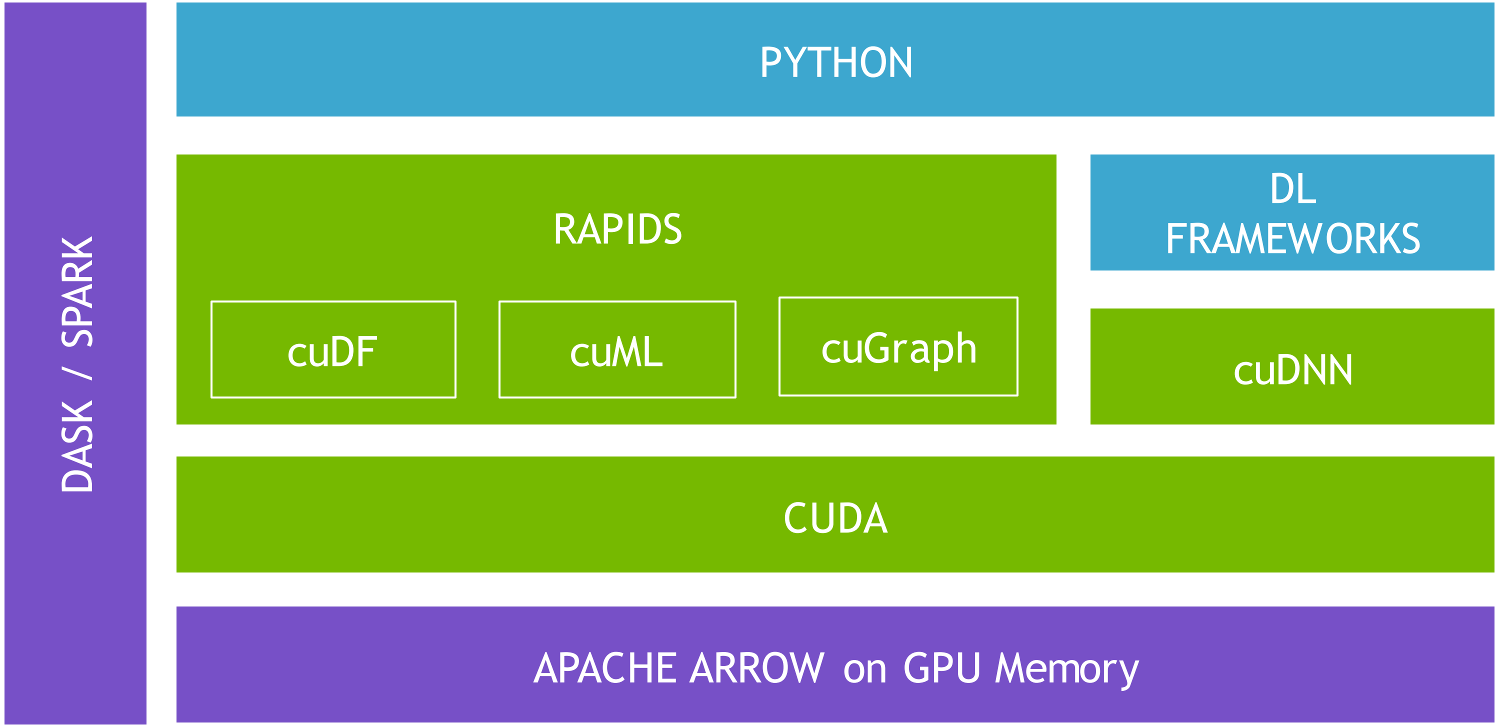
UNIFIED MEMORY FOR DATA ANALYTICS AND DEEP LEARNING

Nikolay Sakharnykh, Chirayu Garg, and Dmitri Vainbrand, Thu Mar 19, 3:00 PM

RAPIDS

CUDA-accelerated Data Science Libraries



MORTGAGE PIPELINE: ETL

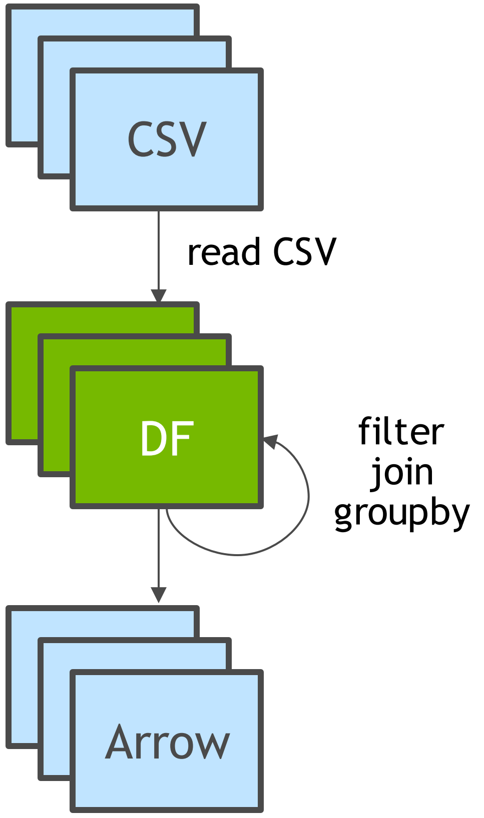
https://github.com/rapidsai/notebooks/blob/master/mortgage/E2E.ipynb

In [ ]:
```
client.run(initialize_rmm_pool)
```

In [ ]:
```
%%time

# NOTE: The ETL calculates additional features which are then dropped before creating the XGBoost DMatrix.
# This can be optimized to avoid calculating the dropped features.

gpu_dfs = []
gpu_time = 0
quarter = 1
year = start_year
count = 0
while year <= end_year:
    for file in glob(os.path.join(perf_data_path + "/Performance_" + str(year) + "Q" + str(quarter) + "*")):
        gpu_dfs.append(process_quarter_gpu(year=year, quarter=quarter, perf_file=file))
        count += 1
    quarter += 1
    if quarter == 5:
        year += 1
        quarter = 1
wait(gpu_dfs)
```

In [ ]:
```
client.run(cudf._gdf.rmm_finalize)
```

In [ ]:
```
client.run(initialize_rmm_no_pool)
```



MORTGAGE PIPELINE: PREP + ML

https://github.com/rapidsai/notebooks/blob/master/mortgage/E2E.ipynb

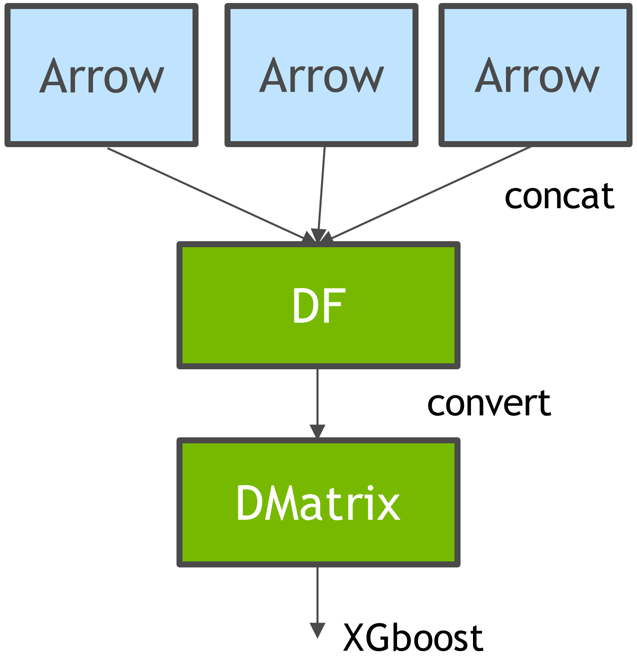
Load the data from host memory, and convert to CSR

In [ ]:
```
%%time

gpu_dfs = [delayed(DataFrame.from_arrow)(gpu_df) for gpu_df in gpu_dfs[:part_count]]
gpu_dfs = [gpu_df for gpu_df in gpu_dfs]
wait(gpu_dfs)

tmp_map = [(gpu_df, list(client.who_has(gpu_df).values())[0]) for gpu_df in gpu_dfs]
new_map = {}
for key, value in tmp_map:
    if value not in new_map:
        new_map[value] = [key]
    else:
        new_map[value].append(key)

del(tmp_map)
gpu_dfs = []
for list_delayed in new_map.values():
    gpu_dfs.append(delayed(cudf.concat)(list_delayed))

del(new_map)
gpu_dfs = [(gpu_df[['delinquency_12']], gpu_df[delayed(list)(gpu_df.columns.difference(['delinquen
cy_12']))]) for gpu_df in gpu_dfs]
gpu_dfs = [(gpu_df[0].persist(), gpu_df[1].persist()) for gpu_df in gpu_dfs]

gpu_dfs = [dask.delayed(xgb.DMatrix)(gpu_df[1], gpu_df[0]) for gpu_df in gpu_dfs]
gpu_dfs = [gpu_df.persist() for gpu_df in gpu_dfs]
gc.collect()
wait(gpu_dfs)
```

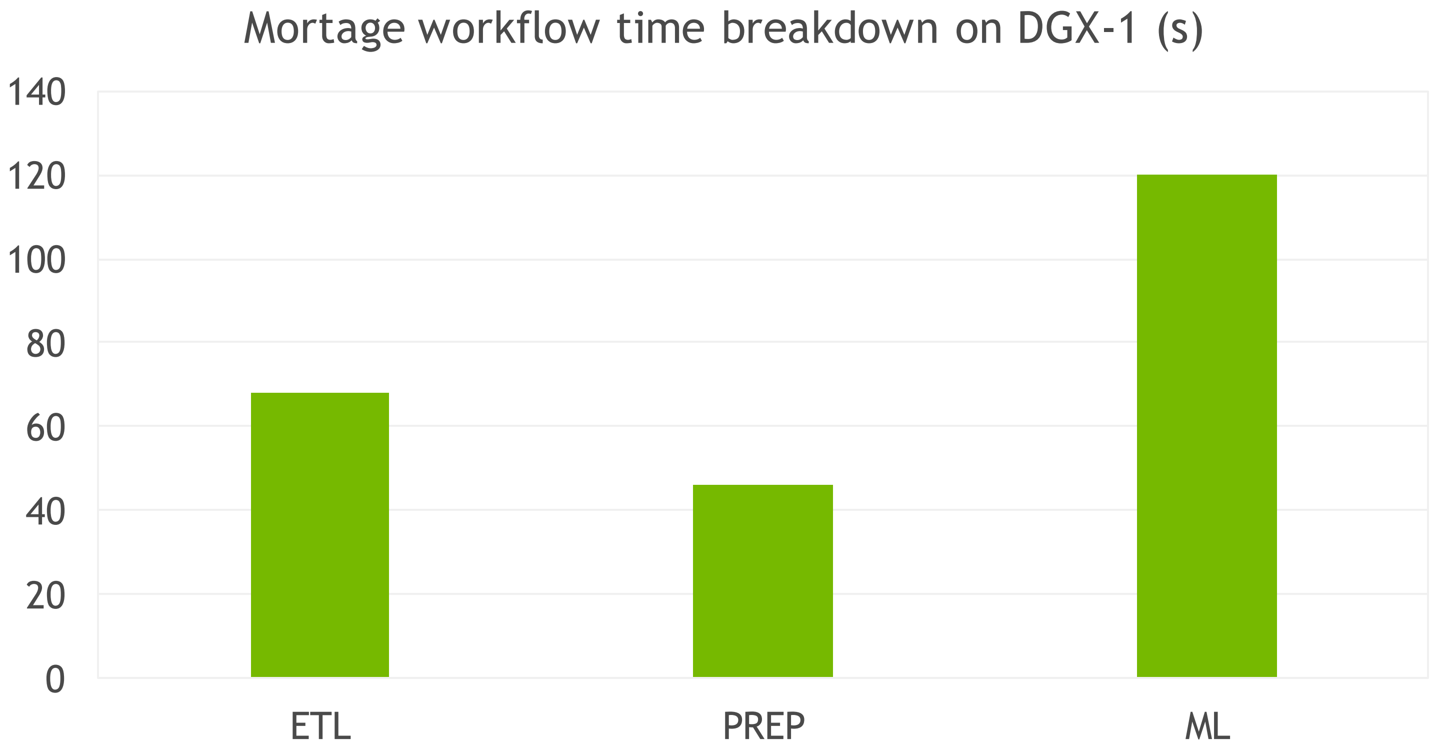
Train the Gradient Boosted Decision Tree with a single call to

```
dask_xgboost.train(client, params, data, labels, num_boost_round=dxgb_gpu_params['nround'])
```

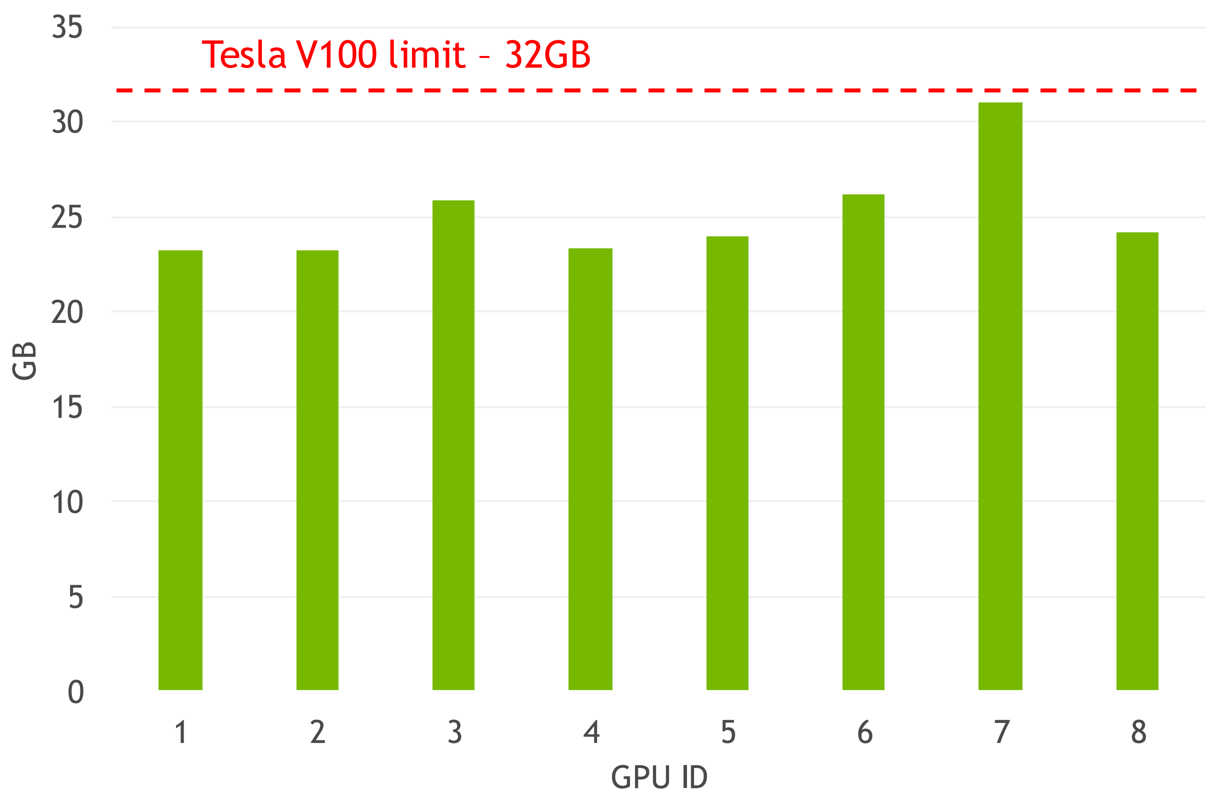
In [ ]:
```
%%time
labels = None
bst = dxgb_gpu.train(client, dxgb_gpu_params, gpu_dfs, labels, num_boost_round=dxgb_gpu_params['nr
ound'])
```



GTC EU KEYNOTE RESULTS ON DGX-1

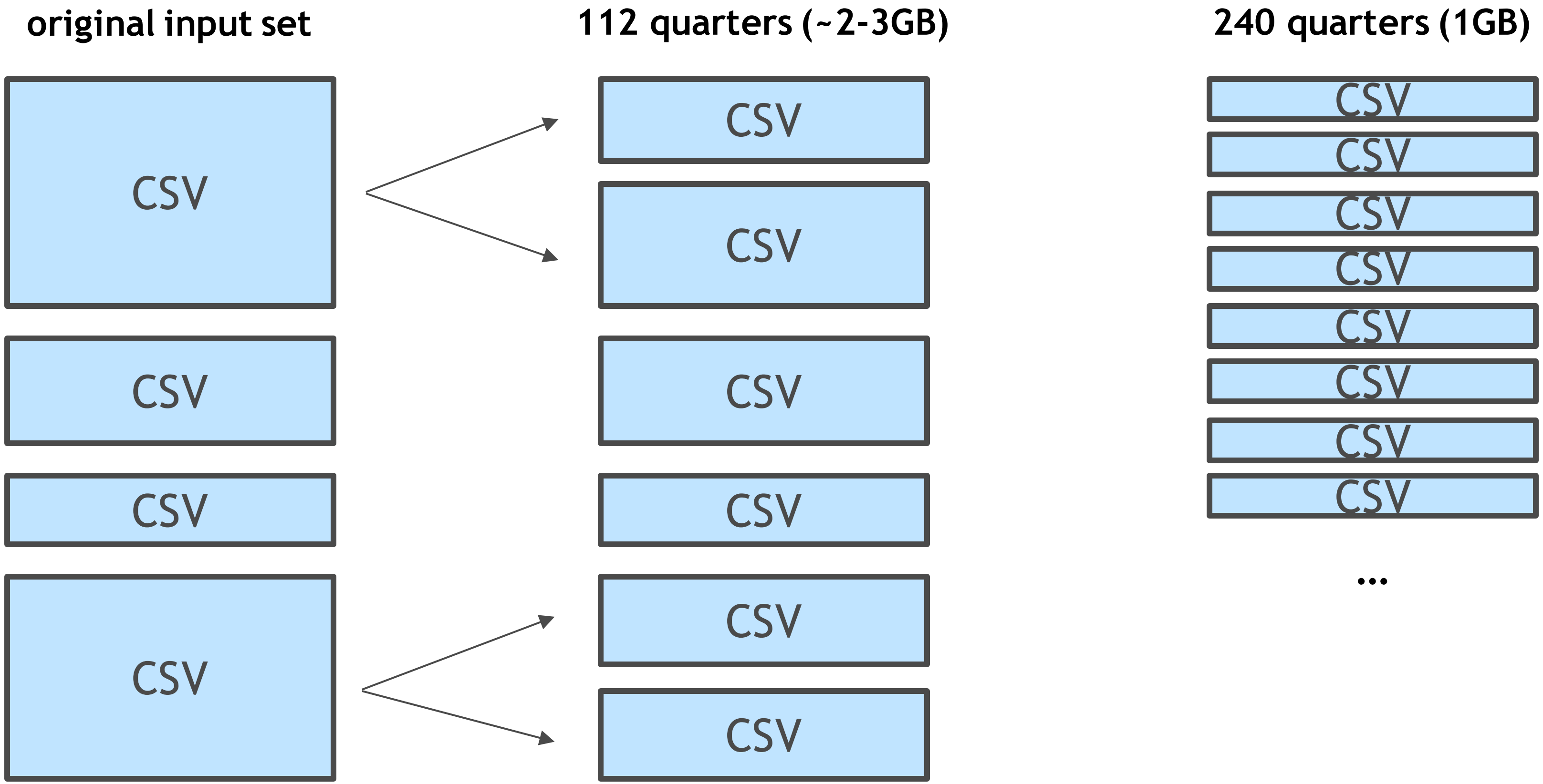


MAXIMUM MEMORY USAGE ON DGX-1

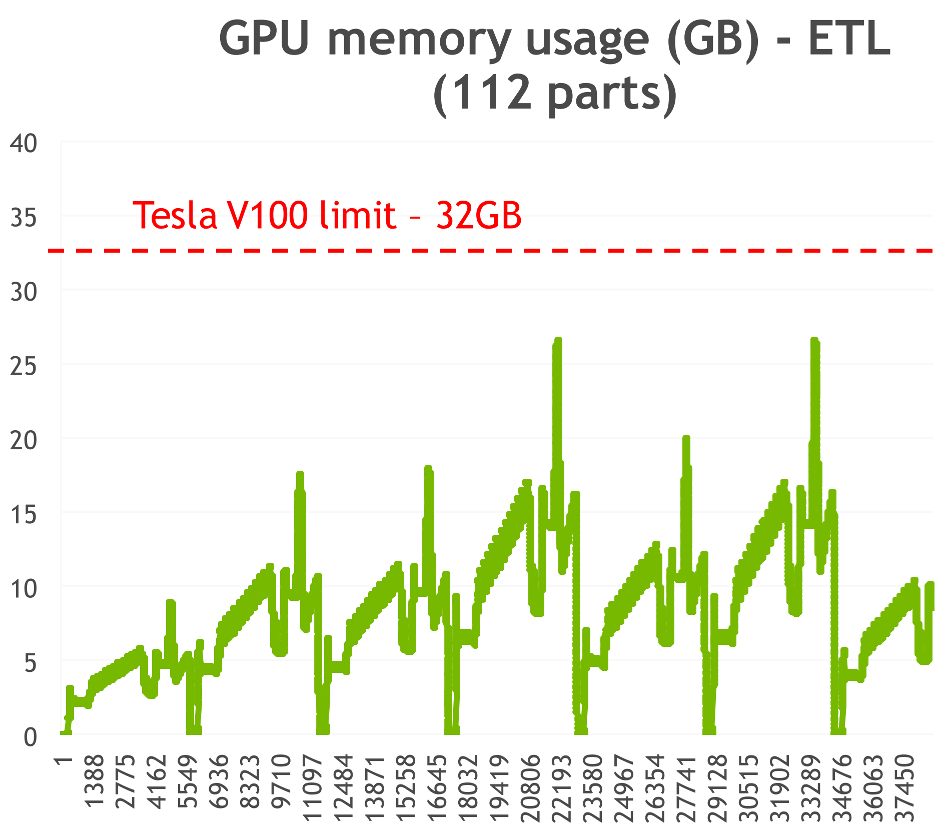


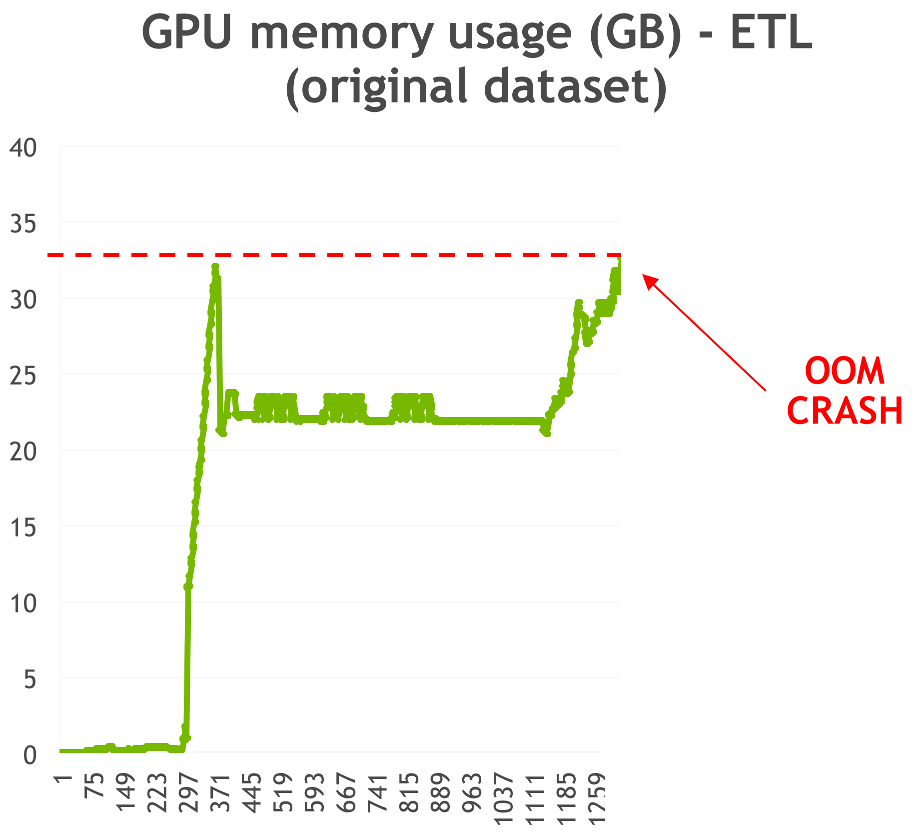
ETL INPUT

https://rapidsai.github.io/demos/datasets/mortgage-data



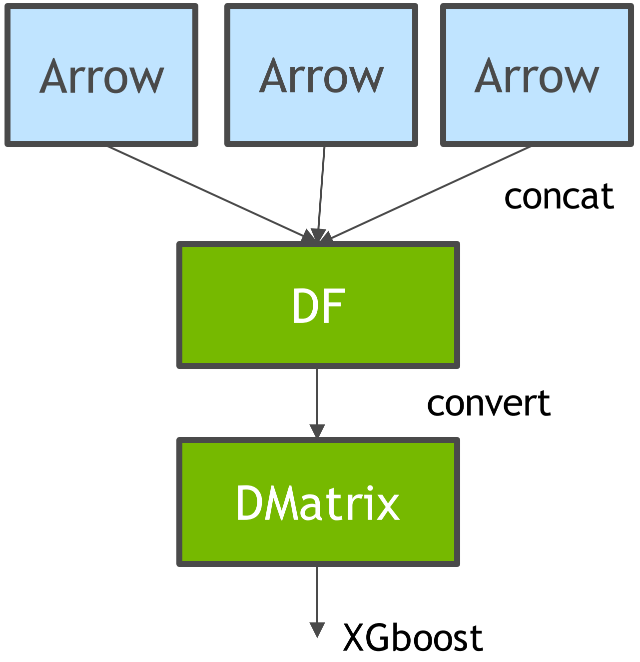
CAN WE AVOID INPUT SPLITTING?



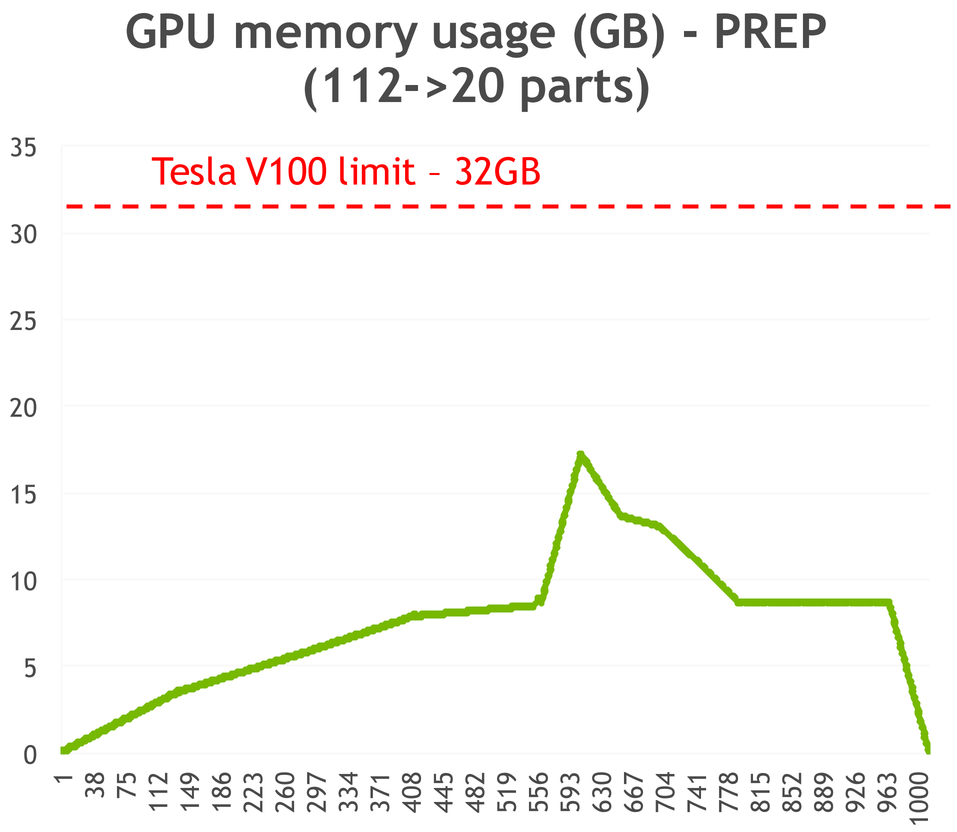


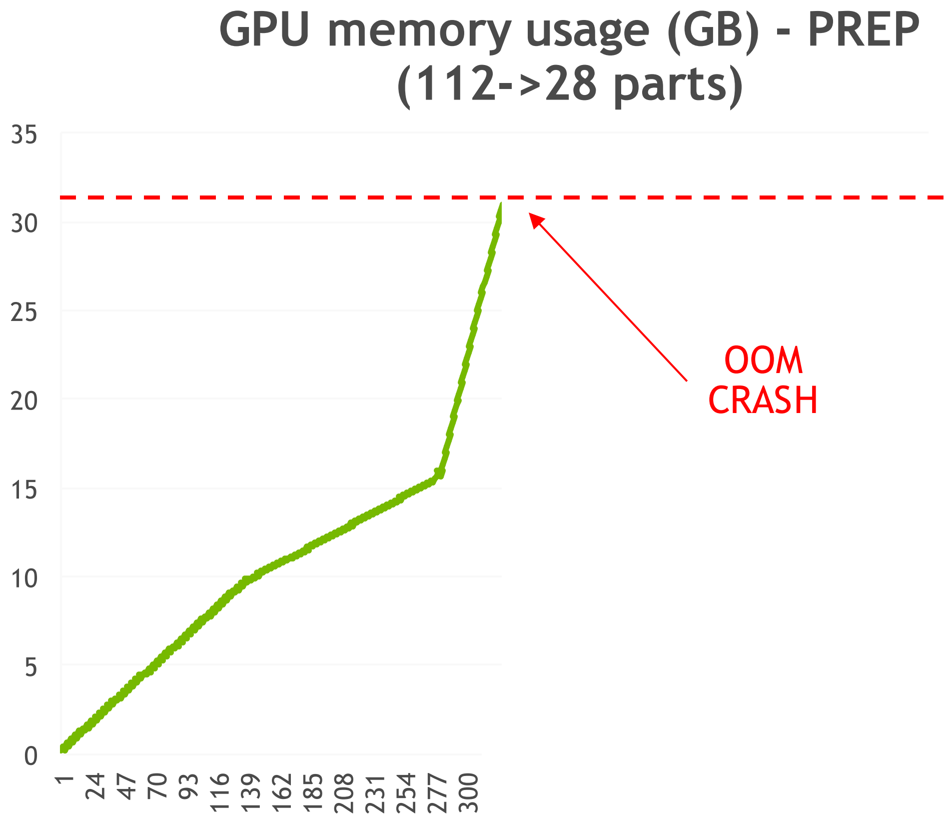
ML INPUT

Some # of quarters are used for ML training



CAN WE TRAIN ON MORE DATA?





HOW MEMORY MANAGED IN RAPIDS

In [ ]:
```
client.run(initialize_rmm_pool)
```

In [ ]:
```
%%time

# NOTE: The ETL calculates additional features which are then dropped before creating the XGBoost
DMatrix.
# This can be optimized to avoid calculating the dropped features.

gpu_dfs = []
gpu_time = 0
quarter = 1
year = start_year
count = 0
while year <= end_year:
    for file in glob(os.path.join(perf_data_path + "/Performance_" + str(year) + "Q" + str(quarter
) + "*")):
        gpu_dfs.append(process_quarter_gpu(year=year, quarter=quarter, perf_file=file))
        count += 1
    quarter += 1
    if quarter == 5:
        year += 1
        quarter = 1
wait(gpu_dfs)
```

In [ ]:
```
client.run(cudf._gdf.rmm_finalize)
```

In [ ]:
```
client.run(initialize_rmm_no_pool)
```

Load the data from host memory, and convert to CSR

In [ ]:
```
%%time

gpu_dfs = [delayed(DataFrame.from_arrow)(gpu_df) for gpu_df in gpu_dfs[:part_count]]
gpu_dfs = [gpu_df for gpu_df in gpu_dfs]
wait(gpu_dfs)

tmp_map = [(gpu_df, list(client.who_has(gpu_df).values())[0]) for gpu_df in gpu_dfs]
new_map = {}
for key, value in tmp_map:
    if value not in new_map:
```


NVIDIA

RAPIDS MEMORY MANAGER

https://github.com/rapidsai/rmm

RAPIDS Memory Manager (RMM) is:

- A replacement allocator for CUDA Device Memory
- A pool allocator to make CUDA device memory allocation faster & asynchronous
- A central place for all device memory allocations in cuDF and other RAPIDS libraries

WHY DO WE NEED MEMORY POOLS

cudaMalloc/cudaFree are synchronous

- block the device

cudaMalloc/cudaFree are expensive

- cudaFree must zero memory for security
- cudaMalloc creates peer mappings for all GPUs

Using cnmem memory pool **improves RAPIDS ETL time by 10x**

```
cudaMalloc(&buffer, size_in_bytes);

cudaFree(buffer);
```

RAPIDS MEMORY MANAGER (RMM)

Fast, Asynchronous Device Memory Management

C/C++

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

RMM_FREE(buffer, stream_id);
```

Python: drop-in replacement for Numba API

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

Thrust: device vector and execution policies

```
#include <rmm_thrust_allocator.h>
rmm::device_vector<int> dvec(size);

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

MANAGING MEMORY IN THE E2E PIPELINE

perf optimization

```
In [ ]: client.run(initialize_rmm_pool)
```

```
In [ ]: %%time

# NOTE: The ETL calculates additional features which are then dropped before creating the XGBoost
DMatrix.
# This can be optimized to avoid calculating the dropped features.

gpu_dfs = []
gpu_time = 0
quarter = 1
year = start_year
count = 0
while year <= end_year:
    for file in glob(os.path.join(perf_data_path + "/Performance_" + str(year) + "Q" + str(quarter
) + "*")):
        gpu_dfs.append(process_quarter_gpu(year=year, quarter=quarter, perf_file=file))
        count += 1
    quarter += 1
    if quarter == 5:
        year += 1
        quarter = 1
wait(gpu_dfs)
```

```
In [ ]: client.run(cudf._gdf.rmm_finalize)
```

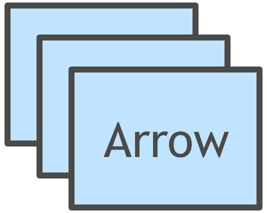
required to avoid OOM

```
In [ ]: client.run(initialize_rmm_no_pool)
```

Load the data from host memory, and convert to CSR

```
In [ ]: %%time

gpu_dfs = [delayed(DataFrame.from_arrow)(gpu_df) for gpu_df in gpu_dfs[:part_count]]
gpu_dfs = [gpu_df for gpu_df in gpu_dfs]
wait(gpu_dfs)

tmp_map = [(gpu_df, list(client.who_has(gpu_df).values())[0]) for gpu_df in gpu_dfs]
new_map = {}
for key, value in tmp_map:
    if value not in new_map:
```

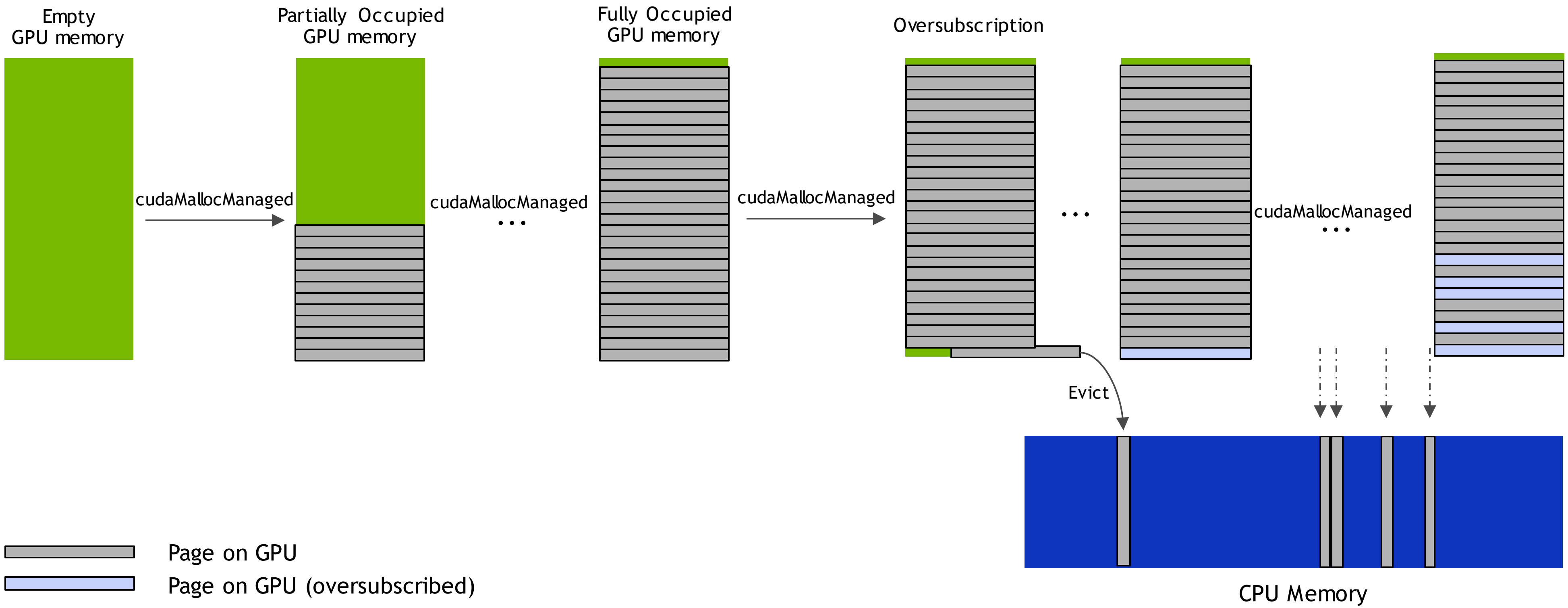
At this point all ETL processing is done and memory stored in arrow



KEY MEMORY MANAGEMENT QUESTIONS

- Can we make memory management easier?
- Can we avoid artificial pre-processing of input data?
- Can we train on larger datasets?

SOLUTION: UNIFIED MEMORY



HOW TO USE UNIFIED MEMORY IN CUDF

Python

```
from librmm_cffi import librmm_config as rmm_cfg

rmm_cfg.use_pool_allocator = True   # default is False
rmm_cfg.use_managed_memory = True   # default is False
```

IMPLEMENTATION DETAILS

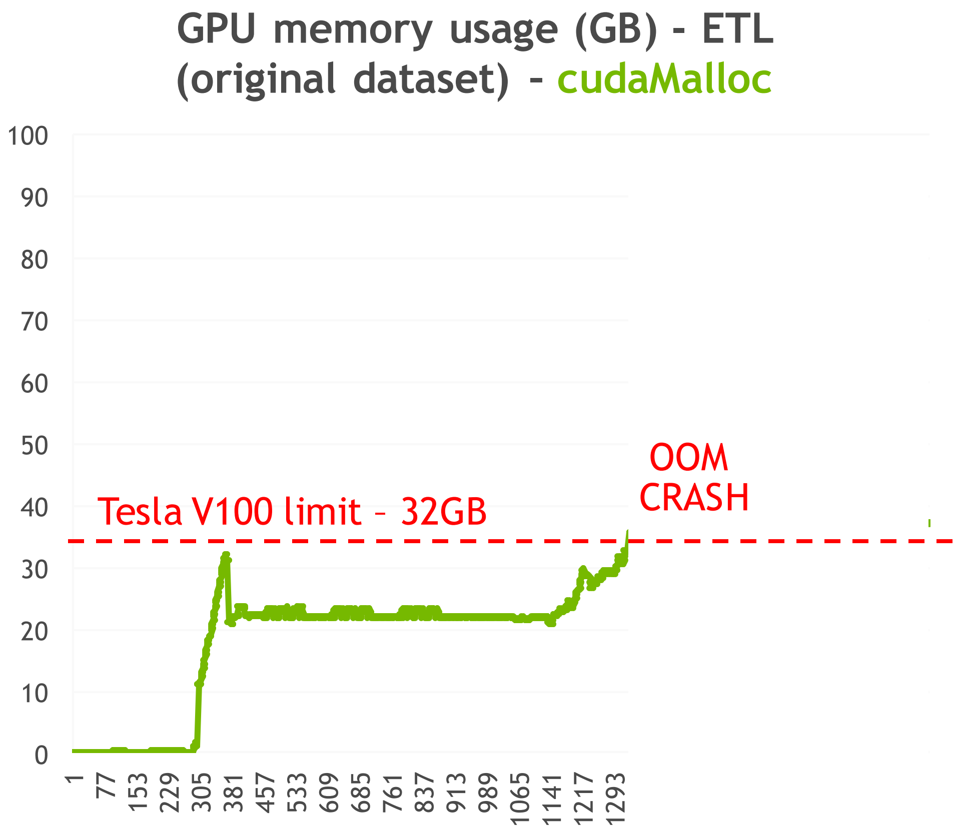
Regular RMM allocation:

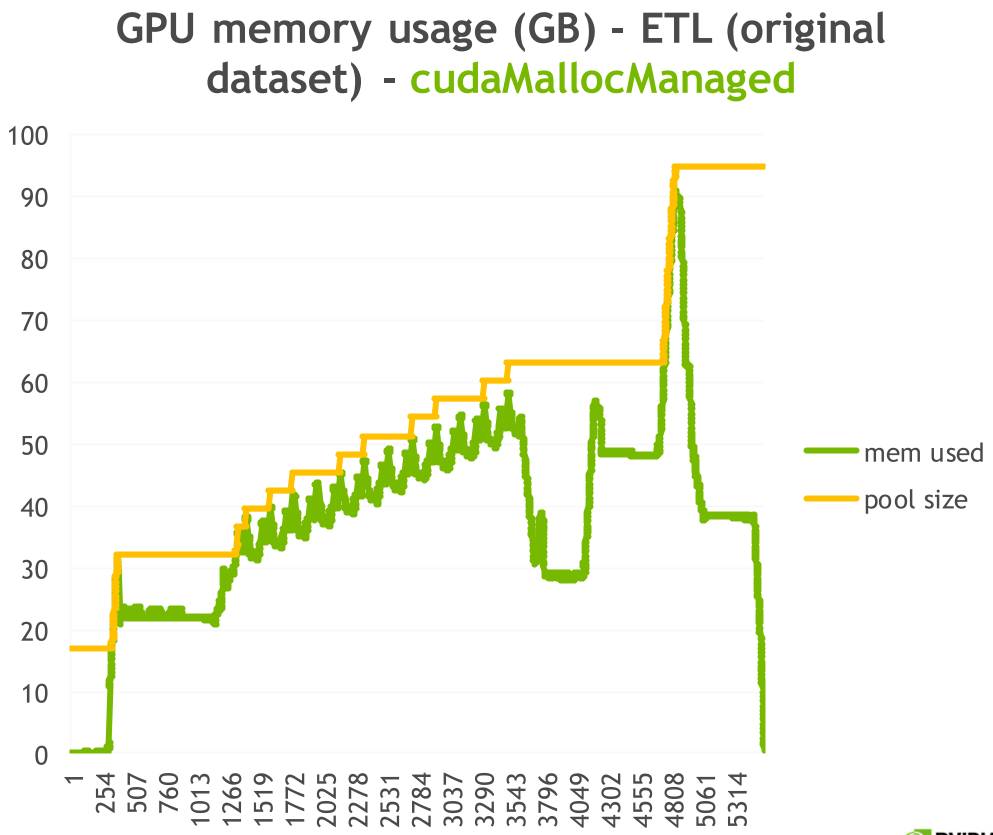
```
if (rmm::Manager::usePoolAllocator()) {
  RMM_CHECK(rmm::Manager::getInstance().registerStream(stream));
  RMM_CHECK_CNMEM(cnmemMalloc(reinterpret_cast<void**>(ptr), size, stream));
}
else if (rmm::Manager::useManagedMemory())
  RMM_CHECK_CUDA(cudaMallocManaged(reinterpret_cast<void**>(ptr), size));
else
  RMM_CHECK_CUDA(cudaMalloc(reinterpret_cast<void**>(ptr), size));
```

Pool allocator (CNMEM):

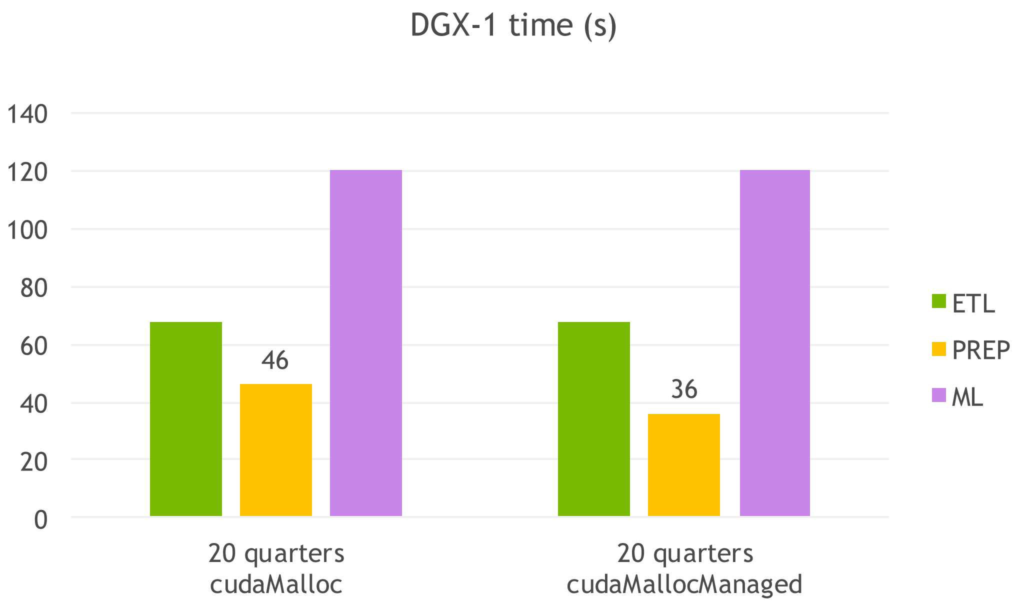
```
if (mFlags & CNMEM_FLAGS_MANAGED) {
  CNMEM_DEBUG_INFO("cudaMallocManaged(%lu)\n", size);
  CNMEM_CHECK_CUDA(cudaMallocManaged(&data, size));
  CNMEM_CHECK_CUDA(cudaMemPrefetchAsync(data, size, mDevice));
}
else {
  CNMEM_DEBUG_INFO("cudaMalloc(%lu)\n", size);
  CNMEM_CHECK_CUDA(cudaMalloc(&data, size));
}
```

1. UNSPLIT DATASET "JUST WORKS"





2. SPEED-UP ON CONVERSION



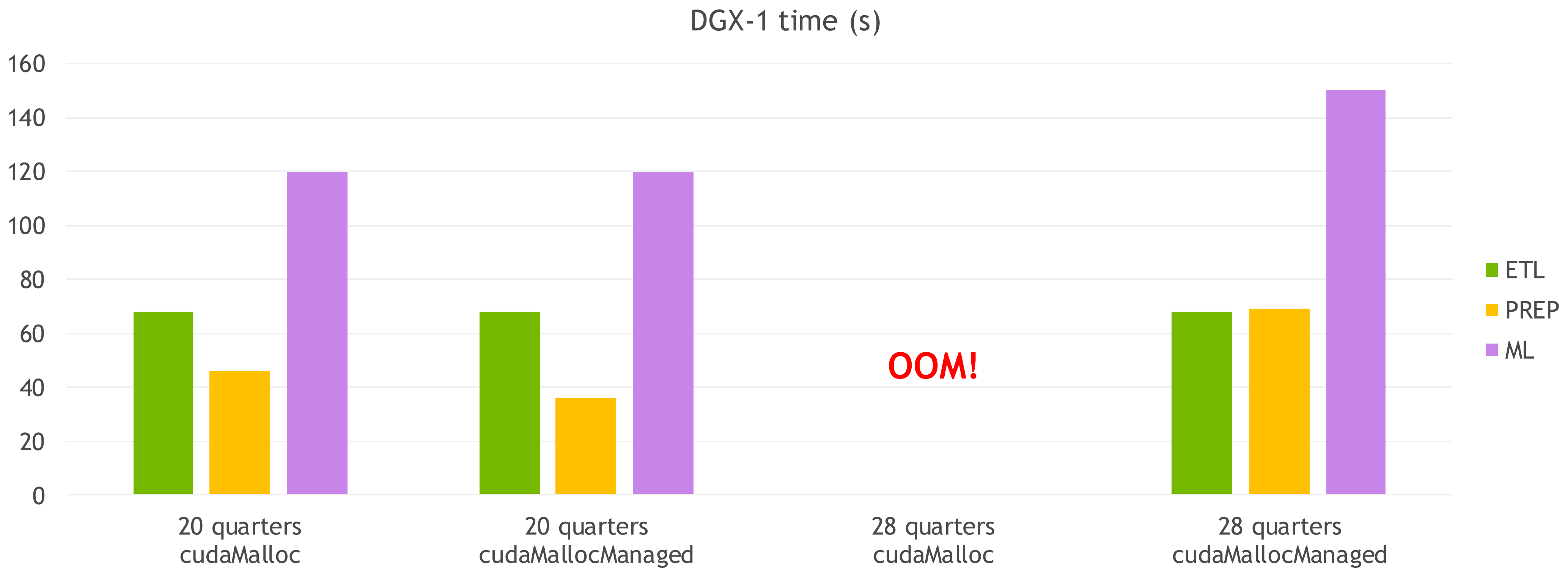
25% speed-up on PREP!

```
In [ ]: client.run(initialize_rmm_pool)

In [ ]: %%time

        # NOTE: The ETL calculates additional features which are then dropped before creating the XGBoost DMatrix.
        # This can be optimized to avoid calculating the dropped features.

        gpu_dfs = []
        gpu_time = 0
        quarter = 1
        year = start_year
        count = 0
        while year <= end_year:
            for file in glob(os.path.join(perf_data_path + "/Performance_" + str(year) + "Q" + str(quarter
        ) + "*")):
                gpu_dfs.append(process_quarter_gpu(year=year, quarter=quarter, perf_file=file))
                count += 1
            quarter += 1
            if quarter == 5:
                year += 1
                quarter = 1
        wait(gpu_dfs)

In [ ]: client.run(cudf._gdf.rmm_finalize)

In [ ]: client.run(initialize_rmm_no_pool)
```

Load the data from host memory, and convert to CSR

```
In [ ]: %%time

        gpu_dfs = [delayed(DataFrame.from_arrow)(gpu_df) for gpu_df in gpu_dfs[:part_count]]
        gpu_dfs = [gpu_df for gpu_df in gpu_dfs]
        wait(gpu_dfs)

        tmp_map = [(gpu_df, list(client.who_has(gpu_df).values())[0]) for gpu_df in gpu_dfs]
        new_map = {}
        for key, value in tmp_map:
            if value not in new_map:
```

3. LARGER ML TRAINING SET



UNIFIED MEMORY GOTCHAS

1. **UVM doesn't work with CUDA IPC - careful when sharing data between processes**

    Workaround - separate (small) cudaMalloc pool for communication buffers

    In the future it will work transparently with Linux HMM

2. **Yes, you can oversubscribe, but there is danger that it will just run very slowly**

    Capture Nsight or nvprof profiles to check eviction traffic

    In the future RMM may show some warnings about this

RECAP

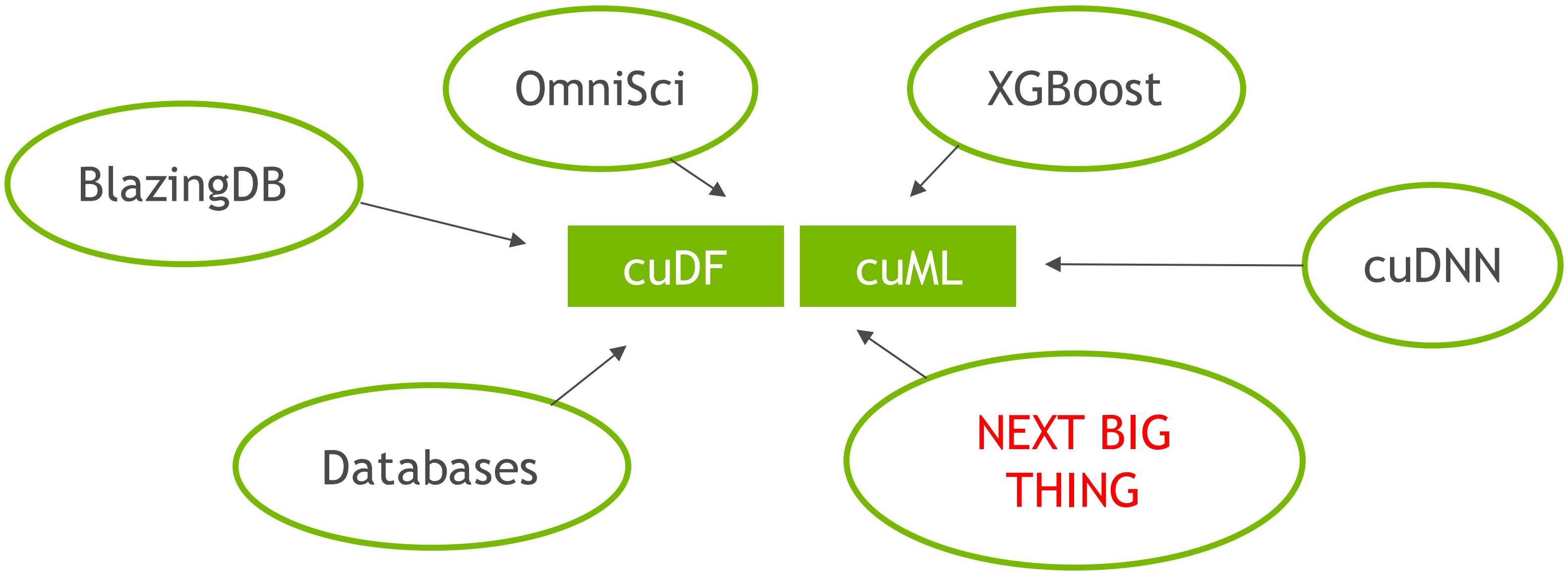
Just to run the full pipeline on the GPU you need

- carefully partition input data
- adjust memory pool options throughout the pipeline
- limit training size to fit in memory

Unified Memory

- makes life easier for data scientists – less tweaking!
- improves performance – sometimes it's faster to allocate less often & oversubscribe
- enables easy experiments with larger datasets

MEMORY MANAGEMENT IN THE FUTURE



Contribute to RAPIDS: https://github.com/rapidsai/cudf

Contribute to RMM: https://github.com/rapidsai/rmm

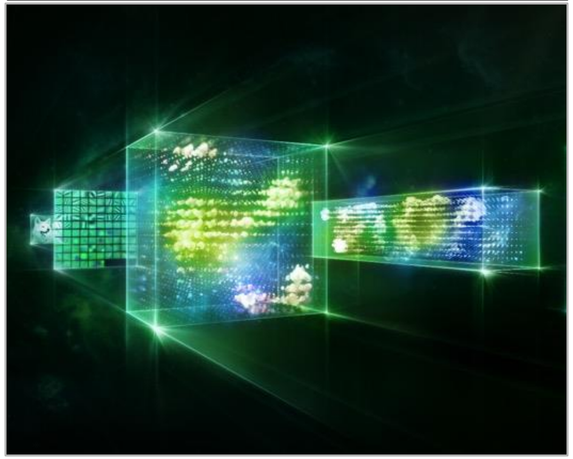
UNIFIED MEMORY FOR DEEP LEARNING

FROM ANALYTICS TO DEEP LEARNING

Data Preparation

Machine Learning

Deep Learning



PYTORCH INTEGRATION

PyTorch uses a caching allocator to manage GPU memory

- Small allocations distributed from fixed buffer (for ex: 1 MB)
- Large allocations are dedicated cudaMalloc's

Trivial change

- Replace cudaMalloc with cudaMallocManaged
- **Immediately** call cudaMemPrefetchAsync to allocate pages on GPU
  - Otherwise cuDNN may select sub-optimal kernels

PYTORCH ALLOCATOR VS RMM

PyTorch Caching Allocator

Memory pool to avoid synchronization on malloc/free

Directly uses CUDA APIs for memory allocations

Pool size not fixed

Specific to PyTorch C++ library

RMM

Memory pool to avoid synchronization on malloc/free

Uses Cnmem for memory allocation and management

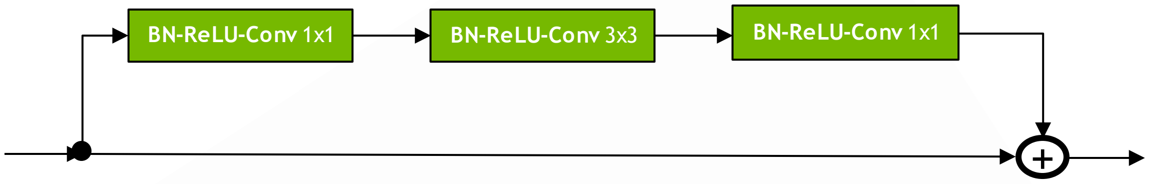
Reserves half the available GPU memory for pool

Re-usable across projects and with interfaces for various languages

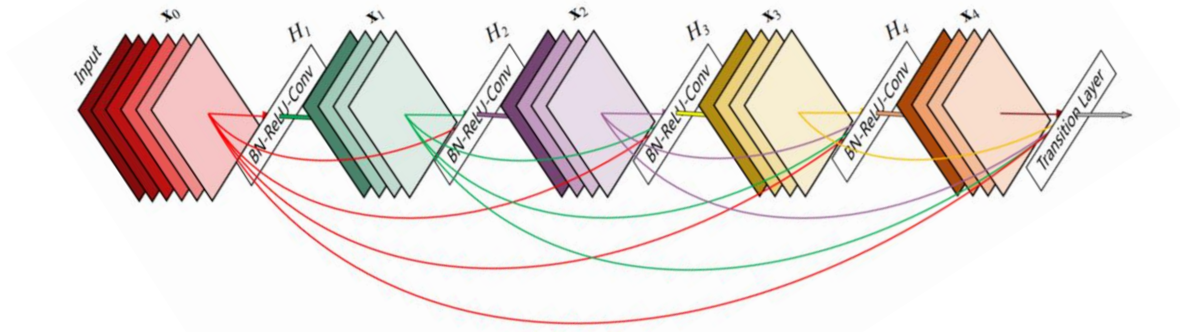
WORKLOADS

Image Models

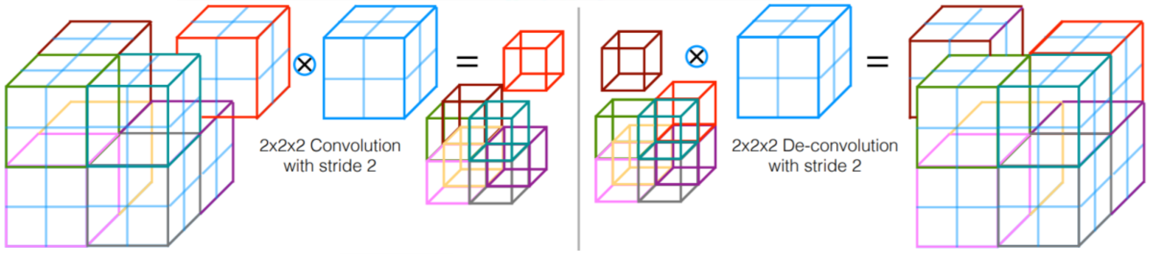
ResNet-1001



DenseNet-264



VNet

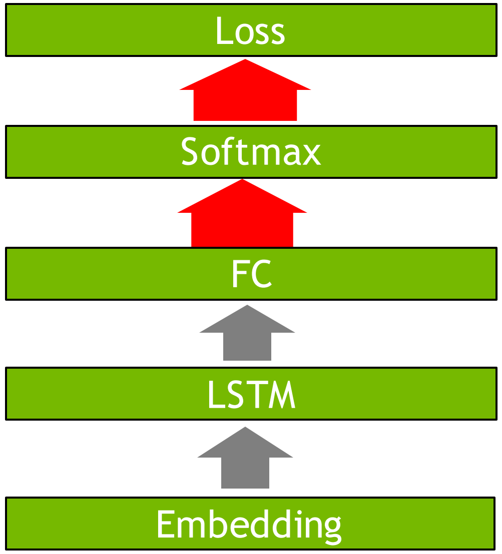


WORKLOADS

Language Models

Word Language Modelling

- Dictionary Size = 33278
- Embedding Size = 256
- LSTM units = 256
- Back propagation through time = 1408 and 2800



WORKLOADS

Baseline Training Performance on V100-32GB

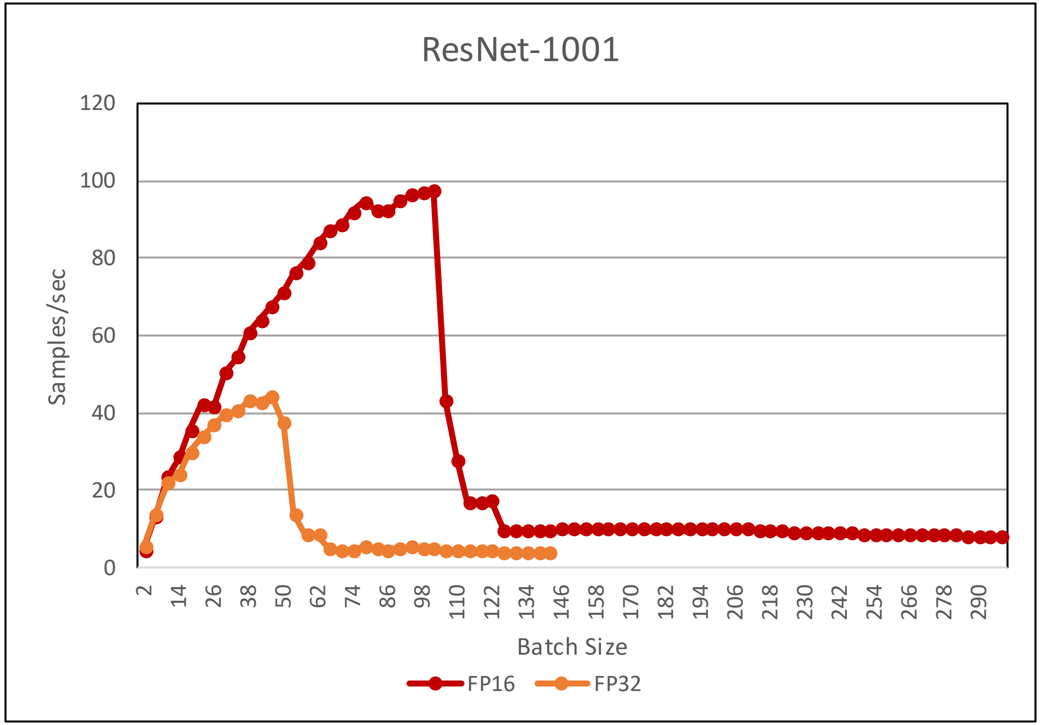
Model	FP16		FP32	
	Batch Size	Samples/sec	Batch Size	Samples/sec
ResNet-1001	98	98.7	48	44.3
DenseNet-264	218	255.8	109	143.1
Vnet	30	3.56	15	3.4
Lang_Model-1408	32	94.9	40	77.9
Lang_Model-2800	16	46.5	18	35.7

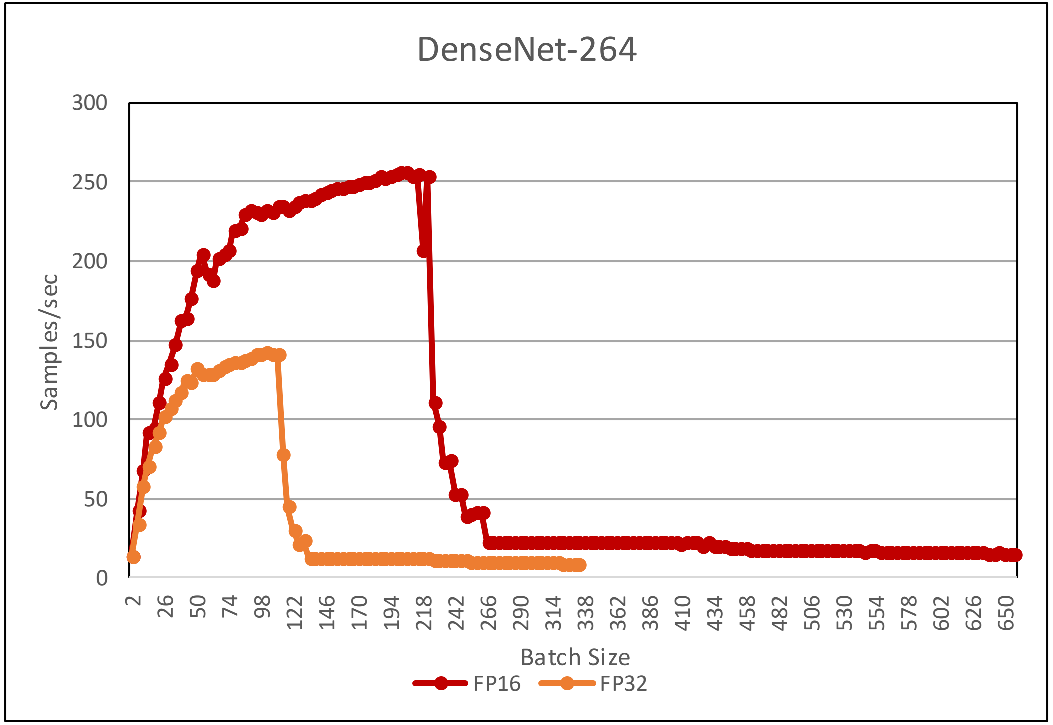
Optimal Batch Size Selected for High Throughput

All results in this presentation are using PyTorch 1.0rc1, R418 driver, Tesla V100-32GB

GPU OVERSUBSCRIPTION

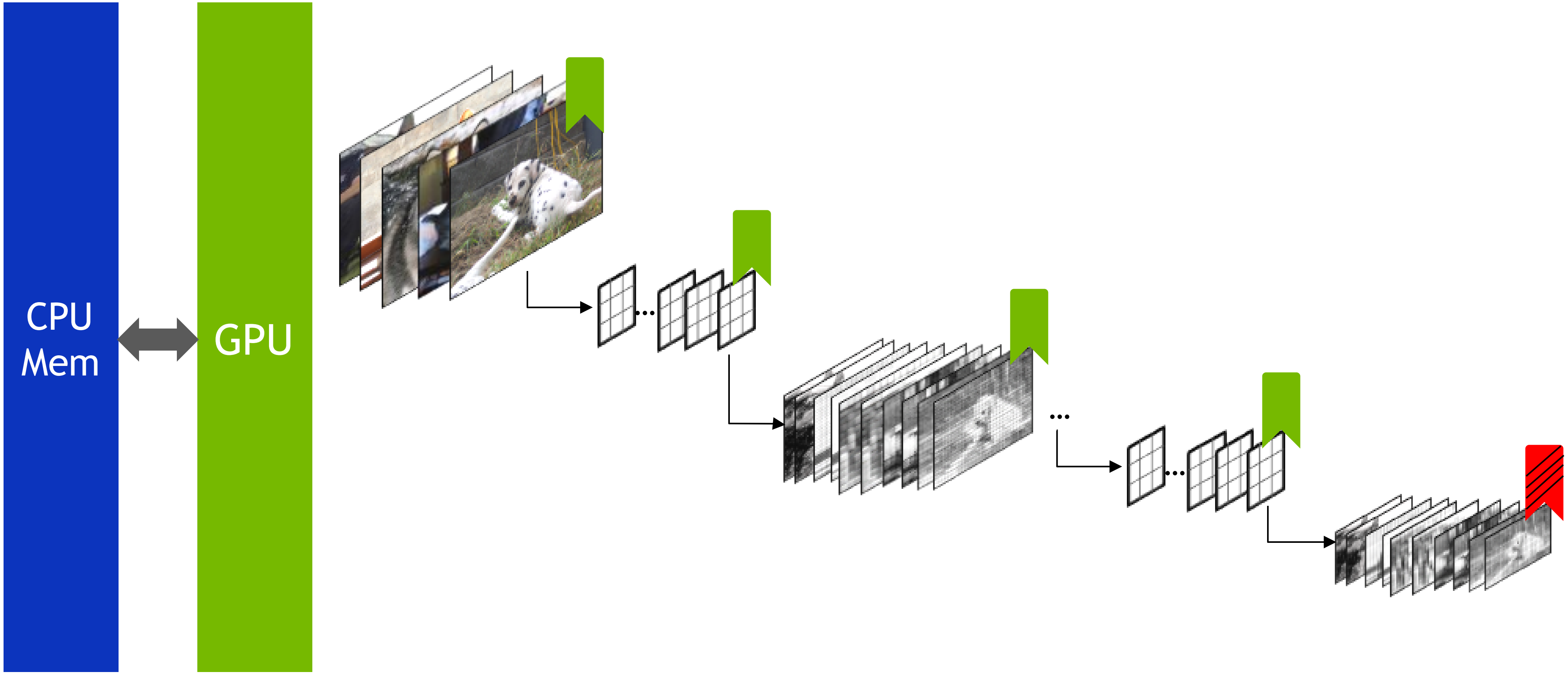
Upto 3x Optimal Batch Size





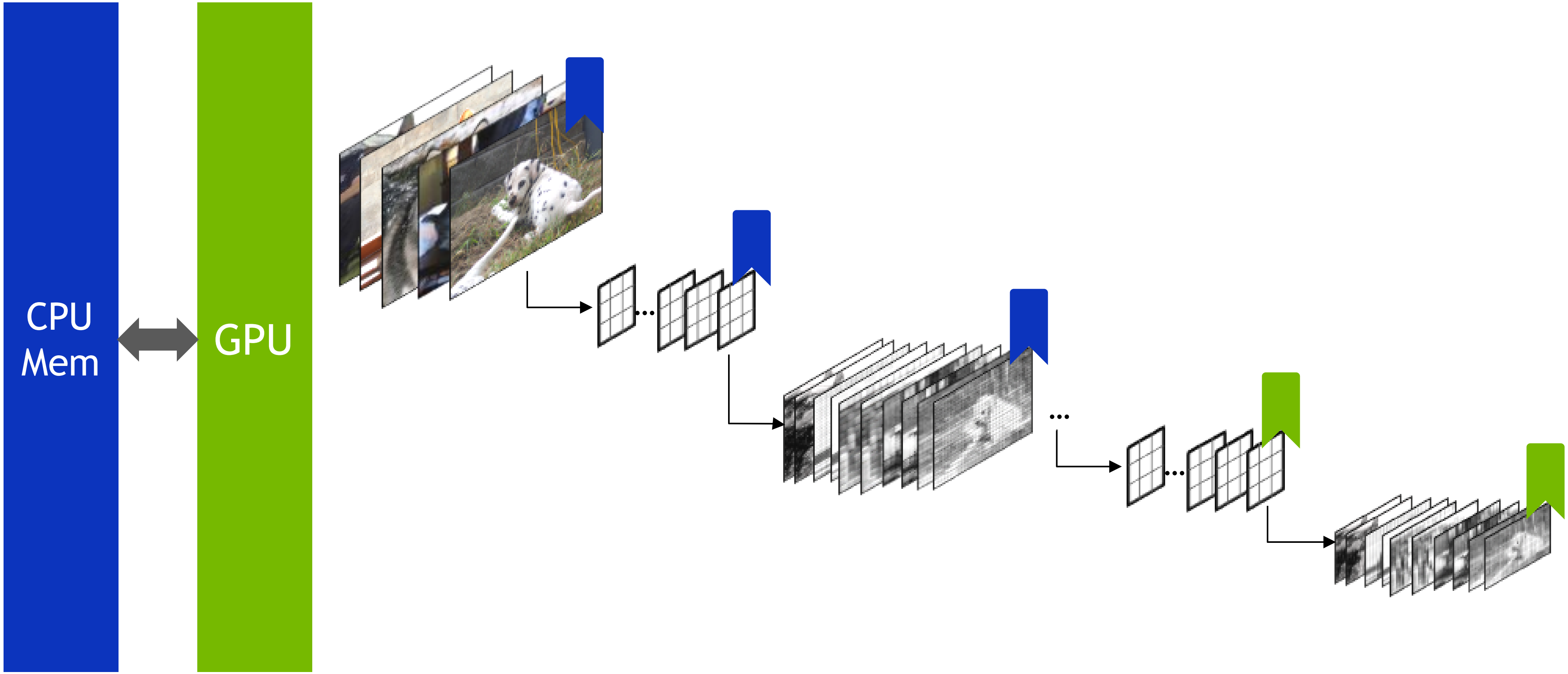
GPU OVERSUBSCRIPTION

Fill



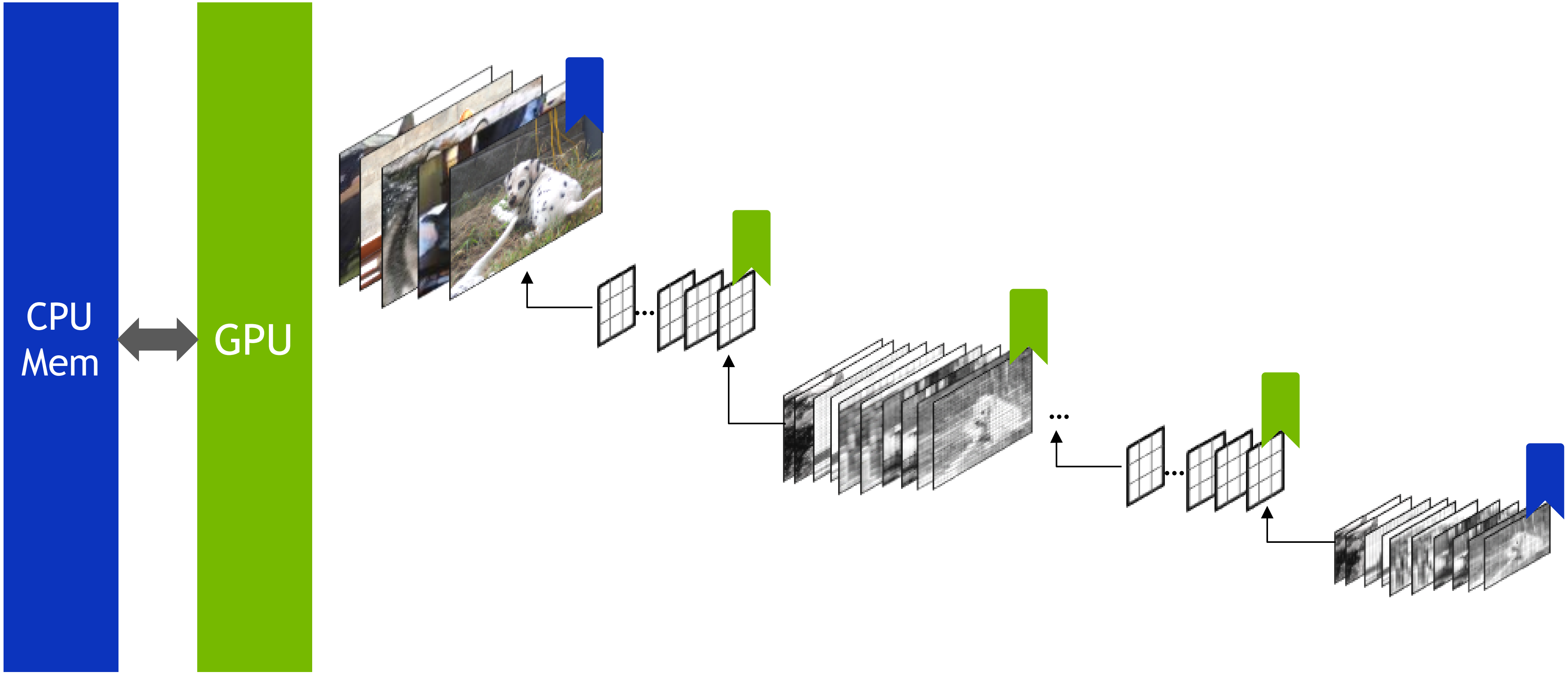
GPU OVERSUBSCRIPTION

Evict



GPU OVERSUBSCRIPTION

Page Fault-Evict-Fetch



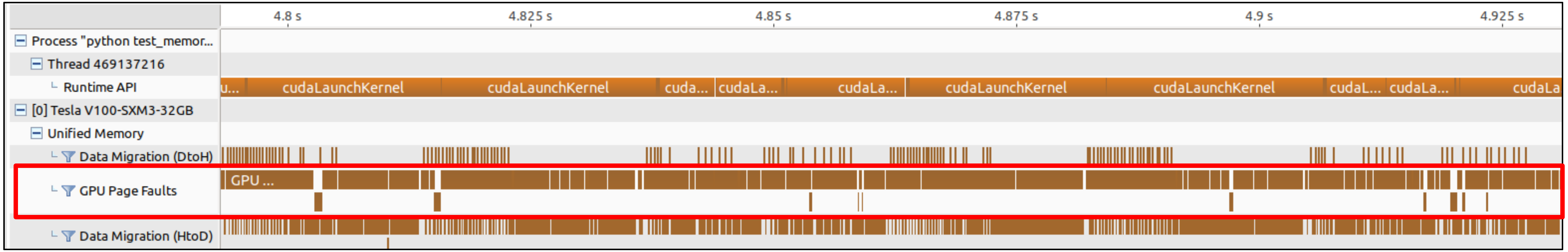
GPU OVERSUBSCRIPTION

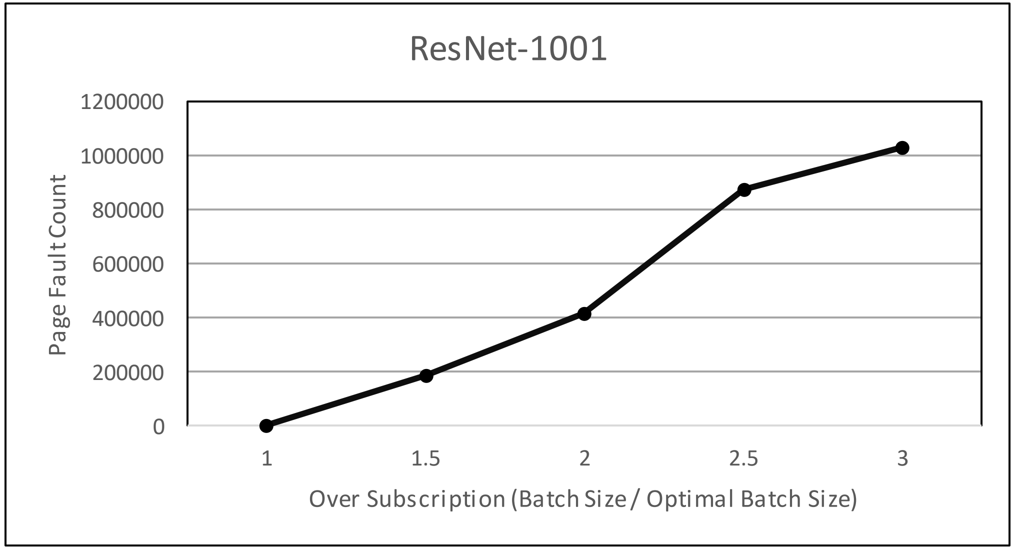
Results

Model	FP16		FP32	
	Batch Size	Samples/sec	Batch Size	Samples/sec
ResNet-1001	202	10.1	98	5
DenseNet-264	430	22.3	218	12.1
VNet	32	3	32	1.1
Lang_Model-1408	44	8.4	44	10
Lang_Model-2800	22	4.1	22	4.9

GPU OVERSUBSCRIPTION

Page Faults - ResNet-1001 Training Iteration





GPU OVERSUBSCRIPTION

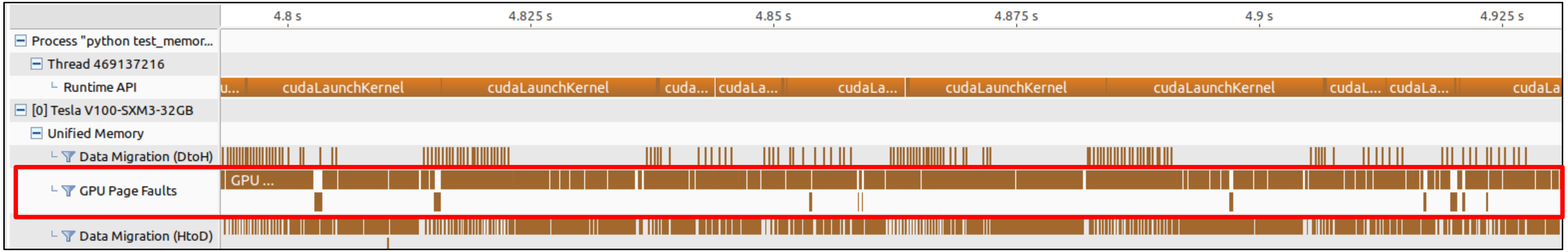
Manual API Prefetch

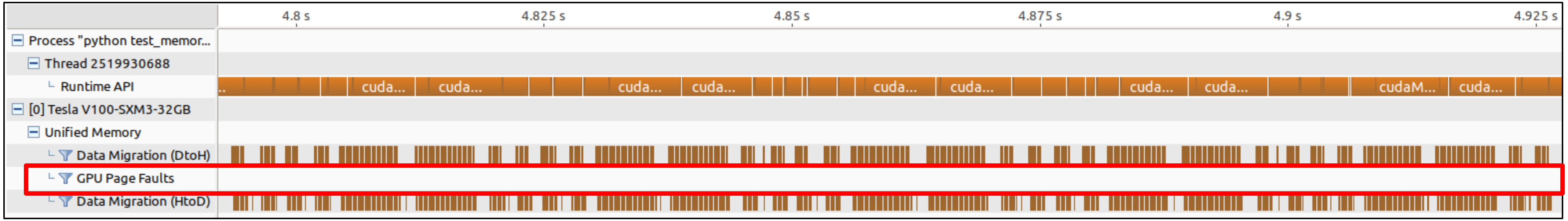
Add cudaMemPrefetchAsync before kernels are called

```
cudaMemPrefetchAsync(…)     // input, output, wparam
cudnnConvolutionForward(…)
---------------------------------------------------
cudaMemPrefetchAsync(…)     // A, B, C
kernelPointWiseApply3(…)
```

GPU OVERSUBSCRIPTION

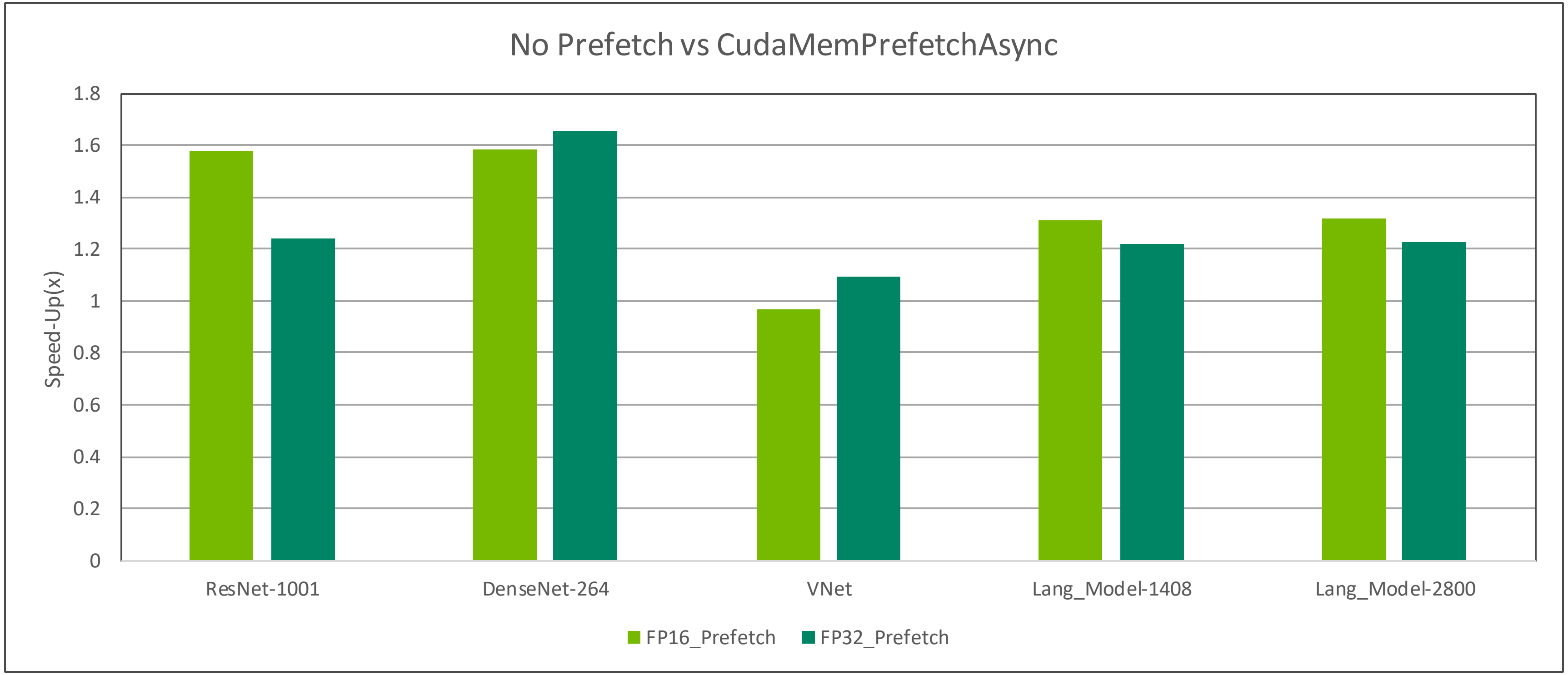
No Prefetch vs Manual API Prefetch





GPU OVERSUBSCRIPTION

Speed up from Manual API Prefetch



Observe upto 1.6x speed-up

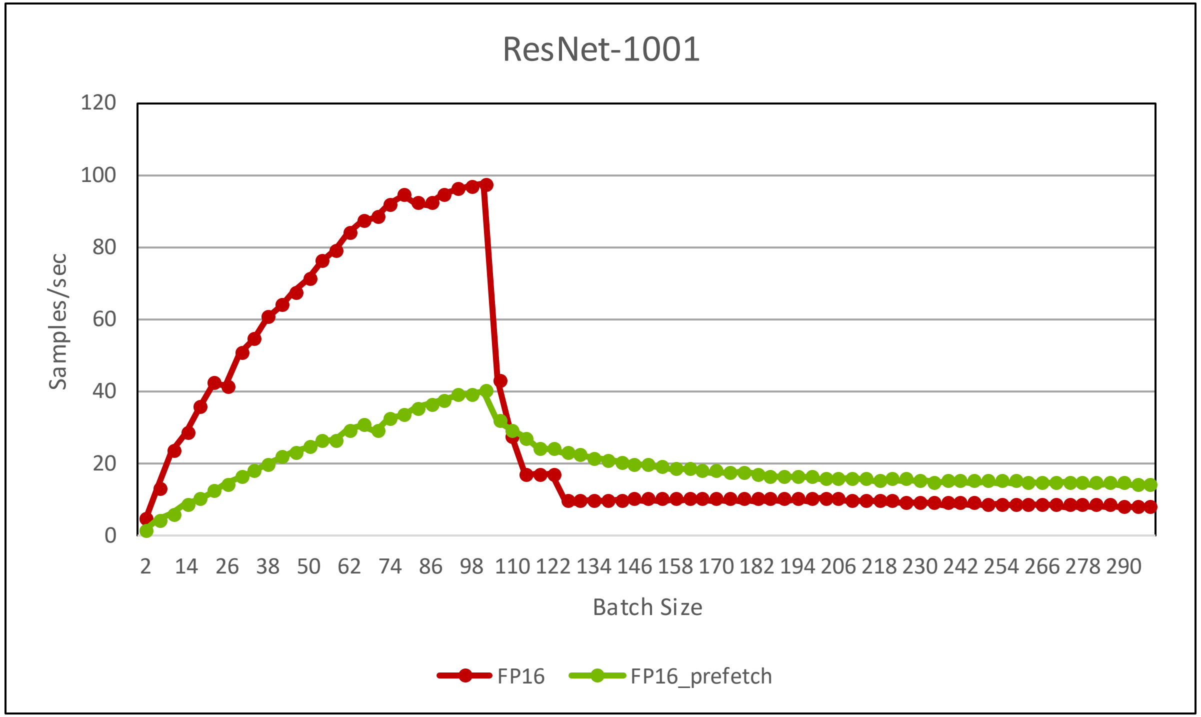
GPU OVERSUBSCRIPTION

Prefetch Only When Needed

Prefetch memory before kernel to improve performance

cudaMemPrefetchAsync takes CPU cycles – degrades performance when not required

Automatic prefetching needed to achieve high performance



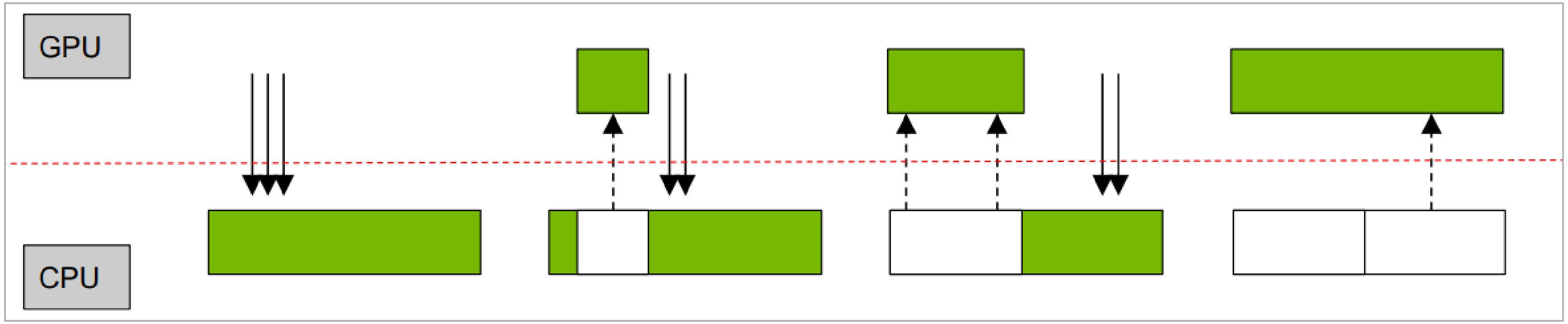
DRIVER PREFETCH

Aggressive driver prefetching

Driver initiated (density) prefetching from CPU to GPU

- GPU pages tracked as chunk of smaller sysmem page
- Driver logic: Prefetch rest of the GPU page when 51% is migrated to GPU
  - Change to 5%

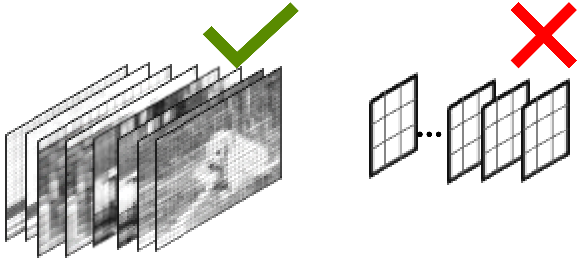
Observe up to 20% gain in performance vs default settings



FRAMEWORK FUTURE

Framework can develop intelligence to insert prefetch before calling GPU kernels

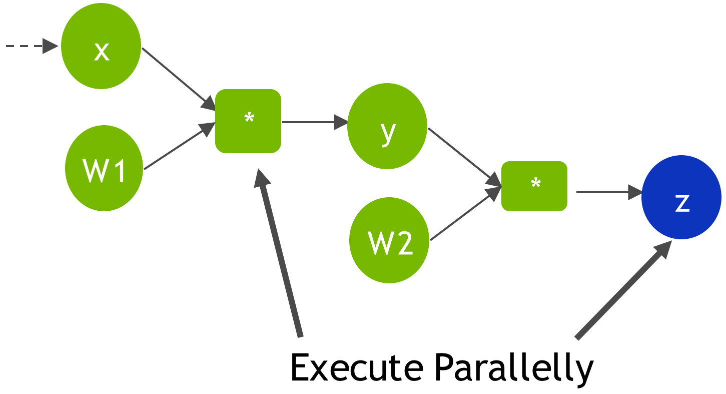
Smart evictions: Activation's only



Lazy Prefetch: Catch kernel calls right before execution and add prefetch calls

```
nn.Conv2d(…)     ← (Hook)

Replace:
nn.Prefetch(…)
nn.Conv2d(…)
```

Eager Prefetch - Identify and add prefetch calls before the kernels are called



TAKEAWAY

Unified Memory oversubscription solves the memory pool fragmentation issue

Simple way to train **bigger models** and on **larger input data**

Minimal user effort, no change in framework programming

Frameworks can get better performance by adding prefetch’s

Try it out and contribute:

https://github.com/rapidsai/cudf

https://github.com/rapidsai/rmm

NVIDIA