Frank de Jong

Frank.Herman.DeJong@gmail.com

Scoring in Plutonium

How to choose the correct hardware generically

Table of Contents

[Memory type 2](#_Toc533890169)

[Memory type property flags 2](#_Toc533890170)

[None 3](#_Toc533890171)

[Device Local 3](#_Toc533890172)

[Host Visible 3](#_Toc533890173)

[Host Coherent 3](#_Toc533890174)

[Host Cached 3](#_Toc533890175)

[Lazily Allocated 3](#_Toc533890176)

[Protected 3](#_Toc533890177)

[Useful properties for the algorithm 4](#_Toc533890178)

[Desired execution 4](#_Toc533890179)

[Ranking 5](#_Toc533890180)

[Physical devices 6](#_Toc533890181)

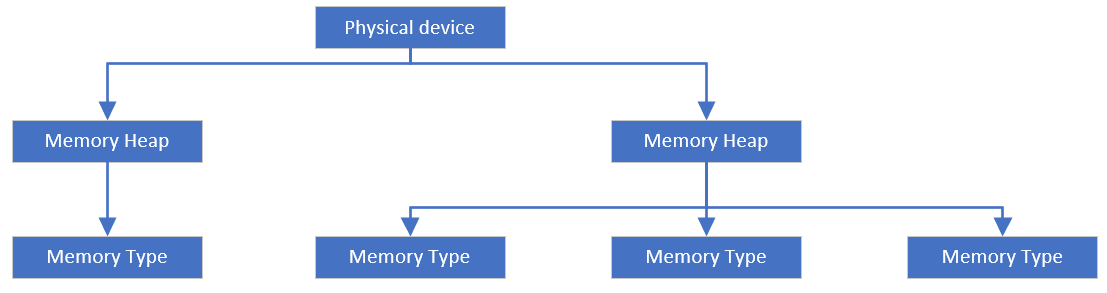
[Types 6](#_Toc533890182)

[Features and limits 6](#_Toc533890183)

[Queue families 6](#_Toc533890184)

[Desired behavior 8](#_Toc533890185)

# Memory type

Whenever a buffer is created, we need to assign it to a memory type on a specific memory heap. A Vulkan compatible graphics device can have multiple heaps available and, on those heaps, multiple memory types.  


The amount of memory heaps depends on the physical device as well as the types. So, we cannot safely pick a memory type at a preset index as that memory type will probably have vastly different properties on another physical device.

For this we need a system that can pick a memory type for a specific buffer usage. This needs to be a memory type that can handle the needs of the buffer and be the most efficient type available.

We can safely ignore the parent memory heap as the only real data that this stores the available bytes.

## Memory type property flags

|  |  |  |
| --- | --- | --- |
| Name | Description | Usefulness |
| None | No flags are set. | High |
| Device Local | Most efficient for device access. | High |
| Host Visible | Can be mapped for host access. | Low |
| Host Coherent | Cache management isn’t needed | Medium |
| Host Cached | Host caches the memory allocated | Medium |
| Lazily Allocated | Memory is allocated on demand | Low |
| Protected | Can be accessed by protected queues | Low |

### None

We know nothing about this memory type, the safest course of action is to just ignore it.

### Device Local

This is memory that is easily accessible by the physical device, this means that it is fast and has the highest change on cache hits, this is thusly a high priority to check.

### Host Visible

The is memory that can be accessed by the physical device and the CPU. This type of memory should only be used for staging resources to more efficient memory types as it’s quite slow.

### Host Coherent

This flag is only set if the memory type is also Host Visible, this additional flag means that we don’t have to flush or invalidate the memory shared by the physical device and the CPU. This makes it even slower than Host Visible as the memory has to always be available to the physical device and the CPU.

### Host Cached

This means the CPU caches the memory allocated with this type. CPU access to this memory is faster fast in this way but, we hardly need this as buffers will probably only be created ones.

### Lazily Allocated

This means the physical device will only allocate the memory once it needs it. this can save some performance in niche environments. For instance, in tile-based rendering. If one tile encapsulates the entire surface, we don’t actually have to copy it to a new buffer we that data will never be allocated with this memory type. This will save some performance.

### Protected

This flag doesn’t affect performance, so we can safely ignore it.

## Useful properties for the algorithm

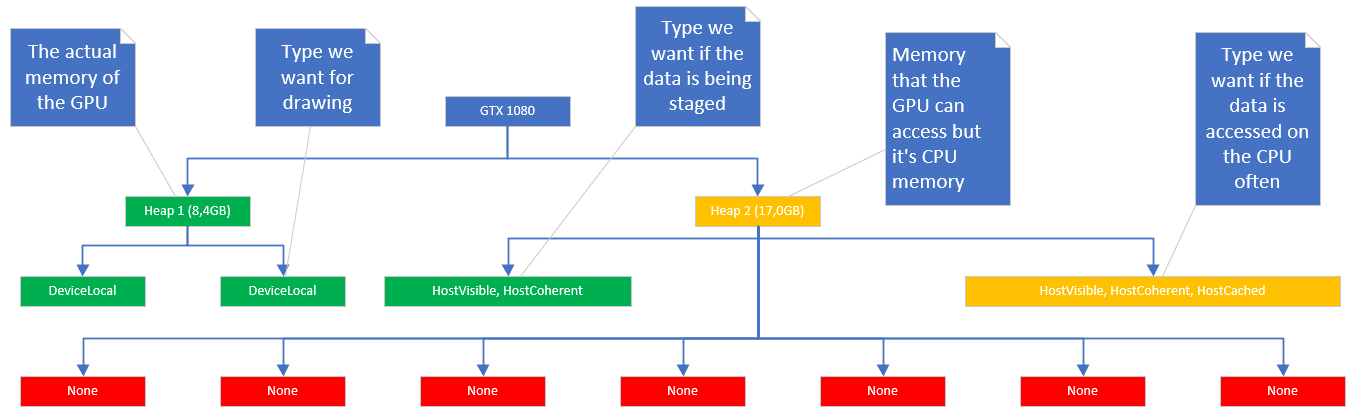
The flags we’ll be using to determine the most optimal memory type will be:

|  |  |  |
| --- | --- | --- |
| Name | Effect | Strength |
| None | Negative | High, fully ignore memory types with no flags |
| Device Local | Positive | High, highly prefer device local memory types |
| Host Visible | Negative | Low, we don’t really want it unless we need it |
| Host Coherent | Negative | Medium, quite slow, we want to ignore it |
| Host Cached | Positive | Low, only useful if CPU buffer |

The amount of memory available on the memory heap should not matter that much as most physical devices will only have one useful memory heap available.

## Desired execution

Here we’ll look at an example of getting the best memory types for two kinds of buffers, one staging buffer and one vertex buffer. This example will go through the available memory heaps and types on a Nvidia GTX 1080.



I colored coded how we want our algorithm to treat all the types and heaps.

* Green: Best suited for our purposes.
* Orange: Should only pick if greens aren’t available
* Red: Never pick these.

For buffers used in drawing commands we want to use any of the two device local memory types of heap 1.   
For staging buffers, we want to use the Host visible, host coherent memory type of heap 2.   
And for buffer that’re accessed by the CPU often we want to use the host visible, host coherent, host cached buffer on heap 2.

### Ranking

#### Vertex buffers (no host access needed)

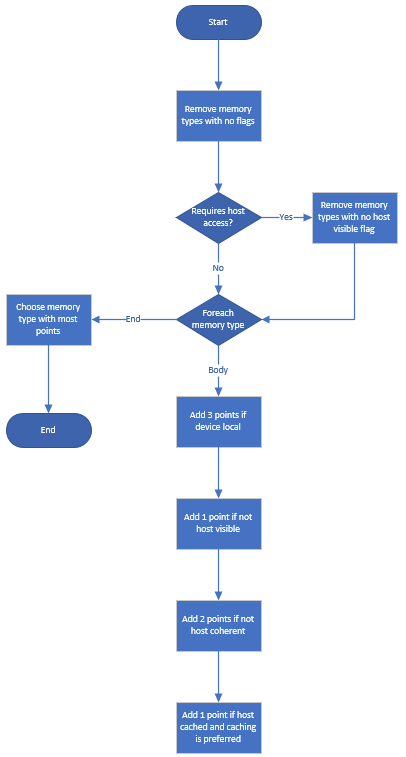
* [Heap 1] Device local (6pts)
* [Heap 1] Device local (6pts)
* [Heap 2] Host visible, host coherent (0pts)
* [Heap 2] Host visible, host coherent, host cached (0pts)

#### Staging buffers (host access required)

* [Heap 2] Host visible, host coherent (0pts)
* [Heap 2] Host visible, host coherent, host cached (0pts)

#### CPU buffers (host access required, caching preferred)

* [Heap 2] Host visible, host coherent, host cached (1pts)
* [Heap 2] Host visible, host coherent (0pts)



# Physical devices

All of our performance intensive command will be running on a Vulkan capable physical device, picking the right one for the job is thusly highly important. Multiple physical devices can be available one a computer and they can very widely in performance and capabilities.

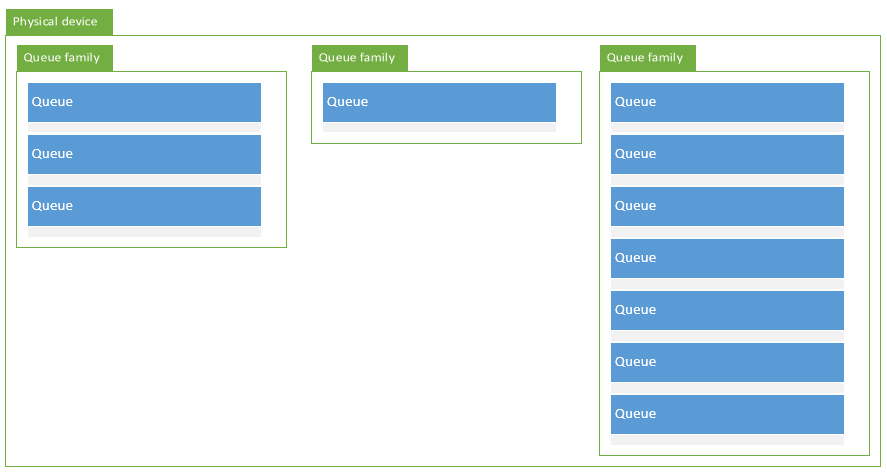
## Types

|  |  |  |
| --- | --- | --- |
| Type | Description | Usefulness |
| Other | Miscellaneous | Low |
| Integrated GPU | Embedded or tightly coupled with the CPU | Medium |
| Discrete GPU | Separate GPU | High |
| Virtual GPU | Virtual node in a virtualization environment | Low |
| CPU | Same processor as the host | Low |

## Features and limits

All physical devices show their features and limits to the host. This means we can eliminate some physical devices by their properties. If we for instance need the physical device to present to a surface. Actually checking for the best GPU this way is however very time consuming and game specific as there are many properties to consider.

## Queue families

The types and number of queues supported on a physical device really dictate which physical device is worth taking and which one isn’t. 

These queue families have some useful properties and flags, the example below shows a Nvidia GTX 1080’s queue families.

|  |  |  |
| --- | --- | --- |
| Name | Description | Usefulness |
| Count | The number of queues that can be created | Medium |
| Graphics | Whether it supports graphics operations | High |
| Compute | Whether it supports compute operations | High |
| Transfer | Whether it supports transfer operations | High |
| Sparse Binding | Whether it supports sparse memory | Low |
| Protected | Whether it supports protected memory | Low |



## Desired behavior

We want to decide on the best physical device based on a few properties shown here, we also want the user to be able to remove physical devices from the list if they don’t suit their needs. For instance, if they want 4K textures but the physical device doesn’t support it.

Checking what kind of queue families are available is also a good measure of performance. Generalist queue families are typically slower at specific operations than their specialized counterparts. In the example we for instance have 16 generalist queues and one transfer specific queue. This queue will most likely be more efficient at data transfer than the generalist queues.

