- Bislang wurden überwiegend alle CUDA-Aktivitäten sequentiell durchgeführt, abgesehen davon, dass die Kernel-Funktionen parallelisiert abgearbeitet werden und der Aufruf eines Kernels asynchron erfolgt.
- In vielen Fällen bleibt so Parallelisierungspotential ungenutzt.
- CUDA-Streams sind eine Abstraktion, mit deren Hilfe mehrere sequentielle Abläufe definiert werden können, die voneinander unabhängig sind und daher prinzipiell parallelisiert werden können.
- Ferner gibt es Synchronisierungsoperationen und das Behandeln von Ereignissen mit CUDA-Streams.

Folgende Aktivitäten können mit Hilfe von CUDA-Streams unabhängig voneinander parallel laufen:

- CPU und GPU können unabhängig voneinander operieren
- der Transfer von Daten und die Ausführung von Kernel-Funktionen.
- Mehrere Kernel können auf der gleichen GPU konkurrierend ausgeführt werden (bei Livingstone können 32 Kernel parallel laufen).
- Wenn mehrere GPUs zur Verfügung stehen, können diese ebenfalls parallel laufen.

Insbesondere bietet es sich an, den Datentransfer und die Ausführung der Kernel-Funktionen zu parallelisieren. Dabei können insbesondere Datentransfers vom Hauptspeicher zur GPU und in umgekehrter Richtung von der GPU zum Hauptspeicher ungestört parallel laufen.

Sobald ein CUDA-Stream erzeugt worden ist, können einzelne Operationen oder der Aufruf einer Kernel-Funktion einem Stream zugeordnet werden:

cudaError_t cudaStreamCreate (cudaStream_t stream)*
: Erzeugt einen neuen Stream. Bei *cudaStream_t* handelt es sich um einen Zeiger auf eine nicht-öffentliche Datenstruktur, die beliebig kopiert werden kann.

cudaError_t cudaStreamSynchronize (cudaStream_t stream)
: Wartet bis alle Aktivitäten des Streams beendet sind.

cudaError_t cudaStreamDestroy (cudaStream_t stream)
: Wartet auf die Beendigung der mit dem Stream verbundenen Aktivitäten und anschließende Freigabe der zum Stream gehörenden Ressourcen.

Für die Datentransfers stehen asynchrone Operationen zur Verfügung, die einen Stream als Parameter erwarten:

cudaError_t cudaMemcpyAsync(**void**∗ *dst*, **const void**∗ *src*, *size_t count*, **enum** *cudaMemcpyKind kind*, *cudaStream_t stream*)

Funktioniert analog zu *cudaMemcpy*, synchronisiert jedoch nicht und reiht den Datentransfer in die zu dem Stream gehörende Sequenz ein.

Beim Aufruf eines Kernels kann bei dem letzten Parameter der Konfiguration ein Stream angegeben werden:

- $<<<$ *Dg*, *Db*, *Ns*, *S* $>>>$
- Der letzte Parameter *S* ist der Stream.
- Bei *Ns* kann im Normalfall einfach 0 angegeben werden.

- Grundsätzlich kann auch ein 0-Zeiger (bzw. **nullptr**) als Stream übergeben werden.
- In diesem Fall werden ähnlich wie bei *cudaDeviceSynchronize* erst alle noch nicht abgeschlossenen CUDA-Operationen abgewartet, bevor die Operation beginnt.
- Das erfolgt aber asynchron, so dass die CPU dessen ungeachtet weiter fortfahren kann.
- Datentransfers und der Aufruf von Kernel-Funktionen ohne die Angabe eines Streams implizieren immer die Verwendung des NULL-Streams. Entsprechend wird in diesen Fällen implizit synchronisiert.
- Wenn versucht wird, mit Streams zu parallelisieren, ist darauf zu achten, dass nicht versehentlich durch die implizite Verwendung eines NULL-Streams eine Synchronisierung erzwungen wird.

Wenn mehrere voneinander unabhängige Kernel-Funktionen hintereinander aufzurufen sind, die jeweils Daten von der CPU benötigen und Daten zurückliefern, lohnt sich u.U. ein Pipelining mit Hilfe von Streams:

- Für jeden Aufruf einer Kernel-Funktion wird ein Stream angelegt.
- Jedem Stream werden drei Operationen zugeordnet:
  - Datentransfer zur GPU
  - Aufruf der Kernel-Funktion
  - Datentransfer zum Hauptspeicher

Wenn der Zeitaufwand für die Datentransfers geringer ist als für die eigentliche Berechnung auf der GPU fällt dieser dank der Parallelisierung weg bei der Berücksichtigung der Gesamtzeit, abgesehen von dem ersten und letzten Datentransfer.

hpc/cuda/copy.hpp

```
template<
   template<typename> class MatrixA,
   template<typename> class MatrixB,
   typename T,
   Require<HostGe<MatrixA<T>>, DeviceGe<MatrixB<T>>> = true
>
void copy(const MatrixA<T>& A, MatrixB<T>& B, Stream& stream) {
   assert(A.numRows() == B.numRows() && A.numCols() == B.numCols() &&
      A.incRow() == B.incRow() && A.incCol() == B.incCol() &&
      consecutively_stored(A) && consecutively_stored(B));
   CHECK_CUDA(cudaMemcpyAsync, B.data(), A.data(),
      A.numRows() * A.numCols() * sizeof(T), cudaMemcpyHostToDevice,
      stream);
}
```

hpc/cuda/copy.hpp

```
template<
   template<typename> class MatrixA,
   template<typename> class MatrixB,
   typename T,
   Require<DeviceGe<MatrixA<T>>, HostGe<MatrixB<T>>> = true
>
void copy(const MatrixA<T>& A, MatrixB<T>& B, Stream& stream) {
   assert(A.numRows() == B.numRows() && A.numCols() == B.numCols() &&
      A.incRow() == B.incRow() && A.incCol() == B.incCol() &&
      consecutively_stored(A) && consecutively_stored(B));
   CHECK_CUDA(cudaMemcpyAsync, B.data(), A.data(),
      A.numRows() * A.numCols() * sizeof(T), cudaMemcpyDeviceToHost,
      stream);
}
```

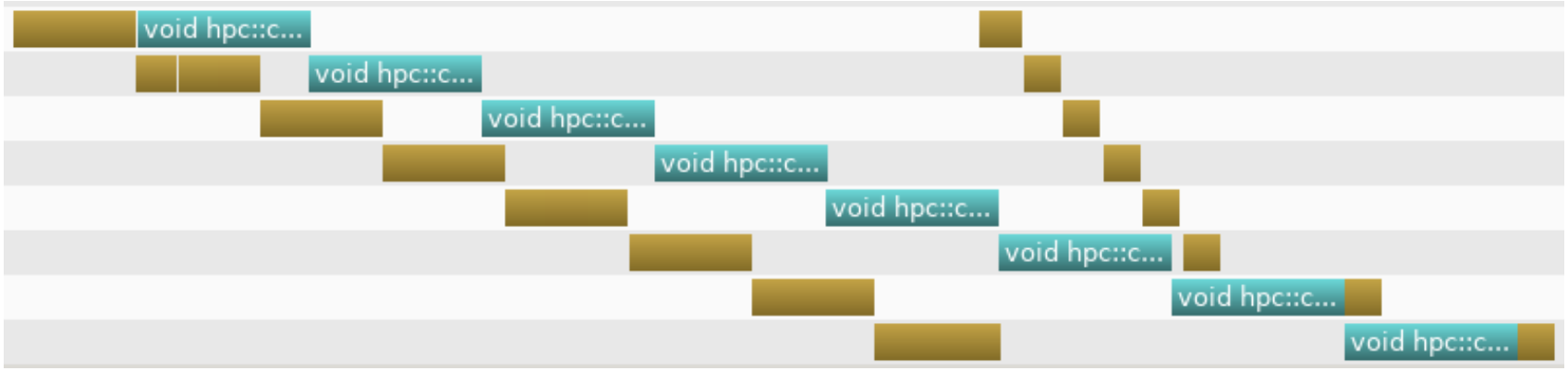
gemm-streamed.cu

```
Stream stream[COUNT];
for (std::size_t index = 0; index < COUNT; ++index) {
   copy(*A_host[index], *A_dev[index], stream[index]);
   copy(*B_host[index], *B_dev[index], stream[index]);
   copy(*C_host[index], *C_dev[index], stream[index]);
   cuda_gemm(alpha, *A_dev[index], *B_dev[index], beta, *C_dev[index],
      stream[index]);
}
for (std::size_t index = 0; index < COUNT; ++index) {
   copy(*C_dev[index], *C_host[index], stream[index]);
}
CHECK_CUDA(cudaDeviceSynchronize); // wait for the kernel to finish
```

- *COUNT* Matrix-Matrix-Multiplikationen werden hier parallel abgewickelt.
- Das entspricht hier dem Fork-And-Join-Pattern, wobei *cudaDeviceSynchronize* dem *join* entspricht.

Matrix-Multiplikationen mit Pipelining


```
livingstone$ nvprof --print-gpu-summary gemm-streamed 2>&1 | cut -b1-104
==13911== NVPROF is profiling process 13911, command: gemm-streamed
8 gemm operations on gpu took 0.103952 s for (1024 x 1024) (1024 x 1024)
time per gemm operation: 0.012994 s
==13911== Profiling application: gemm-streamed
==13911== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   51.45%  93.201ms         8  11.650ms  11.095ms  11.785ms  void hpc::cuda::gemm_kernel
                   37.05%  67.123ms        24  2.7968ms  2.7804ms  3.0706ms  [CUDA memcpy HtoD]
                   11.50%  20.824ms         8  2.6030ms  2.5525ms  2.9126ms  [CUDA memcpy DtoH]
livingstone$ nvprof --print-gpu-summary gemm-serialized 2>&1 | cut -b1-104
==13926== NVPROF is profiling process 13926, command: gemm-serialized
8 gemm operations on gpu took 0.179611 s for (1024 x 1024) (1024 x 1024)
time per gemm operation: 0.0224514 s
==13926== Profiling application: gemm-serialized
==13926== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   50.24%  88.114ms         8  11.014ms  11.003ms  11.022ms  void hpc::cuda::gemm_kernel
                   38.08%  66.788ms        24  2.7828ms  2.7798ms  2.8056ms  [CUDA memcpy HtoD]
                   11.68%  20.476ms         8  2.5595ms  2.5569ms  2.5715ms  [CUDA memcpy DtoH]
livingstone$ nvprof --print-gpu-summary gemm-without-transfers 2>&1 | cut -b1-104
==13940== NVPROF is profiling process 13940, command: gemm-without-transfers
8 gemm operations on gpu took 0.0881717 s for (1024 x 1024) (1024 x 1024)
time per gemm operation: 0.0110215 s
==13940== Profiling application: gemm-without-transfers
==13940== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   50.25%  88.114ms         8  11.014ms  11.005ms  11.022ms  void hpc::cuda::gemm_kernel
                   38.09%  66.784ms        24  2.7827ms  2.7798ms  2.8080ms  [CUDA memcpy HtoD]
                   11.66%  20.455ms         8  2.5568ms  2.5557ms  2.5578ms  [CUDA memcpy DtoH]
livingstone$
```



Das Werkzeug *nvvp* erlaubt es, das Scheduling der einzelnen Streams zu visualisieren:

- Jede Zeile entspricht einem Stream.
- Die x-Achse entspricht der Zeitachse.
- Jeder der Streams führt zunächst die erste Kopieraktion aus (*host to device*, in Ocker dargestellt), dann den Kernel (in türkiser Farbe) und schließlich die zweite Kopieraktion (*device to host*, wieder in Ocker).
- In der überwiegenden Zeit läuft immer eine der Kopieraktionen parallel zu einem der Kernel-Aufrufe.

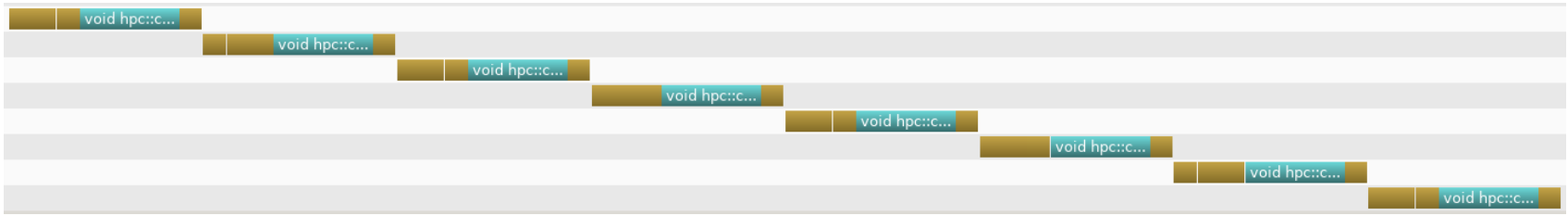
Alternative Variante

gemm-badly-streamed.cu

```
Stream stream[COUNT];
for (std::size_t index = 0; index < COUNT; ++index) {
   copy(*A_host[index], *A_dev[index], stream[index]);
   copy(*B_host[index], *B_dev[index], stream[index]);
   copy(*C_host[index], *C_dev[index], stream[index]);
   cuda_gemm(alpha, *A_dev[index], *B_dev[index], beta, *C_dev[index],
      stream[index]);
   copy(*C_dev[index], *C_host[index], stream[index]);
}
```

- Diese Variante liefert das gleiche Resultat und kommt nur mit einer *for*-Schleife aus.
- Ist sie aber auch genauso gut?

```
livingstone$ nvprof --print-gpu-summary gemm-badly-streamed 2>&1 | cut -b1-104
==16244== NVPROF is profiling process 16244, command: gemm-badly-streamed
8 gemm operations on gpu took 0.1816 s for (1024 x 1024) (1024 x 1024)
time per gemm operation: 0.0227 s
==16244== Profiling application: gemm-badly-streamed
==16244== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   51.27%  91.855ms         8  11.482ms  11.055ms  11.752ms  void hpc::cuda::gemm_kernel
                   37.31%  66.853ms        24  2.7855ms  2.7791ms  2.8216ms  [CUDA memcpy HtoD]
                   11.42%  20.463ms         8  2.5578ms  2.5502ms  2.5712ms  [CUDA memcpy DtoH]
livingstone$ nvprof --print-gpu-summary gemm-streamed 2>&1 | cut -b1-104
==16259== NVPROF is profiling process 16259, command: gemm-streamed
8 gemm operations on gpu took 0.0994354 s for (1024 x 1024) (1024 x 1024)
time per gemm operation: 0.0124294 s
==16259== Profiling application: gemm-streamed
==16259== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   50.24%  88.701ms         8  11.088ms  11.056ms  11.110ms  void hpc::cuda::gemm_kernel
                   37.98%  67.066ms        24  2.7944ms  2.7788ms  3.0685ms  [CUDA memcpy HtoD]
                   11.78%  20.802ms         8  2.6002ms  2.5541ms  2.9147ms  [CUDA memcpy DtoH]
livingstone$
```



- Zu einer Parallelisierung kommt es hier überhaupt nicht.
- Das liegt daran, dass wir hier nur über eine einzige *copy engine* verfügen. Die Queue wird somit strikt in der Reihenfolge abgearbeitet, wie die Jobs eingegangen sind. Entsprechend wartet der erste *device to host* Kopierjob erst auf die Fertigstellung des ersten Kernel-Aufrufs. Erst danach kommen die *host to device* Kopierjobs für die nächste Matrix.
- Es werden in der Queue leider keine Jobs nach vorne gezogen, wenn deren Abhängigkeiten bereits alle erfüllt sind.

- Der GPU steht über die PCIe-Schnittstelle *direct memory access* (DMA) zur Verfügung.
- Dies wäre recht schnell, kommt aber normalerweise nicht zum Zuge, da dazu sichergestellt sein muss, dass die entsprechenden Kacheln im Hauptspeicher nicht zwischenzeitlich vom Betriebssystem ausgelagert werden.
- Alternativ ist es möglich, ausgewählte Bereiche des Hauptspeichers zu reservieren, so dass diese nicht ausgelagert werden können (*pinned memory*).
- Davon sollte zurückhaltend Gebrauch gemacht werden, da dies ein System in die Knie zwingen kann, wenn zuviele physische Kacheln reserviert sind.

Nicht auslagerbarer Speicher (*pinned memory*) muss mit speziellen Funktionen belegt und freigegeben werden:

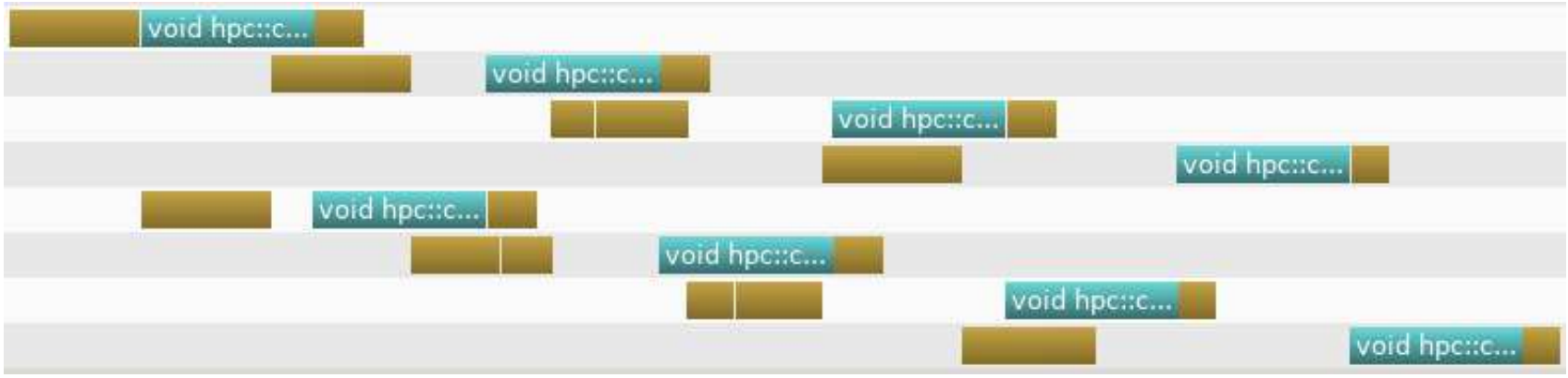
cudaError_t cudaMallocHost(**void**∗∗ *ptr*, *size_t size*)
: belegt ähnlich wie *malloc* Hauptspeicher, wobei hier sichergestellt wird, dass dieser nicht ausgelagert wird.

cudaError_t cudaFreeHost(**void**∗)
: gibt den mit *cudaMallocHost* oder *cudaHostAlloc* reservierten Speicher wieder frei.

hpc/cuda/pinned-buffer.hpp

```
template<typename T>
struct PinnedBuffer {
   void* const hostptr;
   T* const aligned_hostptr;
   PinnedBuffer(std::size_t length, std::size_t alignment = alignof(T)) :
         hostptr(cuda_host_malloc(compute_aligned_size<T>(length, alignment)))
         aligned_hostptr(align_ptr<T>(hostptr, alignment)) {
   }
   ~PinnedBuffer() {
      CHECK_CUDA(cudaFreeHost, hostptr);
   }
   T* data() const {
      return aligned_hostptr;
   }
   /* ... */
};
```

- Als entsprechende RAII-Klasse steht *PinnedBuffer* zur Verfügung.
- Darauf aufbauend gibt es *PinnedDenseVector* und *PinnedGeMatrix* mitsamt den zugehörigen Views, die sich ansonsten genauso wie *DenseVector* und *GeMatrix* verhalten.



- Durch die Verwendung von *PinnedGeMatrix* an Stelle von *GeMatrix* sieht sich die GPU in der Lage, Kopieraktionen zu parallelisieren, wenn sich die Transferrichtungen voneinander unterscheiden.
- Bei diesem Beispiel gibt es jedoch keinen Zeitgewinn, da immer nur eine Kernel-Funktion ausgeführt wird und am Ende noch das Resultat des letzten Kernel-Funktionsaufrufs zu übertragen ist.

Neuere Grafikkarten und Versionen der CUDA-Schnittstelle (einschließlich Livingstone) erlauben die Abbildung nicht auslagerbaren Hauptspeichers in den Adressraum der GPU:

cudaError_t cudaHostAlloc(**void**∗∗ *ptr*, *size_t size*, **unsigned int** *flags*)
belegt Hauptspeicher, der u.a. in den virtuellen Adressraum der GPU abgebildet werden kann. Folgende miteinander kombinierbare Optionen gibt es:

cudaHostAllocDefault emuliert *cudaMallocHost*, d.h. der Speicher wird nicht abgebildet.

cudaHostAllocPortable macht den Speicherbereich allen Grafikkarten zugänglich (falls mehrere zur Verfügung stehen).

cudaHostAllocMapped bildet den Hauptspeicher in den virtuellen Adressraum der GPU ab

cudaHostAllocWriteCombined ermöglicht u.U. eine effizientere Lesezugriffe der GPU zu Lasten der Lesegeschwindigkeit auf der CPU.

Neuere CUDA-Versionen unterstützen *unified virtual memory* (UVM), bei dem die Zeiger auf der CPU- und GPU-Seite für abgebildeten Hauptspeicher identisch sind. (Dies gilt auch dann, wenn wenn die CPU mit 64-Bit- und die GPU mit 32-Bit-Zeigern arbeitet.)
Ohne UVM müssen die Zeiger abgebildet werden:

cudaError_t cudaHostGetDevicePointer(**void**∗∗ *pDevice*, **void**∗ *pHost*, **unsigned int** *flags*)

liefert für Hauptspeicher, der in den Adressraum der GPU abgebildet ist (*cudaDeviceMapHost* wurde angegeben) den entsprechenden Zeiger in den Adressraum der GPU. Bei *unified virtual memory* (UVM) sind beide Zeiger identisch. (Bei den *flags* ist nach dem aktuellen Stand der API immer 0 anzugeben.)

Ob UVM unterstützt wird oder nicht, lässt sich über das Feld *unifiedAddressing* aus der **struct** *cudaDeviceProp* ermitteln, die mit *cudaGetDeviceProperties* gefüllt werden kann.

- Bei integrierten Systemen wird jeglicher Kopieraufwand vermieden (siehe das Feld *integrated* in der **struct** *cudaDeviceProp*).
- Bei nicht-integrierten Systemen werden die Daten jeweils implizit per *direct memory access* transferiert.
- Wenn der Speicher auf der GPU sonst nicht ausreicht.
- Wenn jede Speicherzelle nicht mehr als einmal in konsekutiver Weise gelesen oder geschrieben wird (ansonsten sind Zugriffe durch einen Cache effizienter).
- Reine Schreibzugriffe sind günstiger, da hier die Synchronisierung wegfällt, d.h. die Umsetzung einer Schreib-Operation und die Fortsetzung der Kernel-Funktion erfolgen parallel.
- Bei Lesezugriffen ist der Vorteil geringer, da hier gewartet werden muss.

Datenstruktur für abgebildeten Hauptspeicher

hpc/cuda/mapped-buffer.hpp

```
template<typename T>
struct MappedBuffer {
   void* const hostptr;
   T* const aligned_hostptr;
   void* const devptr;
   T* const aligned_devptr;

   MappedBuffer(std::size_t length, std::size_t alignment = alignof(T)) :
         hostptr(cuda_host_malloc(compute_aligned_size<T>(length, alignment))),
         aligned_hostptr(align_ptr<T>(hostptr, alignment)),
         devptr(convert_host_to_device_ptr(hostptr)),
         aligned_devptr(align_devptr(hostptr, aligned_hostptr, devptr)) {
   }

   ~MappedBuffer() {
      CHECK_CUDA(cudaFreeHost, hostptr);
   }

   HOST_DEV
   T* host_data() const {
      return aligned_hostptr;
   }

   HOST_DEV
   T* dev_data() const {
      return aligned_devptr;
   }

   /* ... */
};
```

hpc/cuda/mapped-buffer.hpp

```
inline void* cuda_host_malloc(std::size_t size) {
   void* hostptr = nullptr;
   CHECK_CUDA(cudaHostAlloc, &hostptr, size, cudaHostAllocMapped);
   return hostptr;
}

inline void* convert_host_to_device_ptr(void* hostptr) {
   void* devptr = nullptr;
   CHECK_CUDA(cudaHostGetDevicePointer, &devptr, hostptr, 0);
   return devptr;
}

template<typename T>
inline T* align_devptr(void* hostptr, T* aligned_hostptr, void* const devptr) {
   auto offset = aligned_hostptr - static_cast<T*>(hostptr);
   return static_cast<T*>(devptr) + offset;
}
```

hpc/cuda/buffer.hpp

```
template<typename T>
constexpr std::size_t
compute_aligned_size(std::size_t length, std::size_t wanted_alignment) {
   return length * sizeof(T) + std::max(wanted_alignment, alignof(T));
}
```