# Customize PyTorch memory allocation strategy and introduction to CUDA Stream-ordered Memory Allocator

20230517

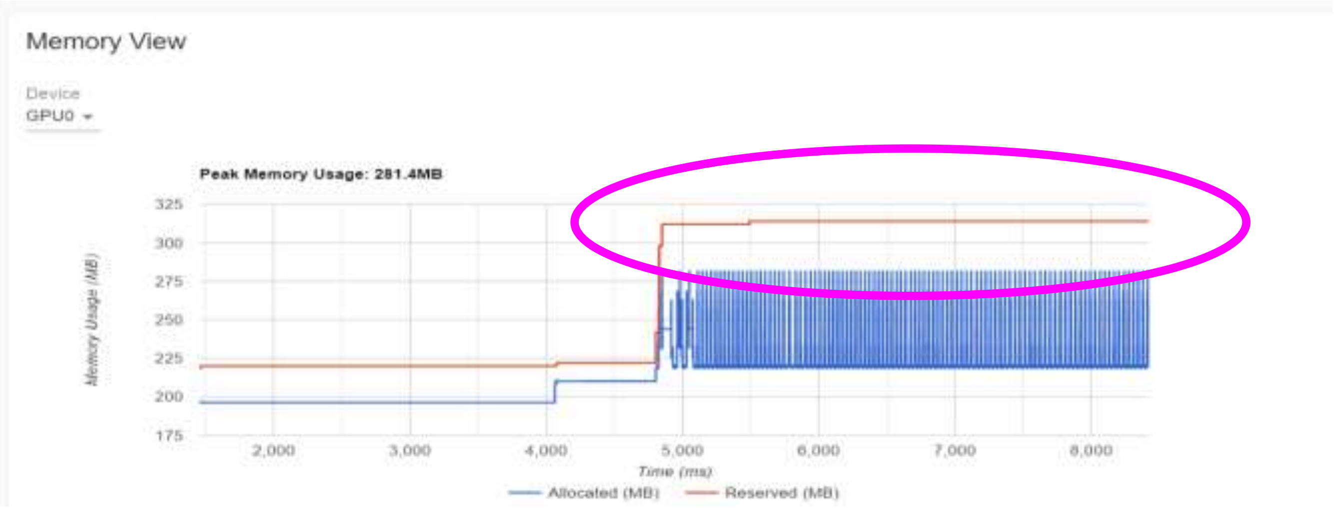
박현우

# Table of Contents

# 1. PyTorch Memory Management

PyTorch uses a caching memory allocator to speed up memory allocations.

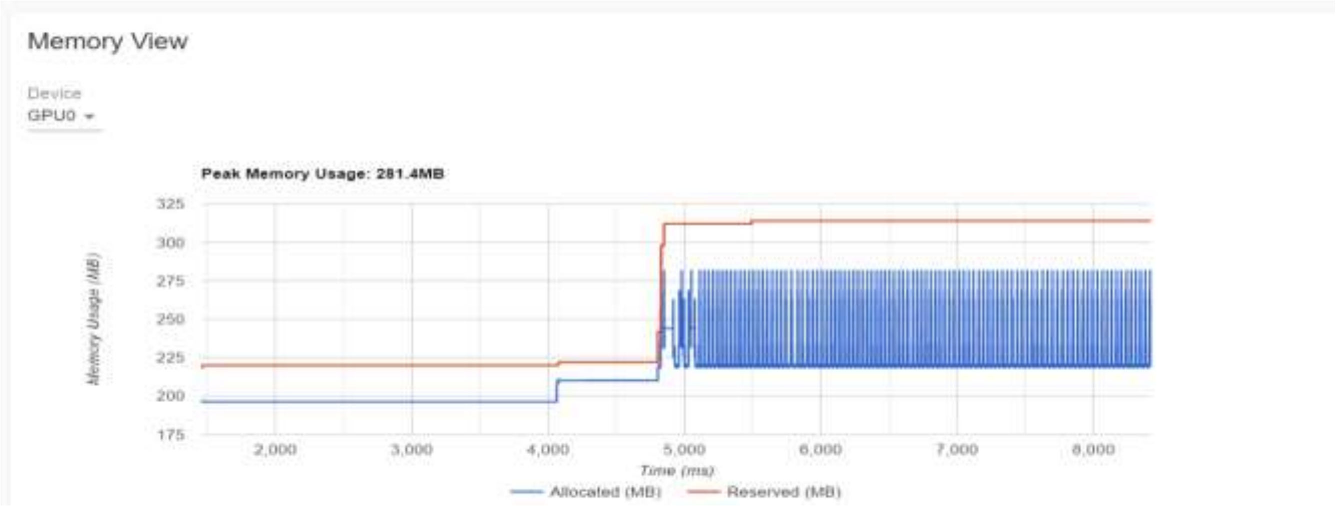
This allows fast memory de/allocation without device synchronizations; D2H, H2D, Device to remote device, etc.



# 1. PyTorch Memory Management

Specifically:

1. Reserved Memory : total amount of memory managed by the caching allocator.
2. Allocated Memory : memory substantially occupied by tensors.



# 1. PyTorch Memory Management

PyTorch currently provides following underlying allocator implementation.

1. native
   (default) which uses PyTorch's native implementation

2. cudaMallocAsync
   which uses CUDA's built-in asynchronous allocator.

Customizing PyTorch memory allocator is only available for cudaMallocAsync.

# 1.1. Customizing PyTorch CUDA Memory Allocator

To define custom CUDA memory allocator,

1. define allocators as simple functions in C/C++
2. and compile them as a shared library.

To apply custom CUDA memory allocator,

supply the path to the .so file and the name of the alloc/free functions (in C/C++) to torch.cuda.memory.CUDAPluggableAllocator.

# 1.1. Customizing PyTorch CUDA Memory Allocator

```
// @ Example for custom allocator
// @ which just traces all the memory operations.

#include <sys/types.h>
#include <cuda_runtime_api.h>
#include <iostream>
// Compile with g++ alloc.cc -o alloc.so -I/usr/local/cuda/include -shared -fPIC
extern "C" {
void* my_malloc(ssize_t size, int device, cudaStream_t stream) {
   void *ptr;
   cudaMalloc(&ptr, size);
   std::cout<<"alloc "<<ptr<<size<<std::endl;
   return ptr;
}

void my_free(void* ptr, ssize_t size, int device, cudaStream_t stream) {
   std::cout<<"free "<<ptr<< " "<<stream<<std::endl;
   cudaFree(ptr);
}
}
```

# 1.1. Customizing PyTorch CUDA Memory Allocator

```python
import torch

# Load the allocator
new_alloc = torch.cuda.memory.CUDAPluggableAllocator(
    'alloc.so', 'my_malloc', 'my_free')
# Swap the current allocator
torch.cuda.memory.change_current_allocator(new_alloc)
# This will allocate memory in the device using the new allocator
b = torch.zeros(10, device='cuda')
```

# 2. CUDA Stream

By default, GPU operations are asynchronous :

When you call a function that uses the GPU, the operations are

1. enqueued to the particular device,
2. but not necessarily executed immediately.

This allows us to execute more computations in parallel.

Therefore, to maximize GPU performance, need to maximize asynchronicity of GPU.

# 2.1. Asynchronicity of GPU ops

However, some ops are synchronous, making the entire device blocked for sync;

e.g. ops involving time (as time measurements without synchronizations are not accurate) or copy

Such synchronization of ops greatly deteriorates the performance of GPU.

# 2.2. Explicitly using non_blocking argument

Several functions (such as to(), copy_()) admit an explicit non_blocking argument, which lets the caller bypass synchronization when it is unnecessary.

(Internally, leveraging copy engine alongside kernel engine.)

Under the hood, PyTorch automatically performs necessary synchronization when copying data

1. between CPU and GPU
2. or between two GPUs.

# 2.2. Explicitly using non_blocking argument

But

1. when customizing, managing such functions is responsible for users,

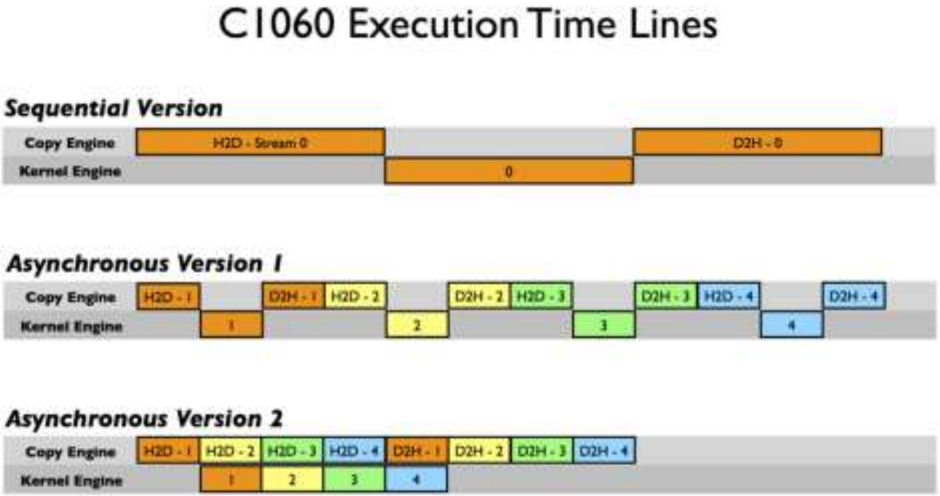
```
for i, (images, target) in enumerate(train_loader):
    # measure data loading time
    data_time.update(time.time() - end)

    if args.gpu is not None:
        images = images.cuda(args.gpu, non_blocking=True)
    if torch.cuda.is_available():
        target = target.cuda(args.gpu, non_blocking=True)

    # compute output
    output = model(images)
    loss = criterion(output, target)
```

# 2.2. Explicitly using non_blocking argument

But

2.            can’t fine-grain control the behavior, thus, there may be no much gain.

(e.g. re-ordering block of sub-ops to overlap.)



# 2.3. CUDA Stream

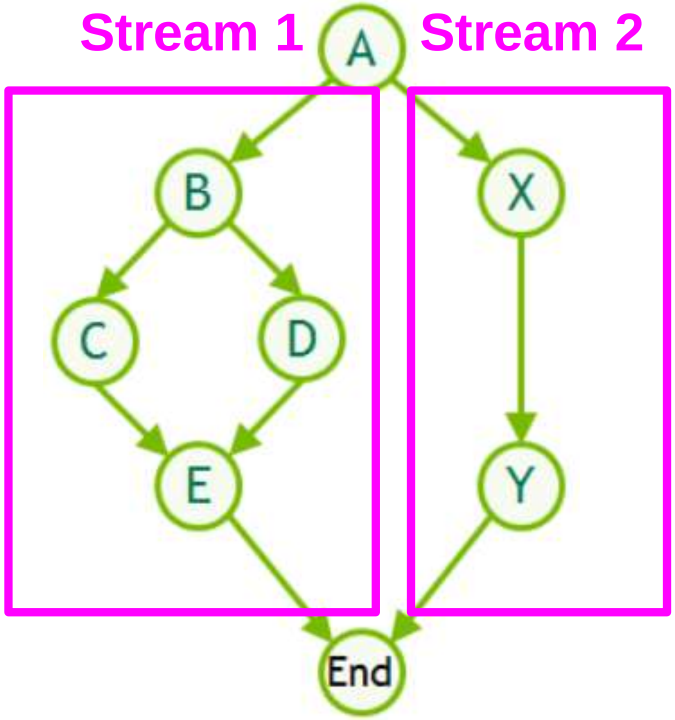
Another exception for such synchronous ops is CUDA stream.

A CUDA stream is a linear sequence of execution that belongs to a specific device.

1. Operations inside each stream are serialized in the order they are created
2. Operations can execute concurrently from different streams in any relative order,

   unless explicit synchronization functions are used.
   (such as synchronize() or wait_stream())

# 2.3. CUDA Stream




From computation graph (CUDA Graph),

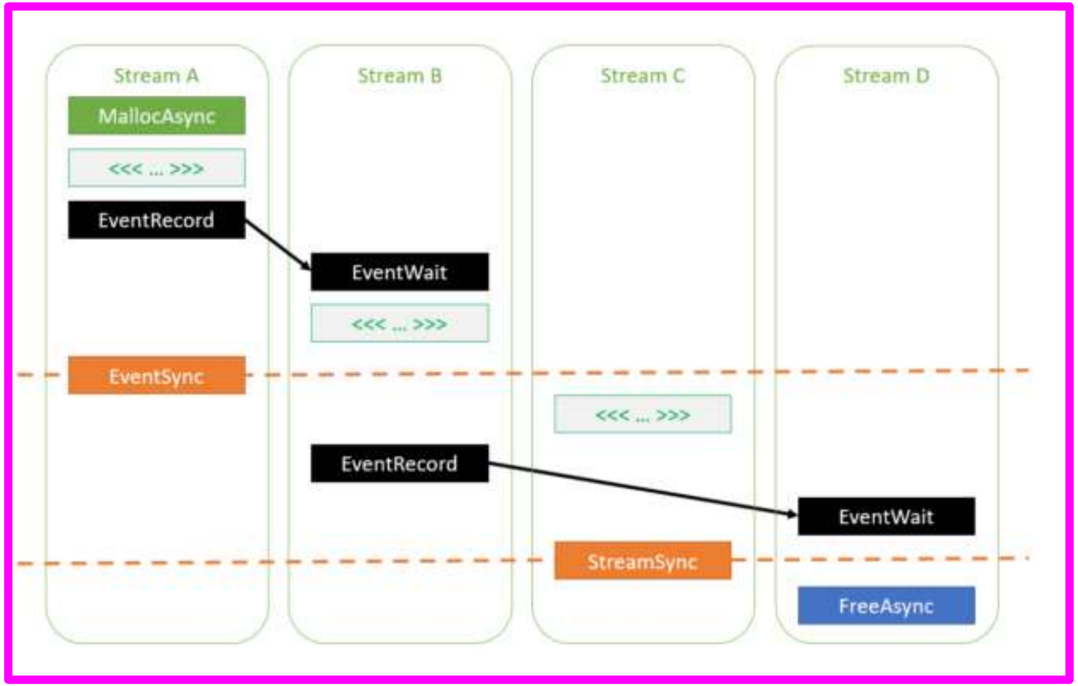
1. Stream 1 and Stream 2 are captured
2. and are executed concurrently.

NOTE : For simplicity, assume using single device.

(But the same logic can be extended to multi-gpu and multi-machine)

# 2.3. CUDA Stream

**Single GPU**



# 2.3. CUDA Stream

```
# warmup
# Uses static_input and static_target here for convenience,
# but in a real setting, because the warmup includes optimizer.step()
# you must use a few batches of real data.
s = torch.cuda.Stream()
s.wait_stream(torch.cuda.current_stream())
with torch.cuda.stream(s):
    for i in range(3):
        optimizer.zero_grad(set_to_none=True)
        y_pred = model(static_input)
        loss = loss_fn(y_pred, static_target)
        loss.backward()
        optimizer.step()
torch.cuda.current_stream().wait_stream(s)
```

# 3. Background for CUDA Stream-ordered Memory allocator

Problem: cudaMalloc and cudaFree API functions, which are used to allocate/release GPU accessible memory, are not stream ordered.

```
cudaMalloc(&ptrA, sizeA);
// @ CUDA C extension
// @ << ... >>> : Kernel invocation
// @ e.g. <<numBlocks, threadsPerBlock>>(input, output)
kernelA<<<..., stream>>>(ptrA);
cudaFree(ptrA); // Synchronizes the device before freeing memory

cudaMalloc(&ptrB, sizeB);

kernelB<<<..., stream>>>(ptrB);
cudaFree(ptrB);
```

This is inefficient

because the first cudaFree call has to wait for kernelA to finish,

so it synchronizes the device before freeing the memory.

# 3.1. Problem: Synchronous cudaMalloc and cudaFree

```
cudaMalloc(&ptr,   max(sizeA, sizeB));
kernelA<<<...,   stream>>>(ptr);
kernelB<<<...,   stream>>>(ptr);
cudaFree(ptr);
```

To make this run more efficiently,

the memory can be

1. allocated upfront
2. and sized to the larger of the two sizes.

So, kernelB now no needs to wait due to synchronization by cudaFree.

# 3.1. Problem: Synchronous cudaMalloc and cudaFree

Problem of mass allocating upfront :

1. Hard to predict when and which kernels are allocated together.
2. Unable to free partially.

2. causes data/library to hold on to the memory longer than it needs to, wasting the space.

# 3.1. Problem: Synchronous cudaMalloc and cudaFree

Some applications take the idea of allocating memory upfront even further by implementing their own custom allocator.

(e.g. when memory de/allocation is very predictable)

This adds a significant amount of complexity to application development.

CUDA aims to provide a low-effort, high-performance alternative.

# 4. CUDA Stream-ordered Memory allocator

CUDA 11.2 introduced a stream-ordered memory allocator to solve these types of problems,

with the addition of cudaMallocAsync and cudaFreeAsync.

These new API functions shift memory de/allocation

from global-scope operations that synchronize the entire device

to stream-ordered operations that is confined to target stream.

# 4. CUDA Stream-ordered Memory allocator

```
cudaMallocAsync(&ptrA, sizeA, stream);
kernelA<<<..., stream>>>(ptrA);
cudaFreeAsync(ptrA, stream); // No synchronization necessary

cudaMallocAsync(&ptrB, sizeB, stream); // Can reuse the memory freed previously
kernelB<<<..., stream>>>(ptrB);
cudaFreeAsync(ptrB, stream);
```

# 4.1. CUDA Memory Pools

The stream-ordered memory allocator introduces the concept of memory pools to CUDA.

A memory pool is a collection of previously allocated memory that can be reused for future allocations.

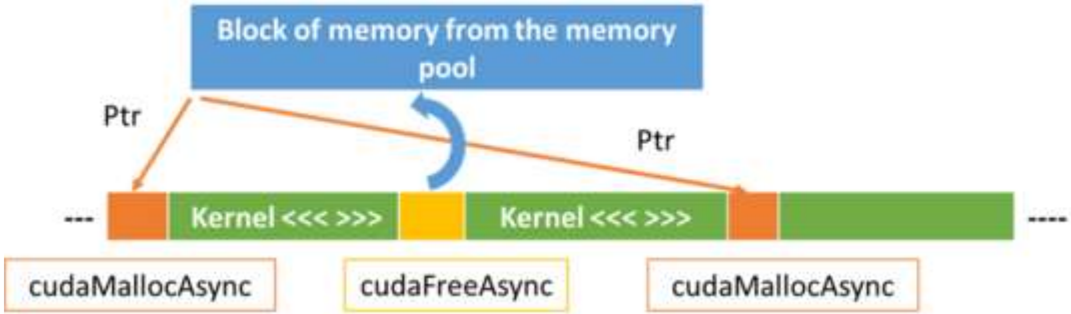
Streams can have their own memory pool or share one with others.

(Device has default memory pool.)

# 4.1. CUDA Memory Pools

Each call to cudaMallocAsync attempts to allocate memory from that device's current pool.

Each call to cudaFreeAsync returns memory to the pool,
which is then available for re-use on subsequent cudaMallocAsync requests.



# 4.2. Managing Memory pool

Principle : System call to OS for memory request and return is considered very expensive.

By default,

1. If the pool has insufficient memory, the CUDA driver calls into the OS to allocate more memory.
2. Unused memory in the pool is returned to the OS during the next synchronization operation.

# 4.3. Release threshold

The application can configure a release threshold to enable unused memory to persist beyond the synchronization operation.

By default, the release threshold is zero.

i.e. all unused memory in the pool is released back to the OS during every synchronization operation.

# 4.3. Release threshold

```
for (int i = 0; i < 100; i++) {
    cudaMallocAsync(&ptr, size, stream);
    kernel<<<..., stream>>>(ptr);
    cudaFreeAsync(ptr, stream);
    cudaStreamSynchronize(stream);
}
```

In this example, release happens at the end of every iteration.

As a result,

1. there is no memory to reuse for the next cudaMallocAsync call
2. and instead memory must be allocated through an expensive system call.

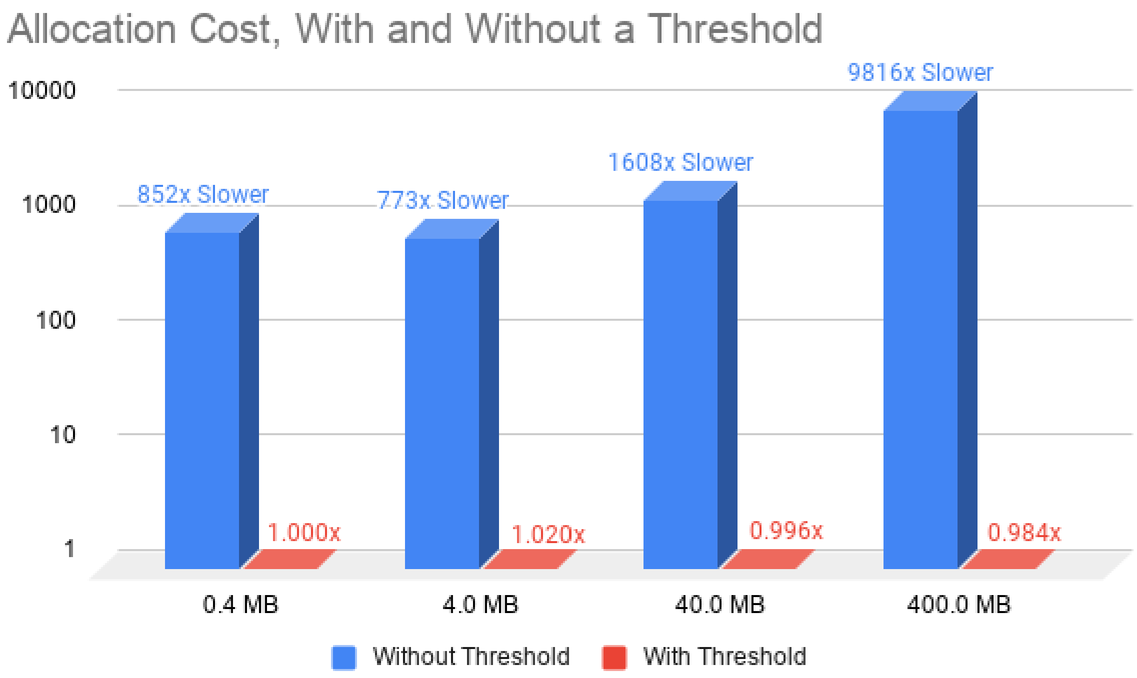
# 4.3. Release threshold

Therefore, good memory allocator (or manager) should search for “good” release threshold.

Example : When using maximum release threshold. (result in next)

```
cudaMemPool_t mempool;
cudaDeviceGetDefaultMemPool(&mempool, device);
uint64_t threshold = UINT64_MAX;
cudaMemPoolSetAttribute(mempool, cudaMemPoolAttrReleaseThreshold, &threshold);

for (int i = 0; i < 100; i++) {
    cudaMallocAsync(&ptr, size, stream);
    kernel<<<..., stream>>>(ptr);
    cudaFreeAsync(ptr, stream);
    cudaStreamSynchronize(stream);    // Only releases memory down to “threshold” bytes
}
```

# 4.3. Release threshold



# 5.1. Cross-GPU ops in PyTorch CUDA

Cross-GPU operations are not allowed by default,

with the exception of

1. copy_()
2. and other methods with copy-like functionality such as to() and cuda().

Unless you enable peer-to-peer memory access,

any attempts will raise an error;

e.g. to launch ops on tensors spreaded across different devices.

# 5.1. Cross-GPU ops in PyTorch CUDA

```python
# @ There are in total 3 GPUs on the machine.
cuda = torch.device('cuda')     # Default CUDA device
cuda0 = torch.device('cuda:0')  # GPU 0
cuda2 = torch.device('cuda:2')  # GPU 2 (these are 0-indexed)

x = torch.tensor([1., 2.], device=cuda0)
# x.device is device(type='cuda', index=0)
y = torch.tensor([1., 2.]).cuda()
# y.device is device(type='cuda', index=0)
```

```python
with torch.cuda.device(1):  # GPU 1
    # allocates a tensor on GPU 1
    a = torch.tensor([1., 2.], device=cuda)

    # transfers a tensor from CPU to GPU 1
    b = torch.tensor([1., 2.]).cuda()
    # a.device and b.device are device(type='cuda', index=1)

    # You can also use ``Tensor.to`` to transfer a tensor:
    b2 = torch.tensor([1., 2.]).to(device=cuda)
    # b.device and b2.device are device(type='cuda', index=1)

    c = a + b
    # c.device is device(type='cuda', index=1)

    z = x + y
    # z.device is device(type='cuda', index=0)

    # even within a context, you can specify the device
    # (or give a GPU index to the .cuda call)
    d = torch.randn(2, device=cuda2)
    e = torch.randn(2).to(cuda2)
    f = torch.randn(2).cuda(cuda2)
    # d.device, e.device, and f.device are all device(type='cuda', index=2)

    # @ Without P2P setting,
    # @ following raises exception
    # @ as c is on GPU 1 and z is on GPU 0
    i = c + z  # @ Error!
```

# 5.2. With P2P capability

Accessing the memory from any other device requires the two devices to be peer capable, as reported by cudaDeviceCanAccessPeer.

```
// Allocate Memory
uint32_t* dev_0;
cudaSetDevice(gpuid_0);
cudaMalloc((void**)&dev_0, size);

uint32_t* dev_1;
cudaSetDevice(gpuid_1);
cudaMalloc((void**)&dev_1, size);

//Check for peer access between participating GPUs:
int can_access_peer_0_1;
int can_access_peer_1_0;
cudaDeviceCanAccessPeer(&can_access_peer_0_1, gpuid_0, gpuid_1);
cudaDeviceCanAccessPeer(&can_access_peer_1_0, gpuid_1, gpuid_0);
printf("cudaDeviceCanAccessPeer(%d->%d): %d\n", gpuid_0, gpuid_1, can_access_peer_0_1);
printf("cudaDeviceCanAccessPeer(%d->%d): %d\n", gpuid_1, gpuid_0, can_access_peer_1_0);

if (can_access_peer_0_1 && can_access_peer_1_0) {
    // Enable P2P Access
    cudaSetDevice(gpuid_0);
    cudaDeviceEnablePeerAccess(gpuid_1, 0);
    cudaSetDevice(gpuid_1);
    cudaDeviceEnablePeerAccess(gpuid_0, 0);
}
```

```
// Init Stream
cudaStream_t stream;
cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);

// ~~ Start Test ~~
cudaEventRecord(start, stream);

//Do a P2P memcpy
for (int i = 0; i < repeat; ++i) {
    cudaMemcpyAsync(dev_0, dev_1, size, cudaMemcpyDeviceToDevice, stream);
}
```

# 5.2. With P2P capability

By default, memory allocated using cudaMallocAsync is only accessible from the device associated with the specified stream.

Accessing the memory from any other device requires enabling access to the entire pool from that other device.

However, unlike cudaMalloc allocations,

cudaDeviceEnablePeerAccess and cudaDeviceDisablePeerAccess have no effect on memory allocated from memory pools.

# 5.3. IPC support for stream memory pool

Stream memory pool can be wrapped and exported as an handle.

Such pool handle can be shared and transported through common IPC; e.g. UNIX socket or Pipe, (@ possibly TensorPipe)

```
cudaMemPool_t exportPool;

cudaMemPoolProps poolProps = {};
poolProps.allocType = cudaMemAllocationTypePinned;
poolProps.handleTypes = cudaMemHandleTypePosixFileDescriptor;
poolProps.location.type = cudaMemLocationTypeDevice; // The location type Device
poolProps.location.id = deviceId;

cudaMemPoolCreate(&exportPool, &poolProps);
```

```
int fd;
cudaMemAllocationHandleType handleType = cudaMemHandleTypePosixFileDescriptor;
cudaMemPoolExportToShareableHandle(&fd, exportPool, handleType, 0);
```

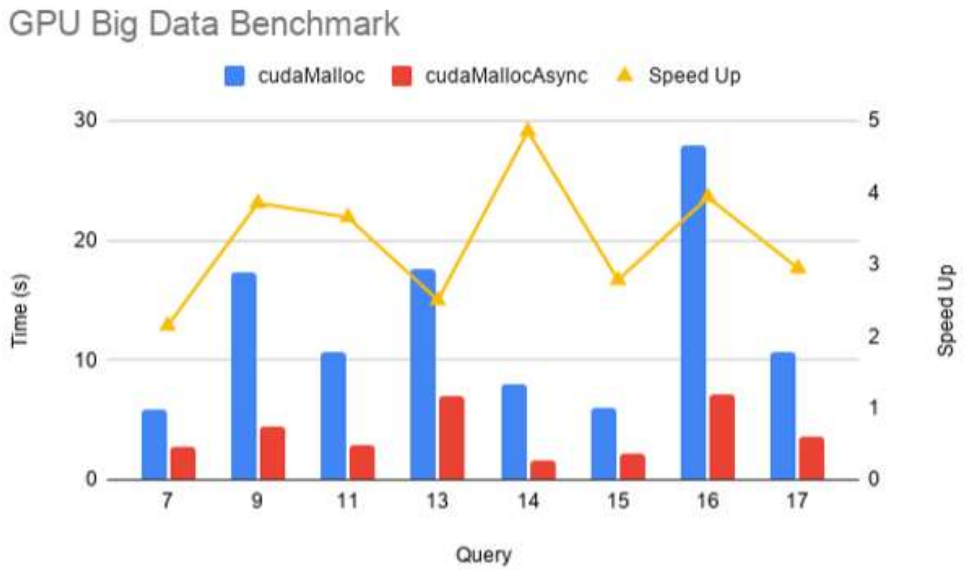
# 6. CUDA Stream-ordered Memory allocator Benchmark

NVIDIA provided benchmark results from the RAPIDS GPU Big Data Benchmark (gpu-bdb).

gpu-bdb is a benchmark of 30 queries representing real-world workflows that can include

1. SQL,
2. user-defined functions,
3. careful subsetting and aggregation,
4. and machine learning.

# 6. CUDA Stream-ordered Memory allocator Benchmark




Thanks to

1. memory reuse (memory pool)
2. and eliminating extraneous synchronization,

there's a 2–5x improvement in end-to-end performance

when using cudaMallocAsync.