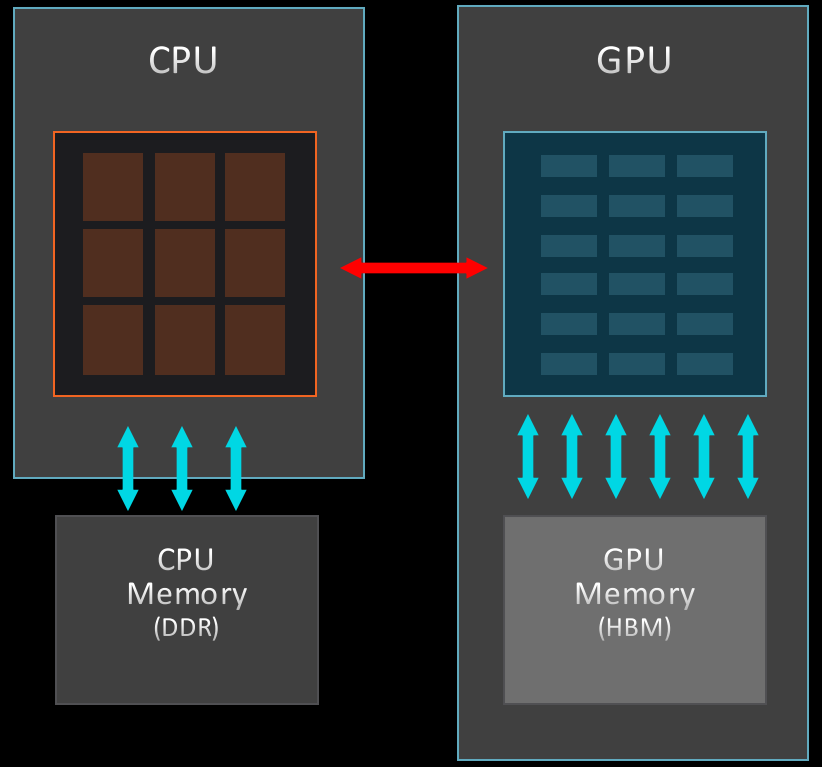
System Memory Allocators

malloc (C/C++)
possix_memalign (C/C++)

new (C/C++)

ALLOCATE (Fortran)

mmap+mbind

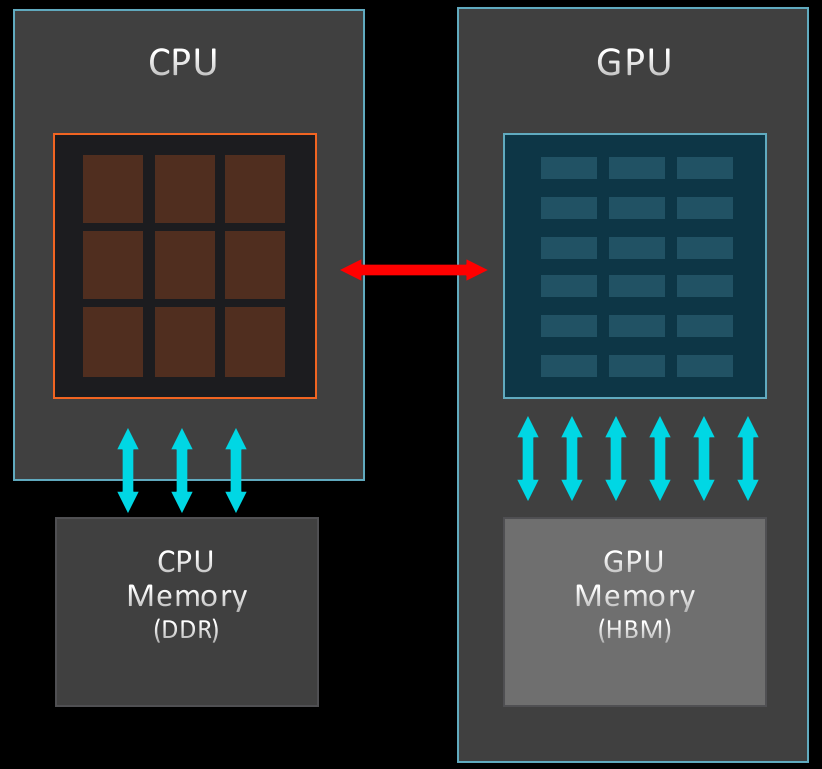


System Memory Allocators: dynamic memory allocations

System memory allocators notify an Operating System that at some point of time N bytes of memory may be required.

The actual allocation of a segment of physical memory happens at the first touch.

```
Type * data;
data = new Type[N];
for (size_t i = 0; i < N; ++i) data[i] = 0.0;
```

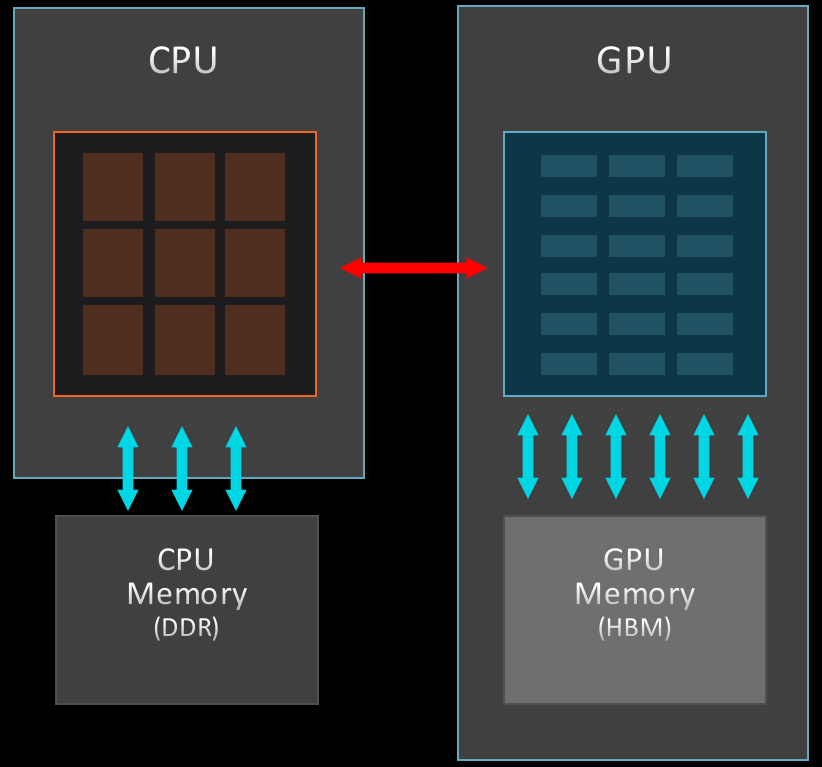


System Memory Allocators: malloc

malloc (C/C++)

https://man7.org/linux/man-pages/man3/malloc.3.html

*"By default, Linux follows an optimistic memory allocation strategy. This means that when **malloc**() returns non-NULL there is no guarantee that the memory really is available. In case it turns out that the system is out of memory, one or more processes will be killed by the OOM killer."*




```
Type * data = (Type*) malloc (N*sizeof(Type));
for (size_t i = 0; i < N; ++i) data[i] = 0.0;
```

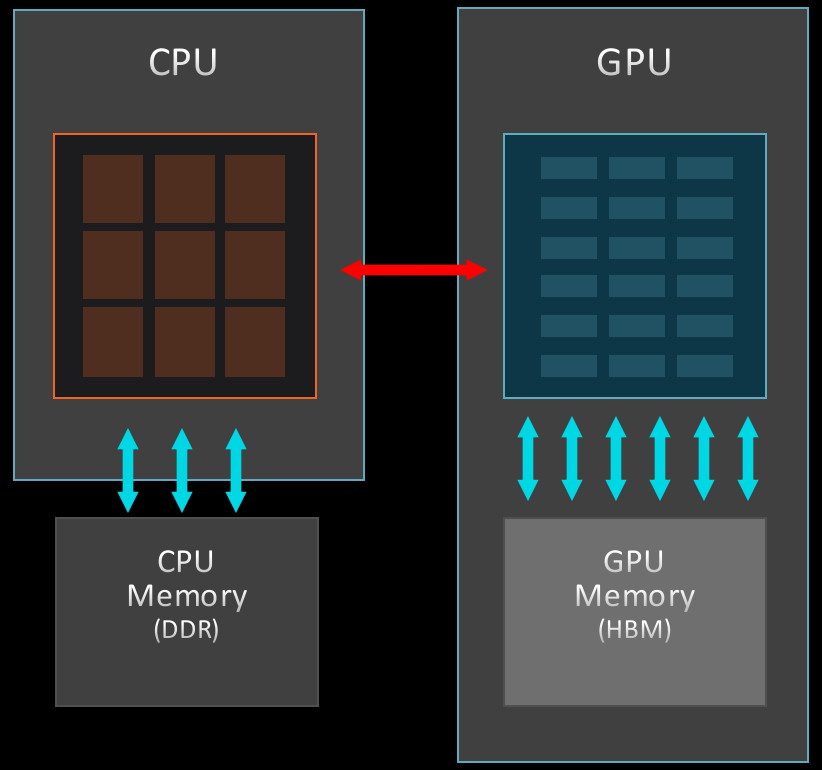
System Memory Allocators: new

“new” (C++)
notifies the OS that at some point N*sizeof(Type) bytes of memory will be required.
If memory is requested for an object with a constructor that initializes the object data physical memory allocation will also occur.

https://en.cppreference.com/w/cpp/language/new

```
#include <new>

Type * data = new Type[N];
for (size_t i = 0; i < N; ++i) data[i] = 0.0;
```



System Memory Allocators: placement new

“placement new” (C++)

“placement new“ is used to construct objects in allocated storage:

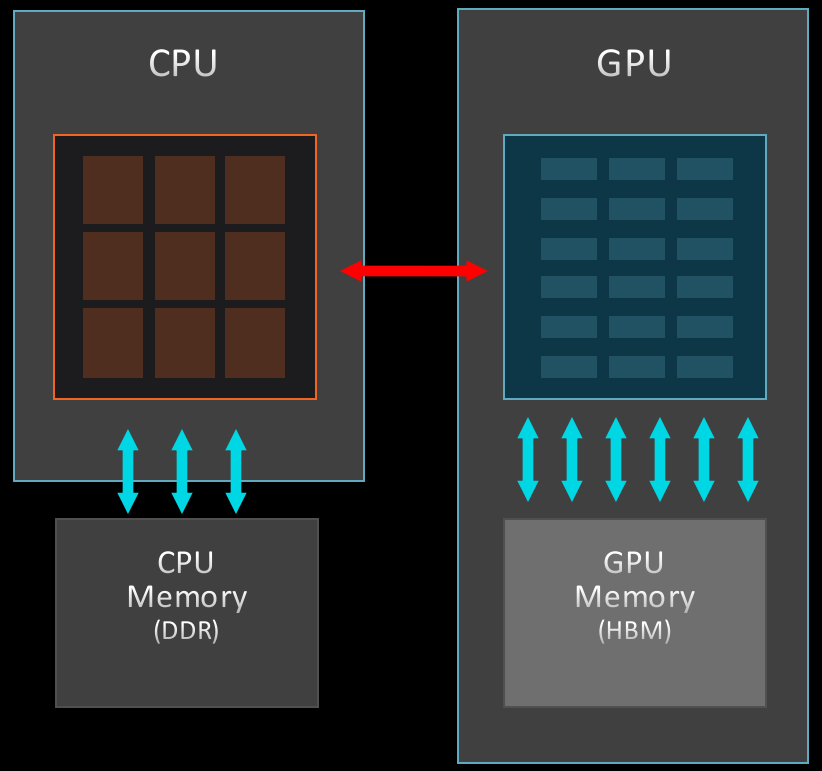
https://en.cppreference.com/w/cpp/language/new#Placement_new

```
#include <new>

void * ptr = some_memory_allocator (N*sizeof(double));
double *data = new(ptr)  double[N] ;


for (size_t i = 0; i < N; ++i) data[i] = 0.0;
```

Note: memory deallocation must be done consistently with allocator used in “some_memory_allocator” function



Use cases:
reuse previous memory allocations (memory pool)
construct objects in “CUDA Managed” memory

System Memory Allocators: new with memory alignment

“new” (C++17)

default memory alignment is defined in __STDCPP_DEFAULT_NEW_ALIGNMENT__

and it is typically 16 bytes

The value stored in __STDCPP_DEFAULT_NEW_ALIGNMENT__
can be controlled via a compilation flag:

-fnew-alignment=<align> Specifies the largest alignment guaranteed by '::operator new(size_t)’

g++ -std=c++17 -fnew-alignment=256 my_program.cpp

https://www.cppstories.com/2019/08/newnew-align/

To view the compiler arguments related to “new”:

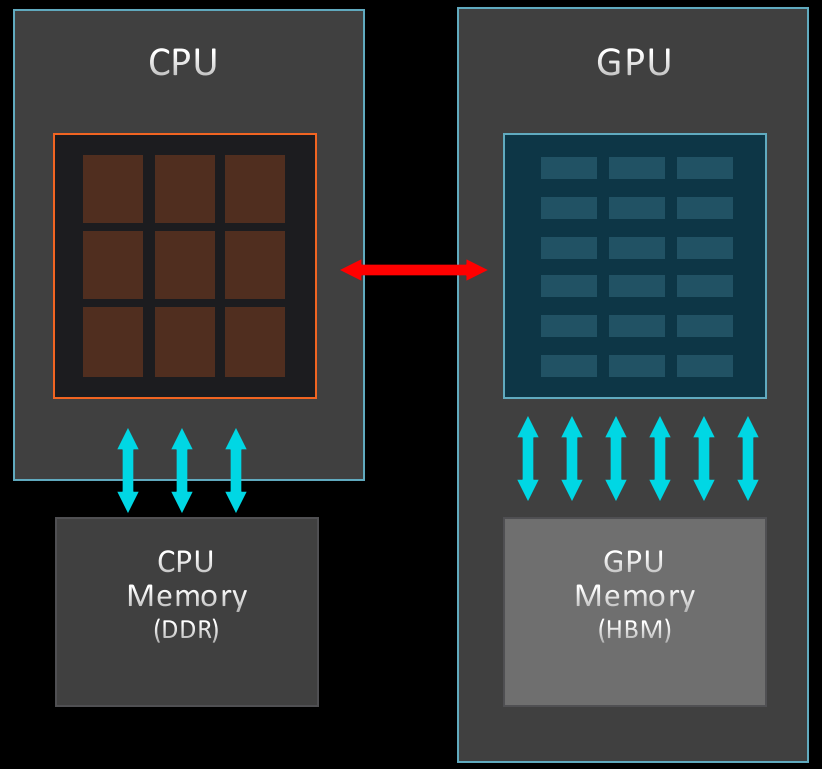
g++ -std=c++17 --help | grep new

```
#include <new>

double *X, *Y;

X = new (std::align_val_t(128)) double[N];
Y = new (std::align_val_t(__STDCPP_DEFAULT_NEW_ALIGNMENT__)) double[N];

for (size_t i = 0; i < N; ++i) data[i] = 0.0;
```



Use cases:
use of system memory for computation on GPUs

Memory Allocators for computing with GPUs

https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html#group__CUDART__MEMORY

cudaError_t cudaMalloc (void** devPtr, size_t size)

cudaError_t cudaFree (void* devPtr)

##

cudaError_t cudaHostAlloc (void** pHost, size_t size, unsigned int flags)

(Allocates page-locked memory on the host)

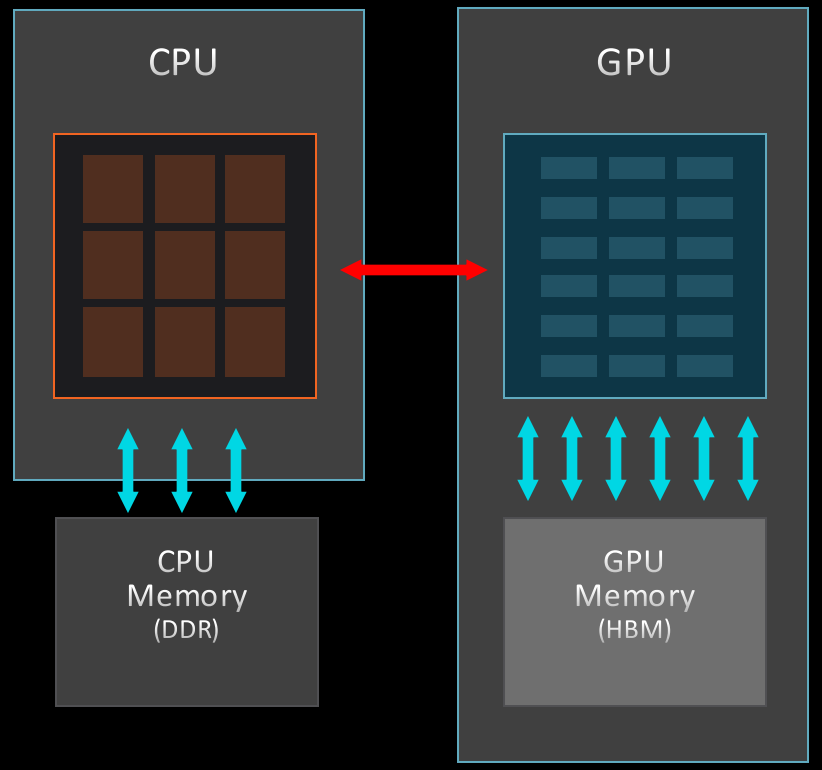
cudaError_t cudaFreeHost (void* ptr)

##

cudaError_t cudaMallocManaged (void** devPtr, size_t size)

cudaError_t cudaFree (void* devPtr)

##

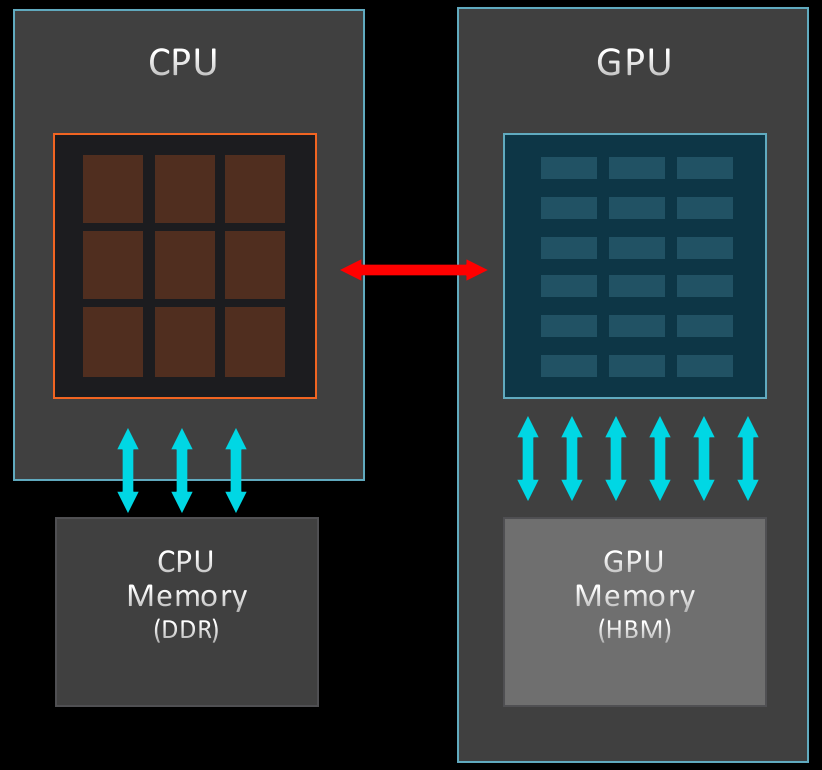


Device memory allocators are actually allocating physical memory

Memory Allocators for computing with GPUs

Device memory allocators are actually allocating physical memory

The good: memory is ready to use at full speed
The bad (maybe) : memory allocation consumes many cycles …
(i.e. needs to be taken into an account when analyzing the application performance)



Memory Allocators for computing with GPUs

https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html#group__CUDART__MEMORY

cudaError_t cudaHostAlloc (void** pHost, size_t size, unsigned int flags)

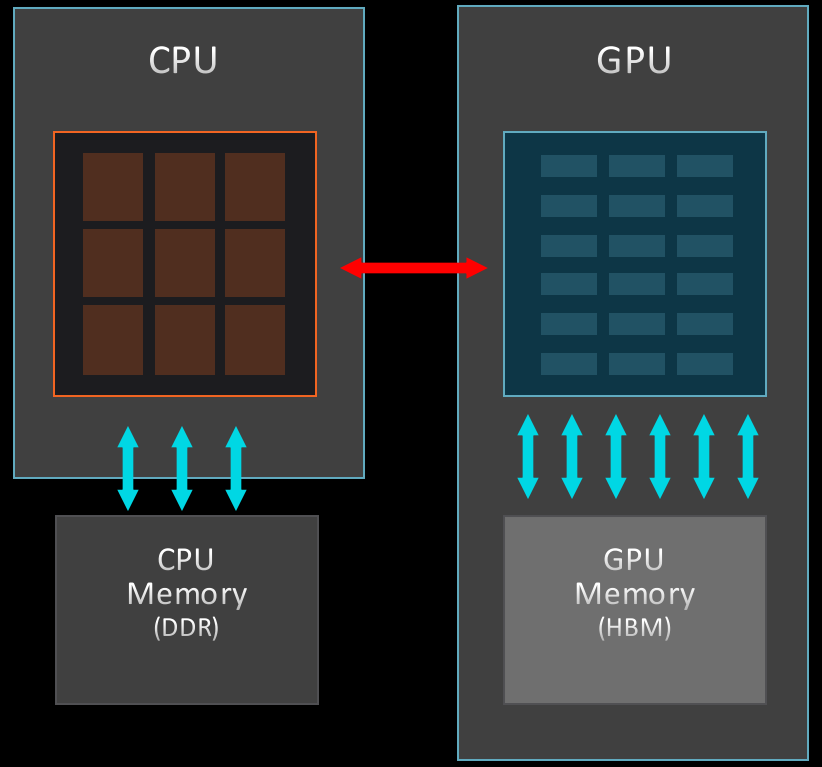
(Allocates page-locked memory on the host)

cudaError_t cudaFreeHost (void* ptr)

```
// Allocate host memory using CUDA allocation calls
cudaHostAlloc((void **)&h_in, sizeIn, cudaHostAllocMapped);
cudaHostAlloc((void **)&h_out, sizeOut, cudaHostAllocMapped);

// Device arrays (CPU pointers) float *d_out, *d_in;
// Get device pointer from host memory. No allocation or memcpy
cudaHostGetDevicePointer((void **)&d_in, (void *) h_in , 0);
cudaHostGetDevicePointer((void **)&d_out, (void *) h_out, 0);

// Launch the GPU kernel
kernel<<<blocks, threads>>>(d_out, d_in);
```

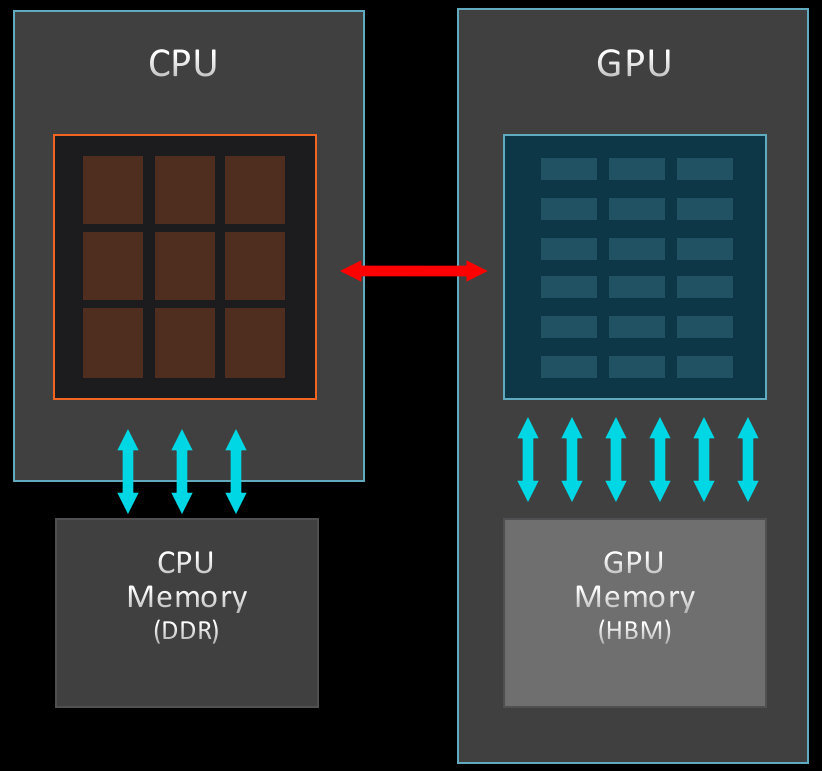


Memory Allocators for computing with GPUs

```
// Allocate host memory using CUDA allocation calls
cudaHostAlloc((void **)&h_in, sizeIn, cudaHostAllocMapped);
cudaHostAlloc((void **)&h_out, sizeOut, cudaHostAllocMapped);

// Device arrays (CPU pointers) float *d_out, *d_in;
// Get device pointer from host memory. No allocation or memcpy
cudaHostGetDevicePointer((void **)&d_in, (void *) h_in , 0);
cudaHostGetDevicePointer((void **)&d_out, (void *) h_out, 0);

// Launch the GPU kernel
kernel<<<blocks, threads>>>(d_out, d_in);
```



Use cases for cudaHostAlloc:

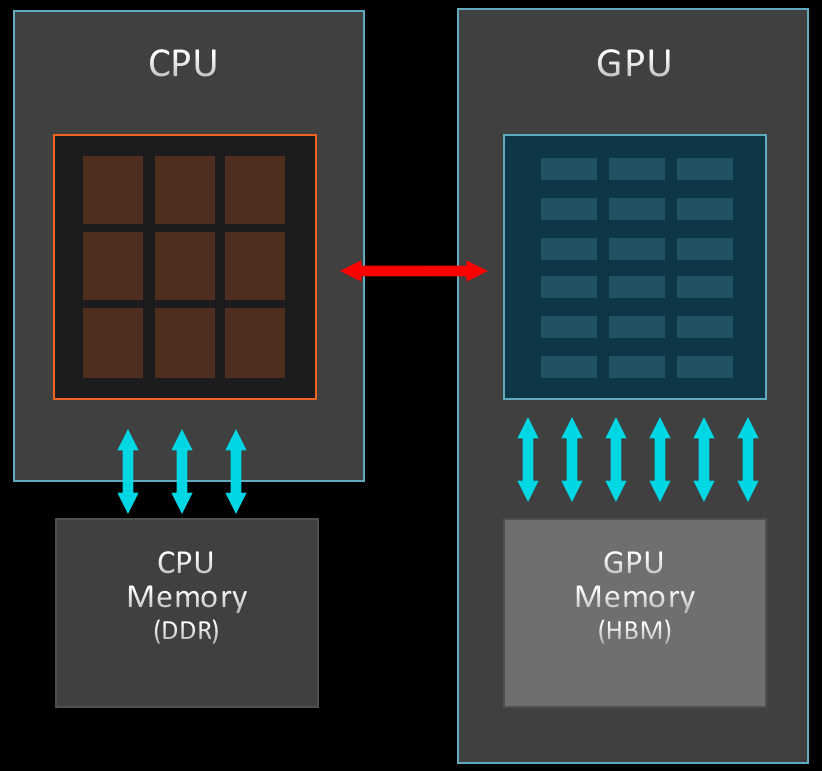
1. Threads running on GPU can access the data stored in the CPU memory
2. cudaMemcpy between the HOST and DEVICE memories are much faster

Memory Allocators for computing with GPUs

```
// Allocate host memory using CUDA allocation calls
cudaHostAlloc((void **)&h_in, sizeIn, cudaHostAllocMapped);
cudaHostAlloc((void **)&h_out, sizeOut, cudaHostAllocMapped);

// Device arrays (CPU pointers) float *d_out, *d_in;
// Get device pointer from host memory. No allocation or memcpy
cudaHostGetDevicePointer((void **)&d_in, (void *) h_in , 0);
cudaHostGetDevicePointer((void **)&d_out, (void *) h_out, 0);

// Launch the GPU kernel
kernel<<<blocks, threads>>>(d_out, d_in);
```



Exercise:
write a code performing dot product of two vectors, each thread-block will store partial results into HOST memory allocated with cudaHostAlloc . Complete the reduction by CPU cores accessing data from the same cudaHostAlloc'ed memory

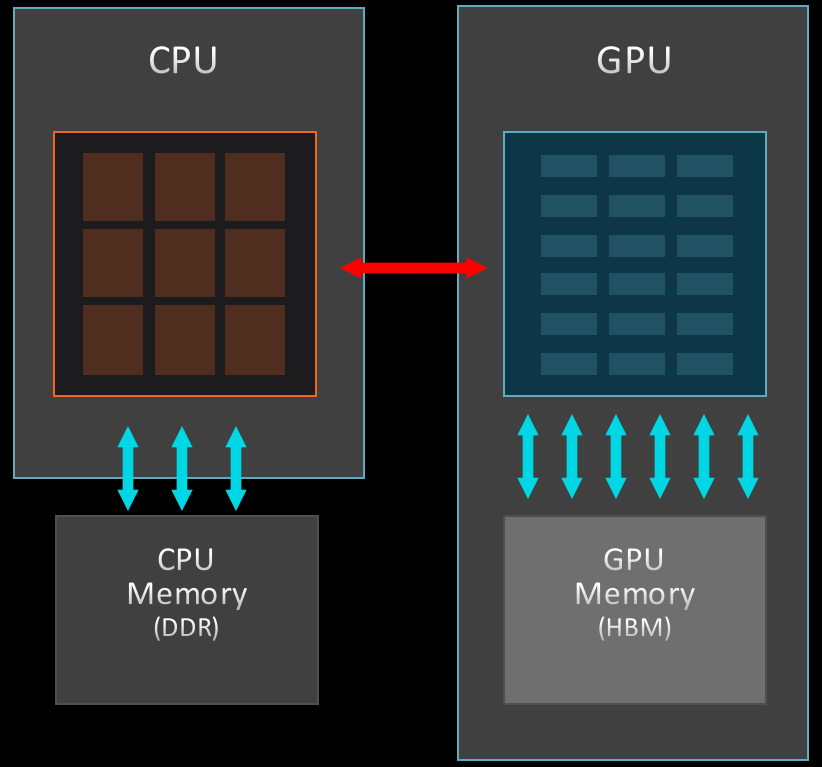
Memory Allocators for computing with GPUs

cudaError_t cudaMallocManaged (void** devPtr, size_t size, unsigned int flags = cudaMemAttachGlobal)

https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__HIGHLEVEL.html#group__CUDART__HIGHLEVEL_1gcf6b9b1019e73c5bc2b39b39fe90816e

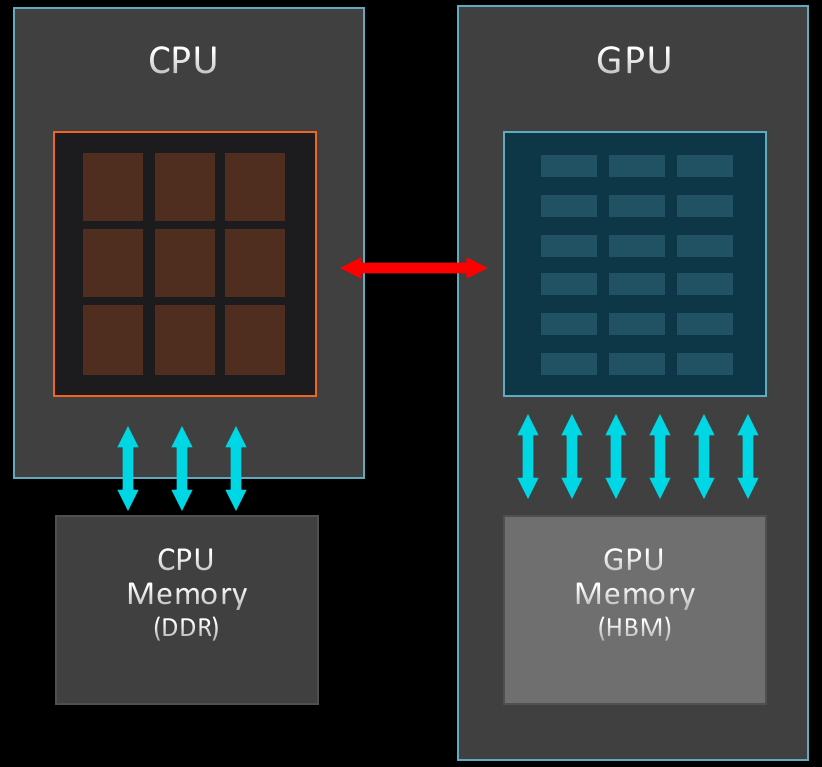
https://developer.nvidia.com/blog/unified-memory-cuda-beginners/

Allocates size bytes of managed memory on the device and returns in *devPtr a pointer to the allocated memory. If the device doesn't support allocating managed memory, cudaErrorNotSupported is returned. Support for managed memory can be queried using the device attribute cudaDevAttrManagedMemory. The allocated memory is suitably aligned for any kind of variable. The memory is not cleared. If size is 0, cudaMallocManaged returns cudaErrorInvalidValue. The pointer is valid on the CPU and on all GPUs in the system that support managed memory.



Memory Allocators for computing with GPUs: CUDA Managed Memory

```
1.int main(void)
2.{
3.int N = 1<<20;
4.float *x, *y;
5.
6.// Allocate Unified Memory -- accessible from CPU or GPU
7.cudaMallocManaged(&x, N*sizeof(float));
8.cudaMallocManaged(&y, N*sizeof(float));
9.
10.// initialize x and y arrays on the host
11.for (int i = 0; i < N; i++) {
12.x[i] = 1.0f;
13.y[i] = 2.0f;
14.}
15.
16.// Launch kernel on 1M elements on the GPU
17.int blockSize = 256;
18.int numBlocks = (N + blockSize - 1) / blockSize;
19.add<<<numBlocks, blockSize>>>(N, x, y);
20.
21.// Wait for GPU to finish before accessing on host
22.cudaDeviceSynchronize();
```

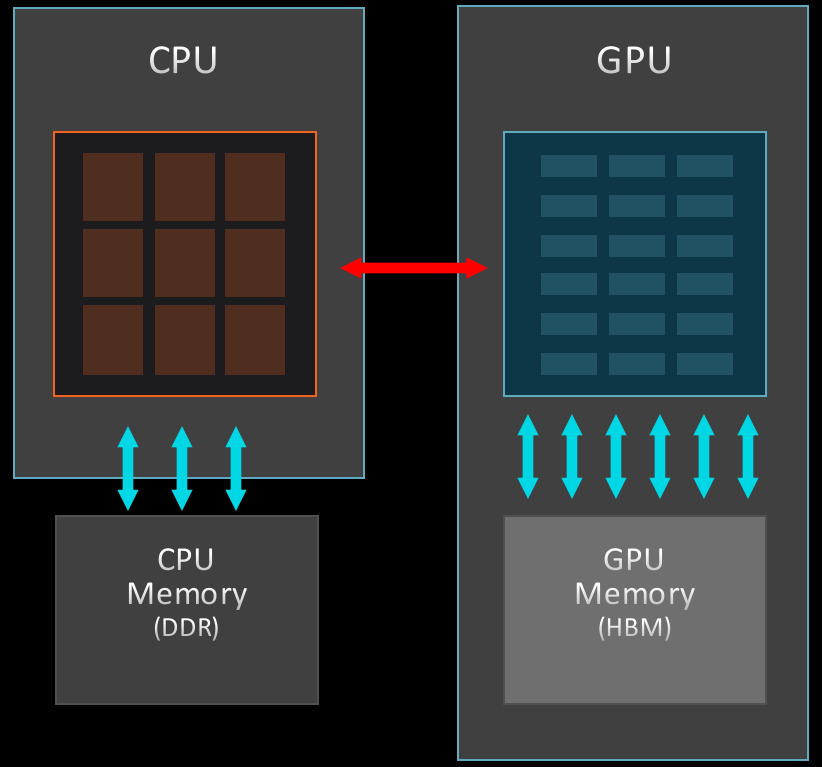


Memory Allocators for computing with GPUs: HMM

https://developer.nvidia.com/blog/simplifying-gpu-application-development-with-heterogeneous-memory-management/

https://www.kernel.org/doc/html/v5.0/vm/hmm.html

Heterogeneous Memory Management (HMM)



How to detect if HMM is supported:

$ nvidia-smi -q | grep Addressing

Addressing Mode : HMM

Memory Allocators for computing with GPUs: HMM

discrete memory model

```
#include <new>

int main() {
  …
  double *X, *Y;
  X = new double[N];
  Y = new double[N];
  for (auto i=0; i<N; i++)
    {X[i] = 1.1; Y[i] = 1.3;}

  double *x_d, *y_d;
  hipMalloc(&x_d,N*sizeoff(double));
  hipMalloc(&y_d,N*sizeoff(double));
  hipMemcpy(x_d,X,…);
  hipMemcpy(y_d,Y,…);
  gpu_func<<< … >>>(x_d, y_d, N);
  hipDeviceSynchronize();
  hipMemcpy(X,x_d,…);
  hipMemcpy(Y,y_d,…);

  cpu_func(X,Y);

  hipFree(x_d); hipFree(y_d);
  delete[] X;
  delete[] Y;
}
compile with : -std=c++17
export HSA_XNACK=1
```

unified memory model

```
#include <new>

int main() {
  …
  double *X, *Y;
  X = new (std::align_val_t(128)) double[N];
  Y = new (std::align_val_t(128)) double[N];
  for (auto i=0; i<N; i++)
    {X[i] = 1.1; Y[i] = 1.3;}

  double *x_d, *y_d;
  hipMalloc(&x_d,N*sizeoff(double));
  hipMalloc(&y_d,N*sizeoff(double));
  hipMemcpy(x_d,X,…);
  hipMemcpy(y_d,Y,…);
  gpu_func<<< … >>>(X, Y, N);(X, Y, N);
  hipDeviceSynchronize();
  hipMemcpy(X,x_d,…);
  hipMemcpy(Y,y_d,…);

  cpu_func(X,Y);

  hipFree(x_d); hipFree(y_d);
  delete[] X;
  delete[] Y;
}
compile with : -std=c++17
export HSA_XNACK=1
```

Concurrent execution: streams

what kind of concurrency we are looking for ?

1. Work done on GPU can overlap with the work done on the CPU
2. Different kernels are executed concurrently on the same GPU
3. Memory copy operations overlap with GPU kernels
4. Memory copy operations overlap with CPU activity
5. Memory copy operations overlap with other memory copy operations
6. Execution on multiple GPUs and multiple CPUs or CPU's cores
7. etc.

https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/

https://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf

https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf

Concurrent execution: streams

cudaError_t cudaStreamCreate (cudaStream_t* pStream)

creates stream which can not overlap with the default stream , but can overlap with other streams

cudaError_t cudaStreamCreateWithFlags (cudaStream_t* pStream, unsigned int flags)
when flag = cudaStreamNonBlocking created stream will be able to overlap with any other stream

Exercises

1. Write a code performing dot product of two vectors, each thread-block will store partial results into HOST memory allocated with cudaHostAlloc . Complete the reduction by CPU cores accessing data from the same cudaHostAlloc'ed memory

2. Write a code performing matrix vector multiplication: y = A*x Use multiple streams such that
   rows 0 → N1-1 will be multiplied by a vector on stream 1; copy y[0:N1-1] to host
   rows N1 → N2-1 will be multiplied by a vector on stream 2; copy y[N1:N2-1] to host
   ….
   partial results of the matrix-vector multiplication will be copied to CPU asynchronously

3. Profile your application using
   nsys profile ./a.out
   rocprof --hip-trace ./a.out

   nsys should generate a file with extension *nsys-rep . Copy this file to your laptop and open it with Nsight-systems (you can double-click on the file so it will be opened automatically)