1, 3]], aliases=[[$f0,
  o_custkey, o_totalprice]])
```

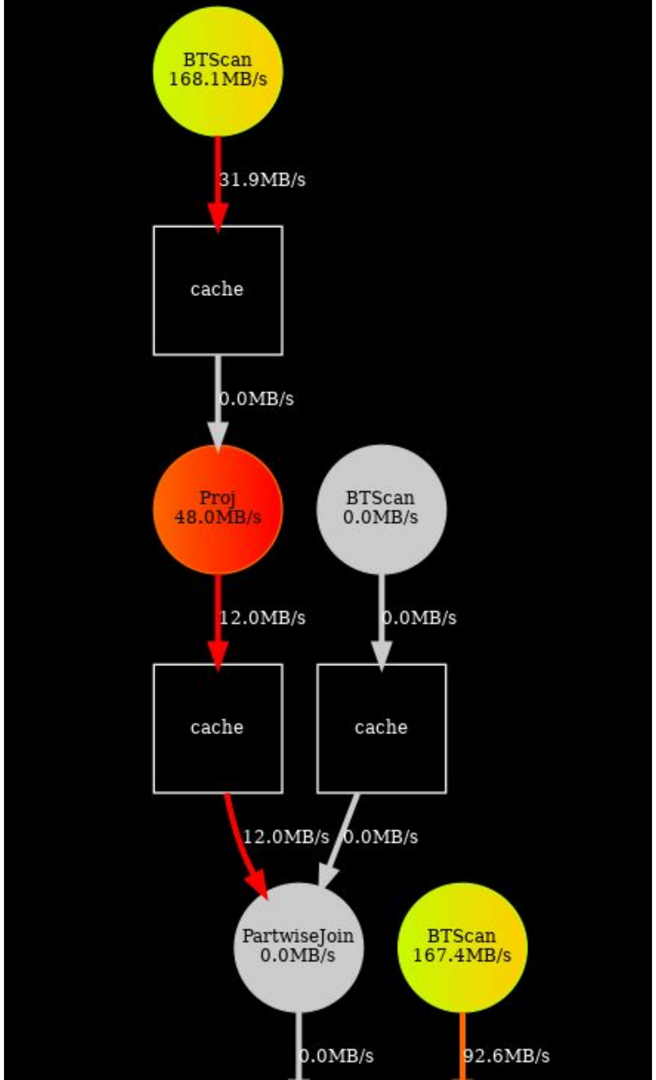
Blazing Physical Plan
```
LogicalProject(o_custkey=[$0], EXPR$1=[CASE(=($2, 0), null:DOUBLE, $1)])
  MergeAggregate(group=[{0}], EXPR$1=[$SUM0($1)], agg#1=[COUNT($1)])
    DistributeAggregate(group=[{0}], EXPR$1=[$SUM0($1)], agg#1=[COUNT($1)])  <-- Shuffles Data
      ComputeAggregate(group=[{0}], EXPR$1=[$SUM0($1)], agg#1=[COUNT($1)])
        LogicalProject(o_custkey=[$1], o_totalprice=[$2])
            BindableTableScan(table=[[main, orders]], filters=[[<($0, 10)]], projects=[[0, 1, 3]],
  aliases=[[$f0, o_custkey, o_totalprice]])
```

How Blazing is Distributed

- 90% of blazingsql is written in C/C++
- Executes within dask process
  - Creating a BlazingContext with a dask client calls c++ apis that start up Context for registering filesystems, creating execution graphs, executing those graphs on the workers
  - python .sql function calls two main blazing c++ apis
    - generate_graph
    - execute_graph
- All communication between workers once graph is executed is directly between the workers in C/C++
  - First attempt actually did this in python but this did not work out well for us

Blazing Execution Graph

- Homogenous
- Distributed
- Parallel
- Configurable



Kernels

- Inputs are CUDFs
  - Scans are only exception
- Outputs are CUDFS
- Perform computation a batch at a time
- May or may not required coordination / shuffling of data to other workers
- Very simple api to implement
  - implements a run function that can get inputs from a CacheMachine and schedules tasks
  - has a do_process function that is called when it receives a batch

Distributing Kernels

- Certain kernels are distributing kernels that can call functions like
  - send
    - one partition to one node
  - scatter
    - send a list of partitions to n nodes
  - broadcast
    - send all partitions to every node
- They are ignorant of HOW the message will be sent
  - don't care if its TCP or UCX

CacheData

- Usually Created from a CUDF
- TableScans generate these from a file handle + parser
- Holds a representation of a dataframe
  - GPU - CUDF
  - CPU - Basically A CUDF in CPU
  - Disk - temp Orc File
  - IOSource - Combines a file source (local, s3, hdfs, etc.) + Parser (parquet, orc, csv, etc.)

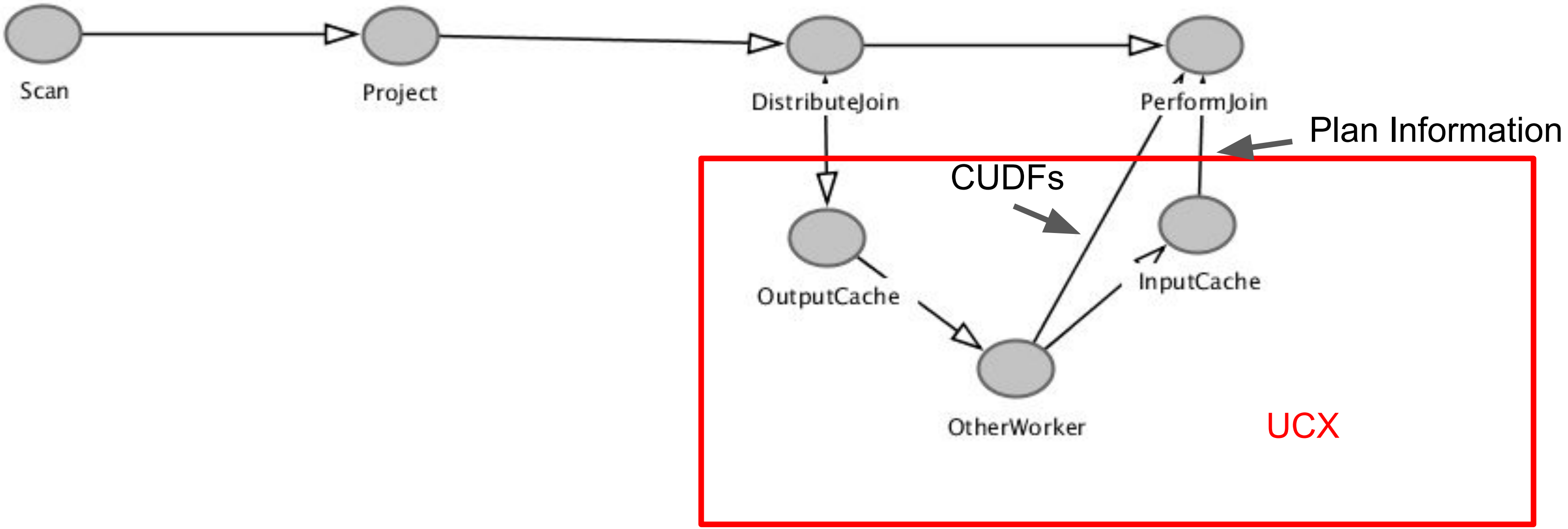
CacheMachine

- All kernels output to CacheMachines
  - Upon adding data to a CacheMachine it decides if it should go to CPU, Disk, or stay in GPU
- All kernels receive inputs from CacheMachines
- Tracks batches / rows processed
- Pulling from a cache returns a unique pointer to a CUDF
  - call waits with condition variable until data is available
  - returns a nullptr if no more data will be received

Execution Graph

Circles are Kernels

Arrows are CacheMachines



Message

- Metadata
    - Origin
    - Destinations
        - Other nodes that will receive this message
        - Can be a specific cache machine to feed a kernel
        - Can be in the generic input_cache when it is to be retrieved by specific business logic
            - mostly used for sending around plans / estimates
    -
- DataFrame
    - CUDF
        - Many GPU Buffers
        - Converted into a format more convenient for transport
            - no copies or allocations on gpu are performed for this

Attempt 1 UCX-PY

Use UCX-PY to start up endpoints to send and receive mesages

```
async def start_listener(self):

    async def handle_comm(comm):
        msg = await comm.read()
        msg = BlazingMessage(**{k: v.deserialize()
                                for k, v in msg.items()})
        self.received += 1
        await self.callback(msg)

    ip, port = parse_host_port(get_worker().address)
    self._listener = await UCXListener(ip, handle_comm)
    await self._listener.start()
    return "ucx://%s:%s" % (ip, self.listener_port())
```

Receives message and invokes callback that routes message to cache

Starts UCX endpoint and begins listening for messages

Attempt 1 UCX-PY

- UCX-PY receives messages and routes to specific cache for a kernel or to general input cache

```
async def route_message(msg):
    worker = get_worker()
    if msg.metadata["add_to_specific_cache"] == "true":
        graph = worker.query_graphs[int(msg.metadata["query_id"])]
        cache = graph.get_kernel_output_cache(
            int(msg.metadata["kernel_id"]),
            msg.metadata["cache_id"]
        )
        cache.add_to_cache(msg.data)
    else:
        cache = worker.input_cache
    cache.add_to_cache_with_meta(msg.data, msg.metadata)
```

Attempt 1 UCX-PY

- Expose input and output caches from each worker to python via cython.

```
cdef cppclass CacheMachine:
    void addCacheData(unique_ptr[CacheData] cache_data, const string & message_id, bool always_add )
    void addToCache(unique_ptr[BlazingTable] table, const string & message_id , bool always_add) nogil except+
    unique_ptr[CacheData] pullCacheData() nogil
    unique_ptr[CacheData] pullCacheData(string message_id) nogil
    bool has_next_now()
```

- Lots of python code to convert CacheData's to CUDFs
    - https://github.com/felipeblazing/blazingsql/blob/feature/adding-caches-for-ucx/engine/bsql_engine/io/io.pyx

Attempt 1 UCX-PY

- Create polling plugin to pull messages from output cache and send over wire

```
async_run_polling(self):
    import asyncio, os
    if self._worker.output_cache.has_next_now():
        df, metadata = self._worker.output_cache.pull_from_cache()
        await UCX.get().send(BlazingMessage(metadata, df))
        await asyncio.sleep(0)
```

Attempt 1 UCX-PY

Problems

- Python is single threaded
    - only one message sent at a time
- UCX-PY is meant for single threaded applications
- Python overhead was a problem situations where low latency was important for performance.
- There were alot of iterations and "flavors" we tried here
- Debugging was really tough and race conditions that would occur every few thousand so messages plagued us

Attempt 2 - UCX-PY endpoint UCX C api send / recv

- Use UCX-PY to set up end points as before
- using ucp_tag_send/recv_nb apis in C++ classes
- define callbacks in C
- Store state that was globally accessible for handling callbacks
- Could now right unit tests in C
  - Easier debugging

Communication Interfaces

MessageSender

- Polls for outgoing messages and sends them

MessageListener

- Polls for incoming messages and routes them

MessageReceiver

- Constructs a message from buffers that transport layer provides

Message Sender

```
class message_sender {
public:
        message_sender(...);
        std::shared_ptr<ral::cache::CacheMachine> get_output_cache();
        void run_polling();
private:
        ctpl::thread_pool<BlazingThread> pool;
        std::shared_ptr<ral::cache::CacheMachine> output_cache;
        std::map<std::string, node> node_address_map;           <-- map from nodes' names to their ucp endpoint
        blazing_protocol protocol;                              <-- Specifies UCX / TCP
        ucp_worker_h origin
        size_t request_size;
        int ral_id;
};
```

Message Listener

```
class ucx_message_listener : public message_listener {
public:
   void poll_begin_message_tag(bool running_from_unit_test);
   void add_receiver(ucp_tag_t tag, std::shared_ptr<message_receiver> receiver);
   std::shared_ptr<message_receiver> get_receiver(ucp_tag_t tag);
   void remove_receiver(ucp_tag_t tag);
   ucp_worker_h get_worker();
   void start_polling() override;
private:
   ucx_message_listener(ucp_context_h context, ucp_worker_h worker, const std::map<std::string, comm::node>& nodes, int
num_threads);
   void poll_message_tag(ucp_tag_t tag, ucp_tag_t mask);
   size_t _request_size;
   ucp_worker_h ucp_worker;
   std::map<ucp_tag_t,std::shared_ptr<message_receiver> > tag_to_receiver;
};
```

Message Receiver

```
class message_receiver {
public:
 message_receiver(...);
 void set_buffer_size(uint16_t index,size_t size);
 void confirm_transmission();
 void * get_buffer(uint16_t index);
 bool is_finished();
 void finish();
private:
 std::vector<ColumnTransport> _column_transports;
 std::shared_ptr<ral::cache::CacheMachine> _output_cache;
 ral::cache::MetadataDictionary _metadata;
 bool _include_metadata;
 std::vector<rmm::device_buffer> _raw_buffers;   <--
 int _buffer_counter = 0;
};
```

ucx_buffer_transport

```
class ucx_buffer_transport : public buffer_transport {
public:
   ucx_buffer_transport(...);
   ~ucx_buffer_transport();
   void send_begin_transmission() override;
protected:
   void send_impl(const char * buffer, size_t buffer_size) override;
private:
   ucp_worker_h origin_node;
   int ral_id;
   ucp_tag_t generate_message_tag();
   ucp_tag_t tag;  /**< The first 6 bytes are the actual tag the last two
                        indicate which frame this is. */
   int message_id;
   size_t _request_size;
};
```

Using Transport

```
std::shared_ptr<buffer_transport> transport;
if(blazing_protocol::ucx == protocol){
	transport = std::make_shared<ucx_buffer_transport>(...);
}else if (blazing_protocol::tcp == protocol){
	transport = std::make_shared<tcp_buffer_transport>(...);
}else{
	throw std::runtime_error("Unknown protocol");
}
transport->send_begin_transmission();
transport->wait_for_begin_transmission(); <-- Now a NO-OP but it wasn't at this point in time
for(size_t i = 0; i < raw_buffers.size(); i++) {
	transport->send(raw_buffers[i], buffer_sizes[i]);
}
transport->wait_until_complete(); <-- So we can limit number of simultaneous transmissions
```

Send Begin Transmission

```
void ucx_buffer_transport::send_begin_transmission() {
	std::vector<char> buffer_to_send = detail::serialize_metadata_and_transports(metadata, column_transports);
	std::vector<ucs_status_ptr_t> requests;
	for(auto const & node : destinations) {
		auto request = ucp_tag_send_nb(
		node.get_ucp_endpoint(), buffer_to_send.data(), buffer_to_send.size(), ucp_dt_make_contig(1), tag,
		send_begin_callback_c);
		if(UCS_PTR_IS_ERR(request)) {
			throw std::runtime_error("Error sending begin transmission");
		} else if(UCS_PTR_STATUS(request) == UCS_OK) {
			recv_begin_transmission_ack();
		} else {
			auto blazing_request = reinterpret_cast<ucx_request *>(&request);
			blazing_request->uid = reinterpret_cast<blazing_ucp_tag *>(&tag)->message_id;
			requests.push_back(request);
		}
	}
	wait_for_begin_transmission(); <-- Calls ucp_progress and waits for completion
}
```

Begin Callback

```
void send_begin_callback_c(void * request, ucs_status_t status) {
	auto blazing_request = reinterpret_cast<ucx_request *>(request);
	auto transport = message_uid_to_buffer_transport[blazing_request->uid];
	transport->recv_begin_transmission_ack();
	ucp_request_release(request);
}
```

Problems with Approach

- Blazing is C++ aside from GPU kernels
  - Couldn't pass member functions as callbacks
    - complicated state management
    - Resource management became more complicated since many of our resources are shared
- Lots of race conditions and error prone code
  - The callbacks made debugging a bit more complicated since it was hard to assess where the issue began with many threads sending messages simultaneously
- UCX-PY endpoints were causing issues

Changes

- Use nbr apis so we don't have to invoke callbacks
- Create endpoints in C
  - Share Context with UCX-PY

C endpoints + ucp_tag_send_nbr

- Liked the idea of not having to use callbacks and just checking on request status
- We could manage resources inside of the constructs we had built

```
auto status = ucp_tag_send_nbr(node.get_ucp_endpoint(), buffer_to_send.data(), buffer_to_send.size(),
                               ucp_dt_make_contig(1), tag, requests[i] + 32);

if (status == UCS_INPROGRESS) {
      do {
            ucp_worker_progress(origin_node);
            status = ucp_request_check_status(&requests[i] + sizeof(ucx_request));
      } while (status == UCS_INPROGRESS);
}
if(status != UCS_OK){
      throw std::runtime_error("Was not able to send begin transmission to " + node.id());
}
```

WOOPS!!!

C endpoints + ucp_tag_send_nbr

- Go figure tons of threads spinning calling the exact same apis like ucp_progress was slow and a huge resource hog
  - we were so sleep deprived at the time
- Spent lots of time trying to understand the reasoning behind certain details like the request to nbr being defined as
  - "Request handle allocated by the user. There should be at least UCP request size bytes of available space before the *req*. The size of UCP request can be obtained by **ucp_context_query** function."

Removing Acknowledgements

- Acknowledgements for initializing messages were pointless
  - UCX is smart and will hold onto messages that no one has tried to read yet
    - We had assumed we needed to be probing for a tag before the message was sent
- Acknowledgements added lots of latency
- Greatly improved performance (2x)

UCP Progress Manager

- Checks status of sent messages and runs progress on a single thread
- Lets Kernels know when message is completed sending so we can hold onto resources until they are no longer needed
- Greatly reduced CPU pressure
- Improved performance

UCP Progress Manager

```
class ucp_progress_manager{
public:
   static ucp_progress_manager * get_instance();
   void add_recv_request(char * request, std::function<void()> callback);
   void add_send_request(char * request, std::function<void()> callback);
private:
  struct request_struct{
       char * request;
       std::function<void()> callback;     <-- only needs to be invoked messages which begin a cudf transfer
       bool operator <(const request_struct & other);
   };
   ucp_progress_manager(ucp_worker_h ucp_worker,size_t request_size);
   std::set<request_struct> send_requests;
   std::set<request_struct> recv_requests;
   ucp_worker_h ucp_worker;
   void check_progress();   <-- Checks on progress of requests
};
```

check_progress()

```
ucp_worker_progress(ucp_worker);

 for(const auto & req_struct : cur_send_requests){
     auto status = ucp_request_check_status(req_struct.request + _request_size);
     if (status == UCS_OK){
         {
             std::lock_guard<std::mutex> lock(request_mutex);
             this->send_requests.erase(req_struct);
         }
         delete req_struct.request;
         req_struct.callback();
     } else if (status != UCS_INPROGRESS){
         throw std::runtime_error("Communication error in check_progress for send_requests.");
     }
 }
```

Current State

- IPC fails in UCX if we don't call cudaSetDevice to initialize a cudacontext before any messages are sent
  - this fix leads to other race conditions that we can't replicate with our TCP implementation
  - It is still slower than TCP in this case
- Works without IPC but is much slower than TCP
- Blazing sends many more (smaller) messages than other use cases that our friends at Nvidia have seen
  - many times its just metadata for coordination between the nodes without a big payload

Optimizations

- Packing small messages with many buffers into one buffer
- Pool for requests
- Investigating ipc failures on certain hardware configurations

Thanks!

- Thanks to Peter Entschev, Matt Baker, Corey Nolet for all of their patience and help in implementing and testing.
- Thanks to Oscar Hernandez and Jens Glaser for supporting our work on UCX.