RAPIDS CUDA DataFrame Internals for C++ Developers - S91043

Jake Hemstad - NVIDIA - Developer Technology Engineer

GTC2019 | 03/20/19

What is RAPIDS cuDF?

Open-Source CUDA DataFrame

GPU-accelerated DataFrames

Data science operations: filter, sort, join, groupby,…

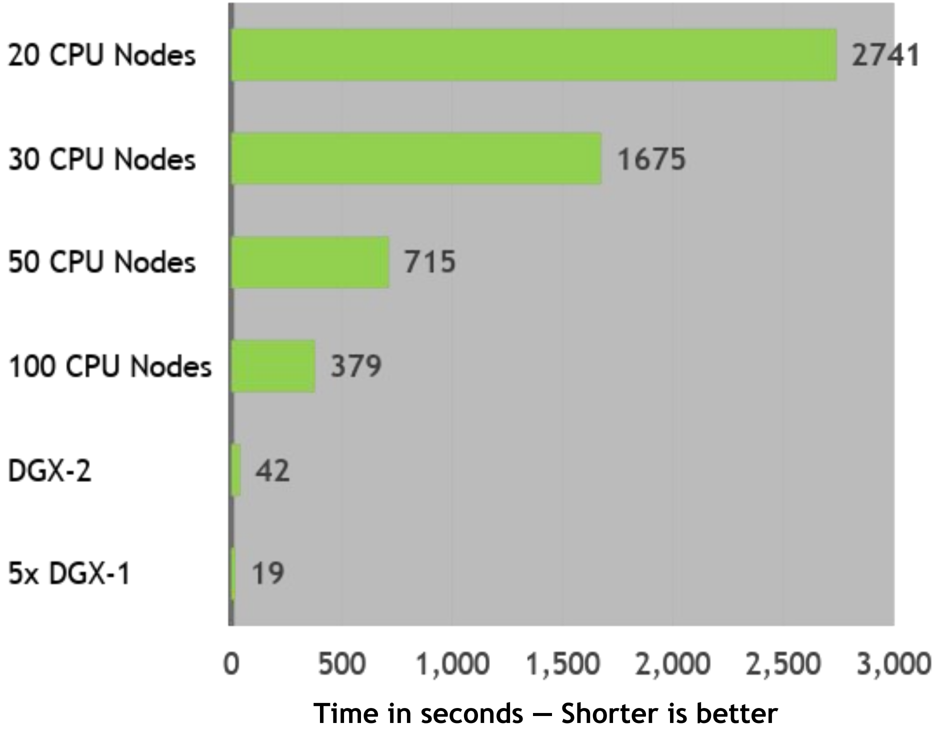
High-level, Python productivity (Pandas-like)

Bare-metal, CUDA/C++ performance

RAPIDS

github.com/rapidsai/cudf
rapids.ai

cuDF
File Read and Data Preparation



200GB CSV dataset; Data preparation includes joins, variable transformations. 5x DGX-1 on InfiniBand network. CPU nodes: 61 GiB of memory, 8 vCPUs, 64-bit platform, Apache Spark

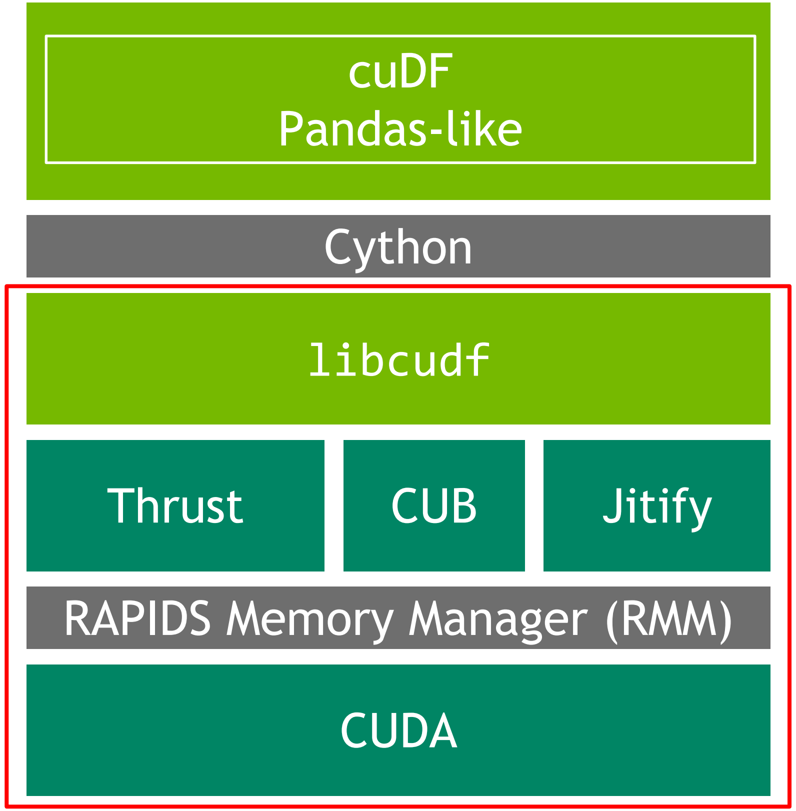
NVIDIA

`libcudf`

Who This Talk is For

You want to learn about:

- `libcudf: cuDF's` underlying C++14 library
- How to use `libcudf` in your applications
- CUDA-accelerated data science algorithms
- How to contribute to `libcudf`



CUDA DataFrame

What is a DataFrame?

Think spreadsheet

Equal length columns of different types

How to store in memory?

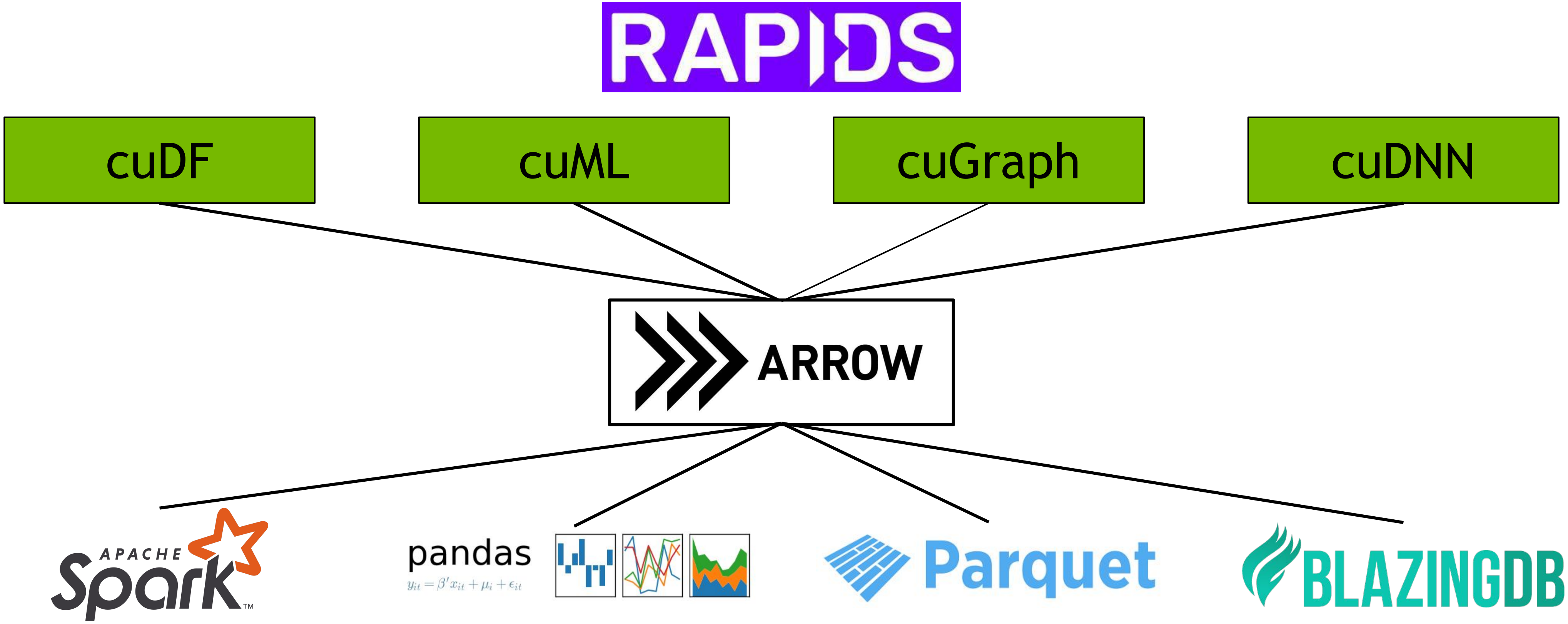
- cuDF uses Apache Arrow[1]
- Contiguous, column-major data representation

Mortgage ID	Pay Date	Amount($)
101	12/18/2018	1029.30
102	12/21/2018	1429.31
103	12/14/2018	1289.27
101	01/15/2018	1104.59
102	01/17/2018	1457.15
103	NULL	NULL

[1] https://arrow.apache.org/docs/memory_layout.html

Apache Arrow Memory Format

Enabling Interoperability



Column Representation

libcudf is column-centric

All operations defined in terms of operations on columns

Type-erased data (void*) allows interoperability with other languages and type systems

Arrow enables efficient, shallow copy data sharing across frameworks/languages

```
struct column {
  void* data;      // contiguous buffer
  int size;        // number of elements
  DType type;      // type indicator
  uint32_t* mask; // null bitmask
};

enum DType {
  INT,    // int value
  FLOAT, // float value
  DATE    // int64_t ms since epoch
  ...
};
```

Null Bitmask

How To Represent Missing Data

Any element can be `NULL` –> undefined

Different from `NaN` –> defined invalid

`NULL` values are represented in bitmask

1-bit per element

- 0 == NULL
- 1 == not NULL

Least-significant bit ordering

```
values =  [0, 1, null, NaN, null, 3]

bitmask = [0  0  1  0  1  0  1  1] = 0x2B
```

Column Example

Apache Arrow Memory Layout

Mortgage ID	Pay Date	Amount
101	12/18/2018	1029.30
102	12/21/2018	1429.31
103	12/14/2018	1289.27
101	01/15/2018	1104.59
102	01/17/2018	1457.15
103	NULL	NULL

Mortgage ID

```
data = [101, 102, 103, 101, 102, 103]
size = 6
type = INT
bitmask = [0x3F] = [0 0 1 1 1 1 1 1]
```

Note LSB order

Pay Date

```
data = [1545091200000, 1545350400000, 1544745600000,
        1514764800000, 1516147200000, *garbage* ]
size = 6
type = DATE
bitmask = [0x1F] = [0 0 0 1 1 1 1 1]
```

Amount

```
data = [1029.30, 1429.31, 1289.27,
        1104.59, 1457.15, *garbage*]
size = 6
type = FLOAT
bitmask = [0x1F] = [0 0 0 1 1 1 1 1]
```

libcudf Operations

All functions act on one or more columns

Operations include:

- Sort
- Join
- Groupby
- Filtering
- Transpose
- etc.

Input columns are generally immutable

```
void some_function( cudf::column const* input,
                    cudf::column * output,
                    args...)
{
    // Do something with input
    // Produce output
}
```

Example Operation

Sort

`in->data` is *type-erased*

1. Deduce `T` from enum `in->type`
2. Call function template with `T`
3. Cast `in->data` to `T*`
4. Perform `thrust::sort` with `typed_data`

Common pattern in libcudf

Problem: Duplicated `switch`es are difficult to maintain and error-prone

```
void sort(cudf::column * in){
  switch(in->type){
       case INT:
          typed_sort<int32_t>(in); break;
       case FLOAT:
          typed_sort<float>(in); break;
       case DATE:
          typed_sort<int64_t>(in); break;
       ...
  }
}

template <typename T>
void typed_sort(cudf::column * in){
   T* typed_data{ static_cast<T*>(in->data) };
   thrust::sort(thrust::device,
                typed_data, typed_data + in->size);
}
```

Type Dispatching

libcudf's Solution

Centralize and abstract the `switch`

`type_dispatcher`

1. Maps `type` enum to `T`
2. Invokes functor `F<T>()`

```
template <typename Functor>
auto type_dispatcher(DType type,
                     Functor F)
{
  switch(type){
      case INT:   return F<int32_t>();
      case FLOAT: return F<float>();
      case DATE:  return F<int64_t>();
      ...
  }
}
```

Note: The syntax `F<T>()` is abbreviated for clarity.
The correct syntax is `F::template operator()<T>()`.

libcudf's type dispatcher also supports functors with arguments

Type Dispatching

Sort Revisited

Define a functor F with operator() template

type_dispatcher maps type to T and invokes F<T>()

sort_functor casts with T

Perform thrust::sort on typed_data

sort.cu

```
#include <type_dispatcher.cuh>

sort_functor{
   cudf::column _col;
   sort_functor(cudf_column col ) : _col{col} {}

   template <typename T>
   void operator()(){
      T* typed_data = static_cast<T*>(_col->data);
      thrust::sort(typed_data,
                   typed_data + _col->size);
   }
};

void sort(cudf::column * col){
   type_dispatcher(col->type, sort_functor{ *col });
}
```

Type Dispatching

Combinatorial Type Explosion

Binary operations between two columns are common (e.g., sum, minus, div, etc.)

```
out = lhs op rhs
```

Independent types

11+ types, 14+ ops

Problem:

- 11^3 x 14 = ~18,600 instantiations
- 1+ hour to compile just binary operations

```
void binary_op(cudf::column* out, cudf::column* lhs,
               cudf::column* rhs, Op op)
{
   // out, lhs, rhs types are all independent
   // Need to instantiate code for all combinations
   // Repeat for every `op`
}
```

Solution: JIT compilation with Jitify

Simplify CUDA Run-time Compilation

Compiles specialized kernel string at run time

Compiled kernel is cached for reuse

`libcudf` uses Jitify for binary operations

- ~300ms overhead to compile new kernel
- ~150ms to reuse kernel w/ new types
- Trivial overhead to reuse from cache

https://github.com/NVIDIA/jitify

```
const char* program_source = "my_program\n"
    "template<int N, typename T>\n"
    "__global__\n"
    "void my_kernel(T* data) {\n"
    "    T data0 = data[0];\n"
    "    for( int i=0; i<N-1; ++i ) {\n"
    "        data[0] *= data0;\n"
    "    }\n"
    "}\n";
static jitify::JitCache kernel_cache;
jitify::Program program = kernel_cache.program(program_source);

dim3 grid(1); dim3 block(1);
using jitify::reflection::type_of;
program.kernel("my_kernel")
       .instantiate(3, type_of(*data)) // Instantiates template
       .configure(grid, block)
       .launch(data);
```


NVIDIA.

Recap

libcudf so far...

- Apache Arrow memory layout
- Column-centric operations
- Type-erased data
- `type_dispatcher` to reconstruct type
- Runtime compilation w/ Jitify

Many operations require temporary memory allocations

Most cuDF ops not performed in place:
many column allocations/deallocations

```
sort_functor{
   cudf::column _col;
   sort_functor(cudf_column col ) : _col{col} {}

   template <typename T>
   void operator()(){
      T* typed_data = static_cast<T*>(_col->data);

      // Allocates temporary memory!
      thrust::sort(thrust::device,
                   typed_data, typed_data + _col->size);
   }
};

void sort(cudf::column * col){
   type_dispatcher(col->type, sort_functor{ *col });
}
```

Memory Management

Memory Management Overhead

Example: cuDF Mortgage Workflow

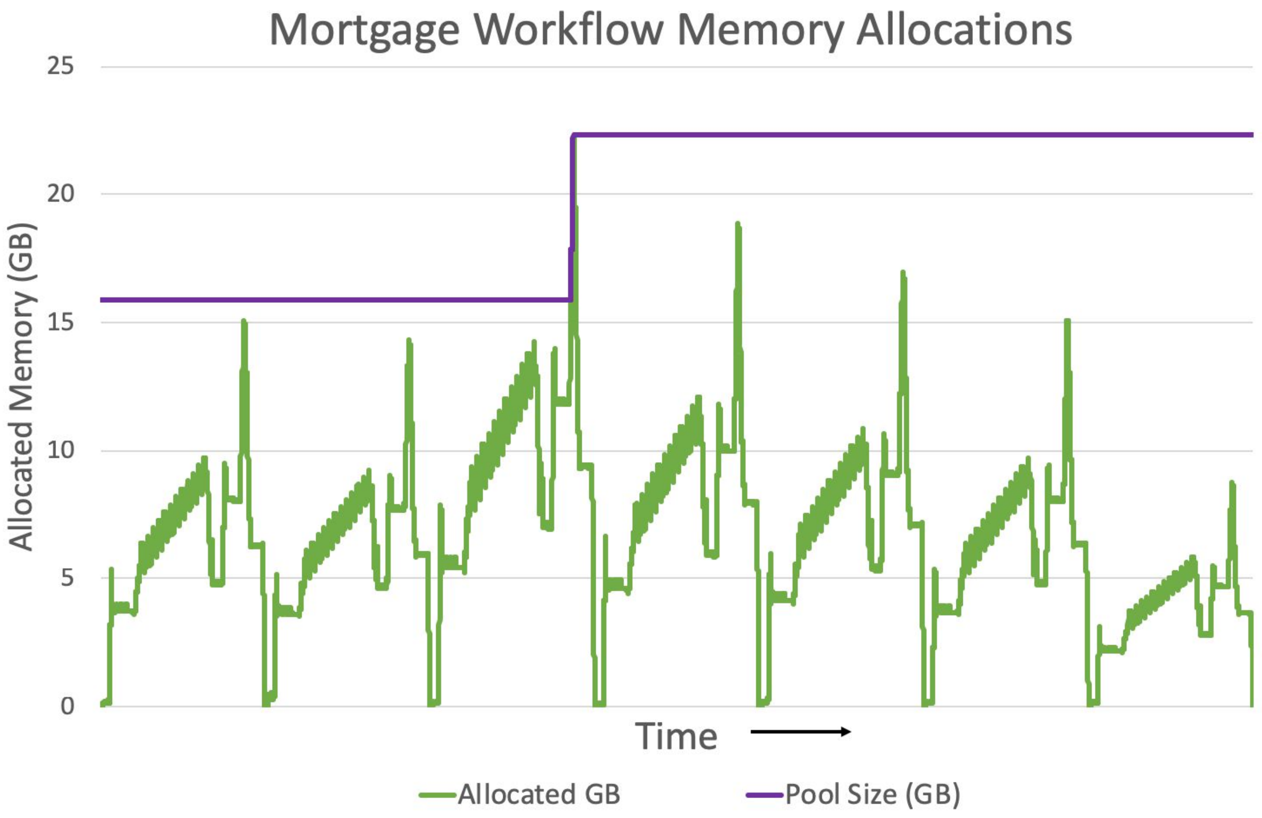
Data cleanup and feature engineering

1. Read CSV files into DataFrames
2. Joins, groupbys, unary/binary ops
3. Create DMatrix for XGBoost

cuDF ops are not in-place

=> frequent malloc/free

88% of cuDF time spent in CUDA memory management!



CUDA Memory Allocation

cudaMalloc / cudaFree: Why are they expensive?

Synchronous: blocks the device

`cudaFree` scrubs memory for security

Peer Access: GPU-to-GPU direct memory access

- cudaMalloc creates peer mappings
- Scales O(#GPUs^2)

```
cudaMalloc(&buffer, size_in_bytes);

cudaFree(buffer);
```

RMM Memory Pool Allocation

https://github.com/rapidsai/rmm

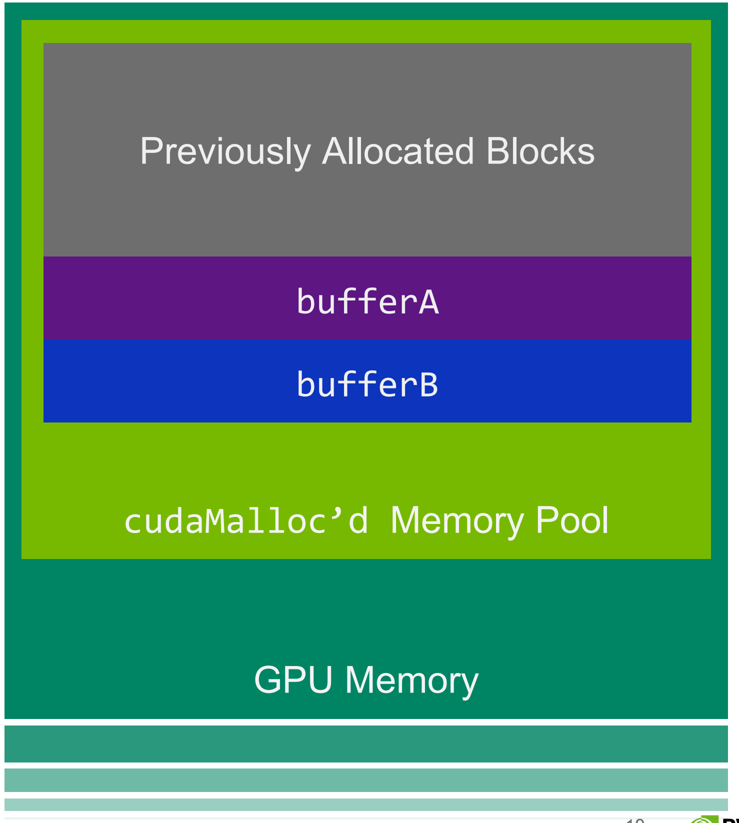
Use large `cudaMalloc` allocation as memory pool

Custom memory management in pool

Streams enable asynchronous malloc/free

RMM currently uses CNMem as it's Sub-allocator
https://github.com/NVIDIA/cnmem

RMM is standalone and free to use in your own projects!



RAPIDS Memory Manager (RMM)

Drop-in Allocation Replacement

```
RMM_ALLOC(&buffer, size_in_bytes, stream_id);

RMM_FREE(buffer, stream_id);
```

Asynchronous

```
rmm::device_vector<int> dvec(size);

thrust::sort(rmm::exec_policy(stream)->on(stream), …);
```

```
dev_ones = rmm.device_array(np.ones(count))
dev_twos = rmm.device_array_like(dev_ones)
# also rmm.to_device(), rmm.auto_device(), etc.
```

NVIDIA CUDA

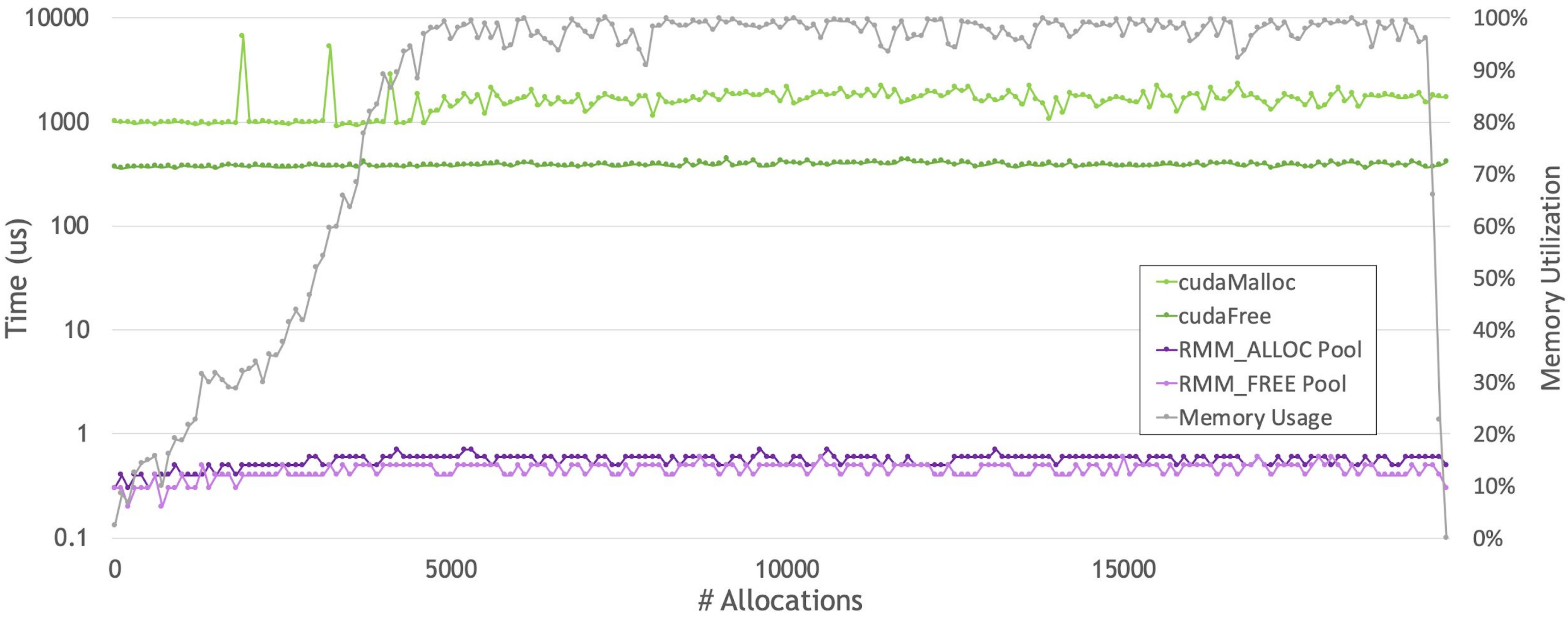
Thrust

python™

Numba

RMM Raw Performance

1000x faster than cudaMalloc/cudaFree (microbenchmark)



RMM: 10x Performance on RAPIDS

Mortgage Workflow on 16x V100 GPUs of DGX-2

	Time spent in malloc/free	Total ETL Time	% Time
cudaMalloc / cudaFree (no pool)	486s	550s	88.3%
rmmAlloc / rmmFree (pool)	0.088s	55s	0.16%

10x

cudaMalloc/cudaFree overhead gets worse with more GPUs

- RMM is valuable even on Single-GPU runs, where the fraction is “only” 14-15%

RMM benefit is combination of low-overhead suballocation and reduced synchronization

Deep Dive

CUDA-Accelerated GroupBy

Deep Dive

Common data science operation

Group unique keys and aggregate associated values —> reduce by key

Answers questions like:

“What’s the avg payment for each mortgage?”

“Which mortgages are delinquent?”

“Which mortgages are paid off early?”

Mortgage ID	Pay Date	Amount	Avg
101	12/18/2018	1029.30	1066.95
101	01/15/2018	1104.59	
102	12/21/2018	1429.31	1443.23
102	01/17/2018	1457.15	
103	12/14/2018	1289.27	1289.27
103	NULL	NULL	

Hash-Based GroupBy

Mortgage ID	Amount
101	1029.30
102	1429.31
103	1289.27
101	1104.59
102	1457.15
103	NULL

Idx	Key	{count, sum}
0	E	E
1	E	E
2	E	E
3	E	E
4	E	E
5	E	E
6	E	E
7	E	E

Hash-Based GroupBy

hash(101) == 4

Mortgage ID	Amount
101	1029.30
102	1429.31
103	1289.27
101	1104.59
102	1457.15
103	NULL

Idx	Key	{count, sum}
0	E	E
1	E	E
2	E	E
3	E	E
4	101	{1, 1029.30}
5	E	E
6	E	E
7	E	E

Hash-Based GroupBy

hash(102) == 1

Mortgage ID	Amount
101	1029.30
102	1429.31
103	1289.27
101	1104.59
102	1457.15
103	NULL

Idx	Key	{count, sum}
0	E	E
1	102	{1, 1429.31}
2	E	E
3	E	E
4	101	{1, 1029.30}
5	E	E
6	E	E
7	E	E

Hash-Based GroupBy

hash(103) == 4

Mortgage ID	Amount
101	1029.30
102	1429.31
103	1289.27
101	1104.59
102	1457.15
103	NULL

Idx	Key	{count, sum}
0	E	E
1	102	{1, 1429.31}
2	E	E
3	E	E
4	101	{1, 1029.30}
5	E	E
6	E	E
7	E	E

Hash-Based GroupBy

Mortgage ID	Amount
101	1029.30
102	1429.31
103	1289.27
101	1104.59
102	1457.15
103	NULL

hash(103) == 4

103 =? 101

Idx	Key	{count, sum}
0	E	E
1	102	{1, 1429.31}
2	E	E
3	E	E
4	101	{1, 1029.30}
5	E	E
6	E	E
7	E	E

Hash-Based GroupBy

Mortgage ID	Amount
101	1029.30
102	1429.31
103	1289.27
101	1104.59
102	1457.15
103	NULL

hash(103) == 4

103 != 101
Collision!

Idx	Key	{count, sum}
0	E	E
1	102	{1, 1429.31}
2	E	E
3	E	E
4	101	{1, 1029.30}
5	E	E
6	E	E
7	E	E

Hash-Based GroupBy

Mortgage ID	Amount
101	1029.30
102	1429.31
103	1289.27
101	1104.59
102	1457.15
103	NULL

hash(103) == 4

Idx	Key	{count, sum}
0	E	E
1	102	{1, 1429.31}
2	E	E
3	E	E
4	101	{1, 1029.30}
5	103	{1, 1289.27}
6	E	E
7	E	E

Hash-Based GroupBy

Mortgage ID	Amount
101	1029.30
102	1429.31
103	1289.27
101	1104.59
102	1457.15
103	NULL

hash(101) == 4

Idx	Key	{count, sum}
0	E	E
1	102	{1, 1429.31}
2	E	E
3	E	E
4	101	{1, 1029.30}
5	103	{1, 1289.27}
6	E	E
7	E	E

Hash-Based GroupBy

Mortgage ID	Amount
101	1029.30
102	1429.31
103	1289.27
101	1104.59
102	1457.15
103	NULL

hash(101) == 4

101 =? 101

Idx	Key	{count, sum}
0	E	E
1	102	{1, 1429.31}
2	E	E
3	E	E
4	101	{1, 1029.30}
5	103	{1, 1289.27}
6	E	E
7	E	E

Hash-Based GroupBy

Mortgage ID	Amount
101	1029.30
102	1429.31
103	1289.27
101	1104.59
102	1457.15
103	NULL

hash(101) == 4

101 == 101

Idx	Key	{count, sum}
0	E	E
1	102	{1, 1429.31}
2	E	E
3	E	E
4	101	{2, 2133.89}
5	103	{1, 1289.27}
6	E	E
7	E	E

Hash-Based GroupBy

Mortgage ID	Amount
101	1029.30
102	1429.31
103	1289.27
101	1104.59
102	1457.15
103	NULL

hash(102) == 1

102 == 102

Idx	Key	{count, sum}
0	E	E
1	102	{2, 2886.46}
2	E	E
3	E	E
4	101	{2, 2133.89}
5	103	{1, 1289.27}
6	E	E
7	E	E

Hash-Based GroupBy

Mortgage ID	Amount
101	1029.30
102	1429.31
103	1289.27
101	1104.59
102	1457.15
103	NULL

hash(102) == 4

103 != 101

Idx	Key	{count, sum}
0	E	E
1	102	{2, 2886.46}
2	E	E
3	E	E
4	101	{2, 2133.89}
5	103	{1, 1289.27}
6	E	E
7	E	E

Hash-Based GroupBy

Mortgage ID	Amount
101	1029.30
102	1429.31
103	1289.27
101	1104.59
102	1457.15
103	NULL

hash(102) == 4

NULL value is ignored

Idx	Key	{count, sum}
0	E	E
1	102	{2, 2886.46}
2	E	E
3	E	E
4	101	{2, 2133.89}
5	103	{1, 1289.27}
6	E	E
7	E	E

Hash-Based GroupBy

Idx	Key	{count, sum}
0	E	E
1	102	{2, 2886.46}
2	E	E
3	E	E
4	101	{2, 2133.89}
5	103	{1, 1289.27}
6	E	E
7	E	E

Extract non-empty entries and perform (sum/count)

Mortgage ID	Avg Amount
102	1443.23
101	1066.95
103	1289.27

concurrent_unordered_map

Enabling Hash-based GroupBy

```
template<typename KeyT, typename PayloadT>
__device__ void insert(KeyT const& new_key, PayloadT new_value){
   uint32_t hash_value = hash_function(new_key);
   int index           = hash_value % hash_table_size;
   while (not insert_success) {
      // Attempt to update hash bucket
       KeyT old_key = atomicCAS(&hash_table[index].key, EMPTY, new_key);

      // If the bucket was empty, or already contains "new_key"
      // Then update the associated payload
       if ( (EMPTY == old_key) or
            (new_key == old_key ){
          // Update payload
          atomicAdd(&hash_table[index].count, 1);       // count++
          atomicAdd(&hash_table[index].sum, new_value); // sum += new_value
          insert_success = true;
       }
        // Insert failed, advance to next hash bucket
       index = (index + 1) % hash_table_size;
   }
}
```

Note: Code is simplified for clarity. Actual insert code accepts any generic binary operation(s) to be performed between the new and old payload. Likewise, handling of null values is omitted.

Wrapping Up

`libcudf` C++

How to Use `libcudf` in Your Applications

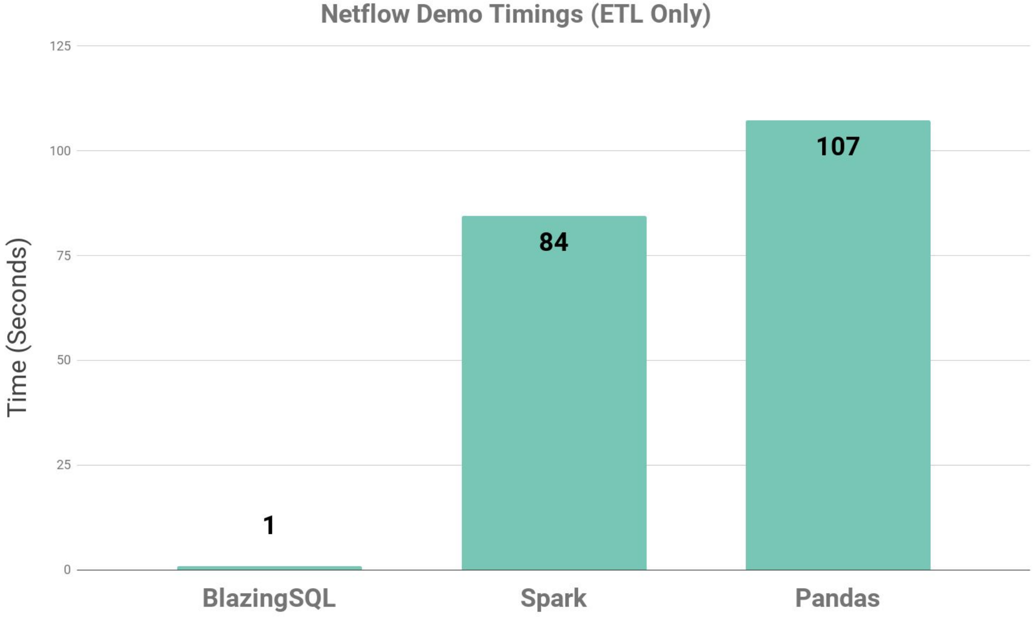
`libcudf` is not built for cuDF alone

Single-GPU primitives to enable building multi-GPU algorithms

`libcudf` C++ API is designed for reuse

Modular, reusable components

- `concurrent_unordered_map`
- Memory Manager (reusable sub-allocator)
- algorithms—join, groupby, etc.



Analyzes VAST NetFlow 5GB data set
BlazingSQL: 1xNVIDIA Tesla T4 16GB
Spark&Pandas: 4x 8 vCPU 32GB

Future Directions

What We Are Working On

- Overhaul of legacy C interface to modern C++
- Feature Completeness
  - Push functionality from Python into C++
- Coming Soon
  - Improved String support, rolling window functions, statistic operations
  - Generic variable-length datatypes
- Future language support
  - Spark Java bindings

Contribute to `libcudf`

Help Us Improve

`libcudf` is open source: Apache 2 license

Many interesting CUDA/C++ engineering and algorithmic problems to solve

Try it out! File an issue or submit a PR!

https://github.com/rapidsai/cudf

Contributors:

NVIDIA

BLAZINGDB

ANACONDA

QUANSIGHT

@WalmartLabs

Learn More at GTC

CUDA Accelerated Data Analytics

Talk with me and others about `libcudf` and accelerating Data Analytics on GPUs

CE9113 - Connect with the Experts: Data Analytics on GPU: Algorithms and Implementations

Tomorrow - 11:00 AM -12:00 PM – SJCC Hall 3 Pod D

Learn how BlazingDB uses `libcudf` to accelerate SQL queries

S9798 - BlazingSQL on RAPIDS: SQL for Apache Arrow in GPU Memory

William Malpica, Rodrigo Aramburu, Felipe Aramburu

Today - 3:00 PM - 03:50 PM – SJCC Room 212A

Learn about accelerating Join on multiple GPUs

S9557 - Effective, Scalable Multi-GPU Joins

Nikolay Sakharnykh, Jiri Kraus, Tim Kaldwey

Today - 4PM - SJCC Room 212A (Concourse Level)

Learn how Unified Memory can help for Data Analytics

S9726 - Unified Memory for Data Analytics and Deep Learning

Nikolay Sakharnykh, Chirayu Garg

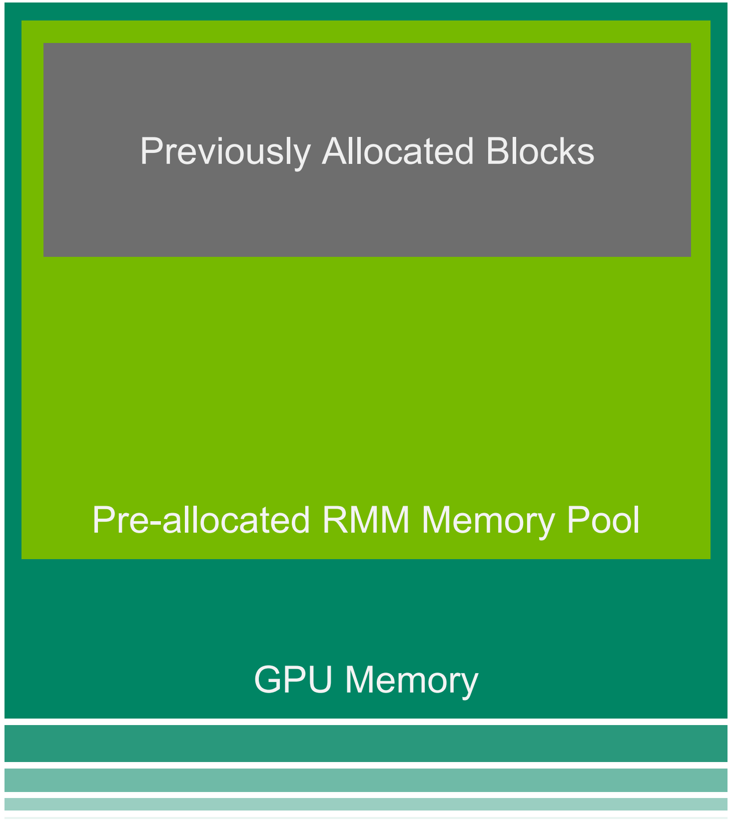
Tomorrow - 3:00 PM - 03:50 PM– SJCC Room 211A

S9793 - cuDF: RAPIDS GPU-Accelerated Data Frame Library (Python API) Keith Kraus (GTC on-demand)

NVIDIA

RMM

Pool Allocation Example

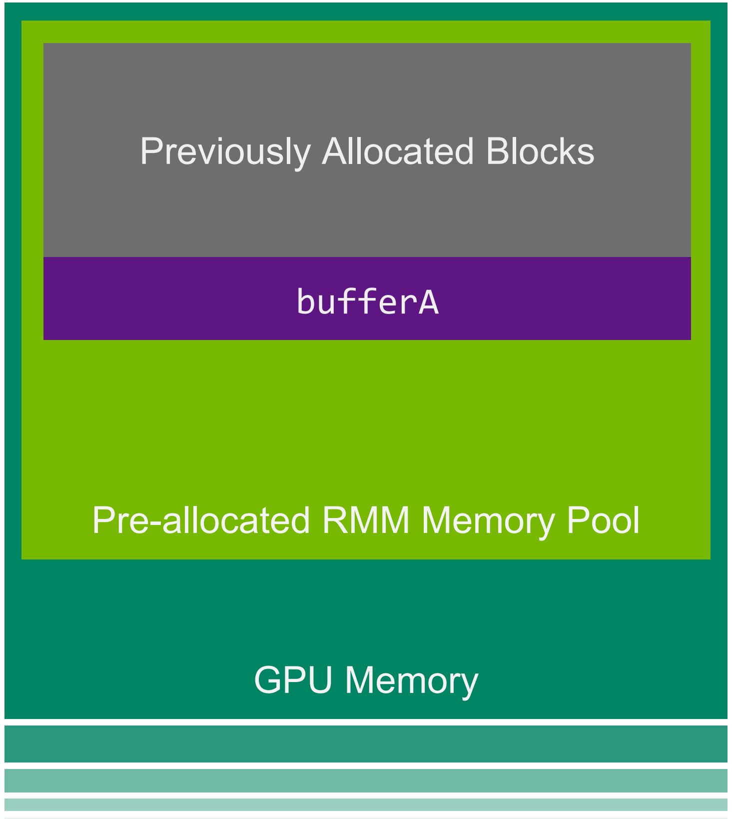



```
...

RMM_ALLOC(&bufferA, sizeA, streamA);
RMM_ALLOC(&bufferB, sizeB, streamB);
...
kernel<<<blocks,threads,streamA>>>(blockA,...);
cudaMemcpy(blockB, hostBuf, sizeB, streamB, ...);
...
RMM_FREE(bufferA, streamA);
...
RMM_FREE(bufferA, streamB);
```

RMM

Pool Allocation Example

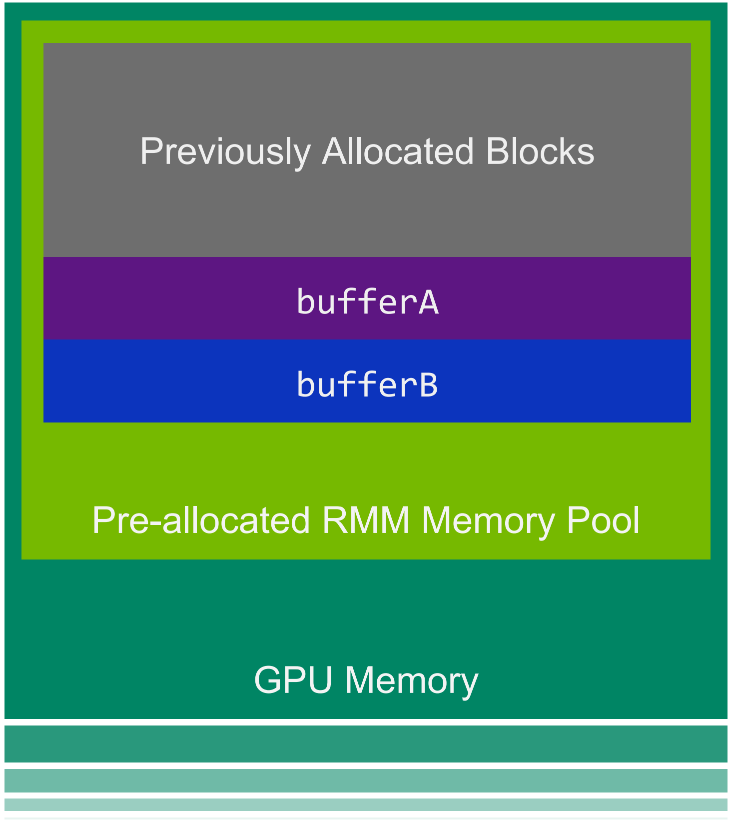



```
...

RMM_ALLOC(&bufferA, sizeA, streamA);
RMM_ALLOC(&bufferB, sizeB, streamB);
...
kernel<<<blocks,threads,streamA>>>(blockA,...);
cudaMemcpy(blockB, hostBuf, sizeB, streamB, ...);
...
RMM_FREE(bufferA, streamA);
...
RMM_FREE(bufferA, streamB);
```

RMM

Pool Allocation Example

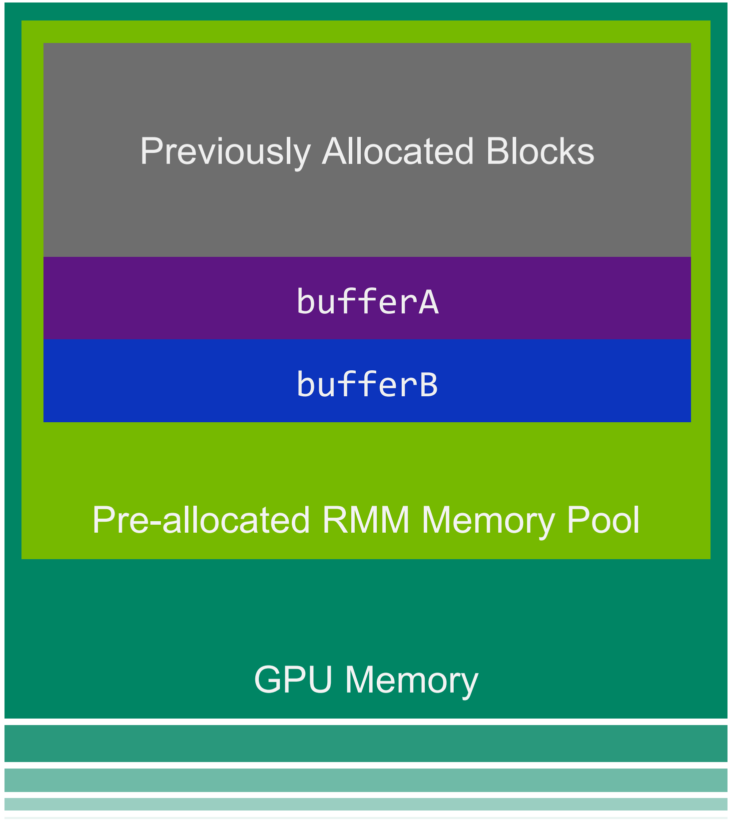



```
...

RMM_ALLOC(&bufferA, sizeA, streamA);
RMM_ALLOC(&bufferB, sizeB, streamB);
...
kernel<<<blocks,threads,streamA>>>(blockA,...);
cudaMemcpy(blockB, hostBuf, sizeB, streamB, ...);
...
RMM_FREE(bufferA, streamA);
...
RMM_FREE(bufferA, streamB);
```

RMM

Pool Allocation Example



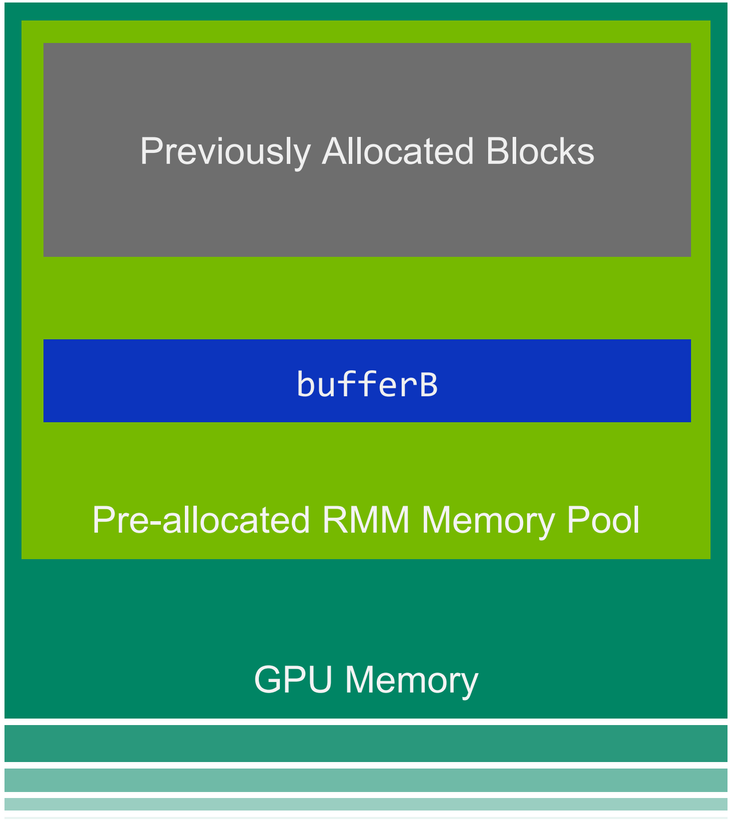

```
...

RMM_ALLOC(&bufferA, szA, streamA);
RMM_ALLOC(&bufferB, szB, streamB);
...
kernel<<<blocks,threads,streamA>>>(blockA,...);
cudaMemcpyAsync(blockB, hostBuf, szB, streamB,…);
...
RMM_FREE(bufferA, streamA);
...
RMM_FREE(bufferA, streamB);
```

Potential overlap!

RMM

Pool Allocation Example

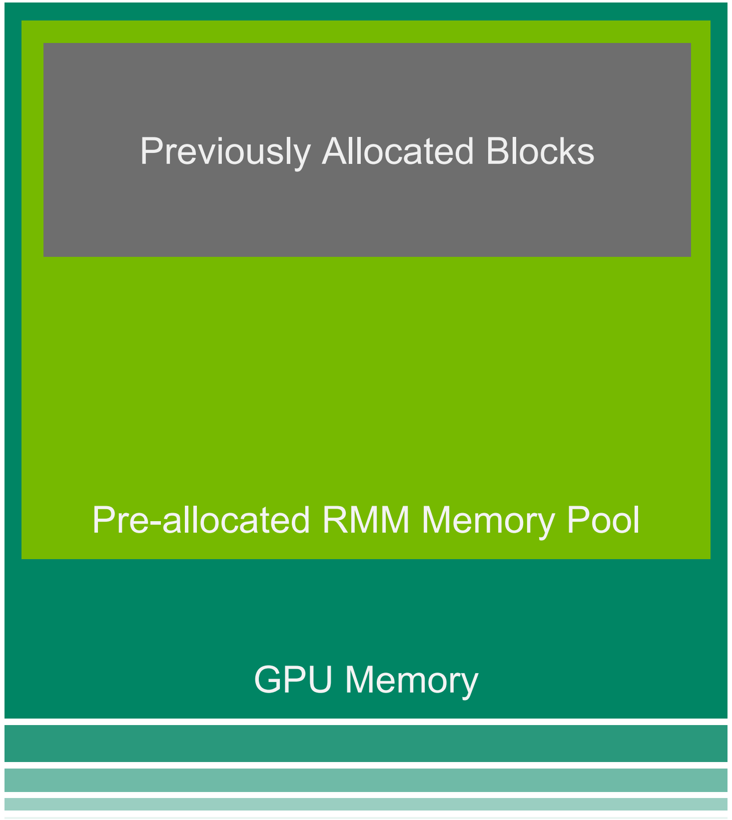



```
...

RMM_ALLOC(&bufferA, szA, streamA);
RMM_ALLOC(&bufferB, szB, streamB);
...
kernel<<<blocks,threads,streamA>>>(blockA,...);
cudaMemcpyAsync(blockB, hostBuf, szB, streamB,…);
...
RMM_FREE(bufferA, streamA);
...
RMM_FREE(bufferA, streamB);
```

RMM

Pool Allocation Example




```
...

RMM_ALLOC(&bufferA, szA, streamA);
RMM_ALLOC(&bufferB, szB, streamB);
...
kernel<<<blocks,threads,streamA>>>(blockA,...);
cudaMemcpyAsync(blockB, hostBuf, szB, streamB,…);
...
RMM_FREE(bufferA, streamA);
...
RMM_FREE(bufferA, streamB);
```