Технология CUDA для высокопроизводительных вычислений на кластерах с GPU

Лихогруд Николай
n.lihogrud@gmail.com

Часть шестая

Pinned-память

время выполнения задачи =
время работы ядра + обмен данными между CPU и GPU

- Как сократить накладные расходы на обмен данными между CPU и GPU?
  - Ускорение копирований
  - Выполнение копирований параллельно с вычислениями

DMA & zero-copy

- “Zero-copy”- копирование памяти без участия центрального процессора
  - Копирование выполняется спец. контроллером, процессор переключается на другие задачи
- DMA (Direct memory access) – прямой доступ к оперативной памяти, без участия ЦП
  - Реализуется через zero-copy операции
  - Скорость передачи увеличивается, так как данные не пересылаются в ЦП и обратно

DMA и виртуальная память

- Виртуальная память организованна в страницы, отображаемые на физические страницы ОП

- Виртуальные страницы могут быть
  - Перемещены в оперативной памяти
  - Отгружены на диск (*swapping*)

В таких условиях реализовать DMA очень сложно!

DMA и виртуальная память

- Запретим перемещение страниц по ОП и их выгрузку на диск
  - Привяжем страницы виртуальной памяти к страницам физической
  - Эти физические страницы теперь недоступны ОС для размещения новых виртуальных страниц (paging)

page-able → page-locked

Для page-locked памяти DMA реализуемо без существенных накладных расходов

Page-locked память & CUDA

- «Pinned» - синоним, «прикрепленный»
- В CUDA можно напрямую выделить page-locked (pinned) память на хосте или сделать таковой память, выделенную ранее
- Операции копирования Host ←→ Device с ней происходят быстрее и могут выполняться параллельно с работой ядер

cudaHostRegister

- “Залочить” память, предварительно выделенную обычным способом:

```
float *ptr = malloc(n * sizeof(float))
cudaHostRegister(ptr, n * sizeof(float),0);
cudaMemcpy(devPtr,prt, n * sizeof(float),
                        cudaMemcpyHostToDevice)
…
cudaHostUnregister(ptr);
```

Mapped pinned-память

- Pinned-память можно отобразить в виртуальное адресное пространство GPU
- Нити смогут обращаться к ней напрямую, без необходимости копирования в память GPU
  - Необходимые копирования будут неявно выполняться асинхронно, параллельно с работой ядра
- С хоста память будет так же доступна

Mapped pinned-память

- Залочить память на хосте и получить указатель, по которому к ней можно обращаться из ядер:

```
cudaHostRegister(ptr,n * sizeof(float),
                 cudaHostRegisterMapped);
float *ptrForDevice = NULL;
cudaHostGetDevicePointer(&ptrForDevice, ptr, 0);
// не нужно выделять память на GPU и копировать в
// неё входные данные
kernel<<<…>>>(ptrForDevice,…);
```

Mapped pinned-память

- Для активации возможности маппирования pinned-памяти:
  - До первого вызова функции из cuda-runtime (т.е. до инициализации устройства) установить флаг инициализации `cudaDeviceMapHost`:

```
cudaSetDeviceFlags(cudaDeviceMapHost);
cudaSetDevice(0);// инициализируется с флагами
```

  - Проверить свойство устройства `canMapHostMemory`:

```
cudaDeviceProp deviceProp;
cudaGetDeviceProperties(0, &deviceProp);
if (deviceProp.canMapHostMemory) {
    …
}
```

Прямое выделение pinned-памяти

- Самое простое:

```
float *ptr = NULL;
cudaMallocHost(&ptr, n * sizeof(float));
```

- С флагами:

```
cudaHostAlloc(&ptr, n * sizeof(float),
                    cudaHostAllocDefault);
```

Возможные флаги:

- `cudaHostAllocDefault`: эмулирование `cudaMallocHost`().
- `cudaHostAllocMapped`: аналогично `cudaHostRegisterMapped`

cudaMemcpy* и pinned-память

- Копировании обычной (pageable) памяти с хоста на GPU происходит через DMA к промежуточному буферу в pinned-памяти
- Управление хосту возвращается после выполнения копирований в этот буфер, но необязательно до завершения DMA

cudaMemcpy* и pinned-память

- Копировании обычной (pageable) памяти с хоста на GPU происходит через DMA к промежуточному буферу в pinned-памяти
  - Поэтому копирование сразу из pinned-памяти быстрее – не нужно выделять память под буфер и копировать в него данные

Тест

```
float *hostPtr = (float *)malloc(numberOfBytes);
cudaEventRecord(startPageable, 0);
cudaMemcpy(devicePtr, hostPtr, numberOfBytes,
                              cudaMemcpyHostToDevice);
cudaEventRecord(stopPageable, 0);

cudaHostRegister(hostPtr, numberOfBytes, 0);

cudaEventRecord(startPinned, 0);
cudaMemcpy(devicePtr, hostPtr, numberOfBytes,
                              cudaMemcpyHostToDevice);
cudaEventRecord(stopPinned, 0);
cudaDeviceSynchronize();
```

Тест

```
float elapsedPinned, elapsedPageable;
cudaEventElapsedTime(&elapsedPageable, startPageable,
                                stopPageable);
cudaEventElapsedTime(&elapsedPinned, startPinned,
                                stopPinned);
printf("Copy from pageable %f\n", elapsedPageable);
printf("Copy from pinned %f\n", elapsedPinned);
```

```
$./a.out
Copy from pageable 555.893066
Copy from pinned 339.375580
```

Замечания

- Выделение pinned-памяти занимает больше времени, чем обычный `malloc`
- Доступный для выделения объем сильно ограничен
- Чрезмерное использование page-locked памяти деградирует систему
  - Для освобождения использовать `cudaFreeHost()`, `cudaHostUnregister()`

UVA

Unified Virtual Address (UVA)

- На 64-битной архитектуре, начиная с поколения Fermi (cc 2.0), используется единое виртуальное адресное пространство для памяти хоста и всех устройств
  - Unified Virtual Address space, UVA

- Если UVA включено, то

```
cudaDeviceProp::unifiedAddressing == 1
```

Unified Virtual Address (UVA)

- Без UVA для каждого указателя хранятся метаданные о том где реально расположена память, на которую он указывает
- С UVA эта информация «вшита» в значение указателя
  - Диапазоны адресов всех GPU и CPU не пересекаются

Unified Virtual Address (UVA)

- Чтобы узнать где реально расположена память:

```
float *ptr;
cudaPointerAttributes pointerAttributes;
cudaPointerGetAttribute
                        (&pointerAttributes, ptr)
```

Unified Virtual Address (UVA)

```
struct cudaPointerAttributes {
    enum cudaMemoryType memoryType;
    int device;
    void *devicePointer;
    void *hostPointer;
}
```

- **memoryType** - cudaMemoryTypeHost | cudaMemoryTypeDevice
- **device** - устройство, на котором расположена память
- **devicePointer** - NULL, если не доступна с текущего устройства
- **hostPointer** – NULL, если не доступна с хоста

Pinned-память и UVA

- С UVA память, выделенная через `cudaHostAlloc()`
  - Автоматически является **mapped**
  - Доступна с хоста и с любого GPU по одному и тому же указателю (т.к. адресное пространство единое)
    - Не нужно использовать `cudaHostGetDevicePointer()`
    - Исключение – `cudaHostAllocWriteCombined`

Pinned-память и UVA

- Для памяти, залоченной через `cudaHostRegister` и для **write-combined** памяти указатели для хоста и для устройства являются разными
  - Нужен `cudaHostGetDevicePointer()`

Пример

- Без UVA и mapped памяти:

```
float *ptr = NULL;
cudaHostAlloc(&ptr, 1024, 0);
float *ptrForDevice = NULL;
cudaMalloc(&ptrForDevice, 1024);
cudaMemcpy(ptrForDevice, ptr, 1024,
                    cudaMemcpyHostToDevice)
kernel<<<…>>>(ptrForDevice,…);
```

Пример

- С mapped памятью:

```
cudaSetDeviceFlags(cudaDeviceMapHost);
cudaDeviceProp deviceProp;
cudaGetDeviceProperties(device, &deviceProp);
if (deviceProp.canMapHostMemory ) {
   float *ptr = NULL;
   cudaHostAlloc(&ptr, 1024, cudaHostAllocMapped)
   float *ptrForDevice = NULL;
   cudaHostGetDevicePointer(&ptrForDevice, ptr, 0);
   kernel<<<…>>>(ptrForDevice,…);
}
```

Пример

- С mapped памятью и UVA:

```
cudaSetDeviceFlags(cudaDeviceMapHost)
cudaDeviceProp deviceProp;
cudaGetDeviceProperties(device, &deviceProp);
if (deviceProp.unifiedAddressing ) {
   float *ptr = NULL;
   cudaHostAlloc(&ptr, 1024, cudaHostAllocMapped)
   kernel<<<…>>>(ptr,…);
}
```

Пример

```
float *ptrForDevice = NULL;
if (deviceProp.unifiedAddressing ) {
   ptrForDevice = ptr
} else if (deviceProp.canMapHostMemory ) {
   cudaHostGetDevicePointer(&ptrForDevice, ptr, 0);
} else {
   cudaMalloc(&ptrForDevice, 1024);
   cudaMemcpy(ptrForDevice, ptr, 1024,
                  cudaMemcpyHostToDevice)
}
kernel<<<…>>>(ptrForDevice,…);
```

cudaMemcpy* и UVA

- С UVA система в состоянии сама определить где находится память
  - Можно указывать `cudaMemcpyDefault` в `cudaMemcpyKind`:

```
float *dstPtr, *srcPtr;
cudaMemcpy(dstPtr, srcPtr,
           n*sizeof(float), cudaMemcpyDefault)
```

Выводы

Выводы

время выполнения задачи =

время работы ядра + обмен данными между CPU и GPU

- page-locked (pinned) память позволяет
  1. Уменьшить время обмена данными
  2. Упростить хост-код при использовании mapped pinned памяти и доступе к ней напрямую из ядер
     - Не нужно возиться с пересылкой данных на GPU и обратно
     - С UVA обращаемся к памяти с хоста и с устройства по одному указателю

CUDA-потоки (cudaStream)

cudaStream

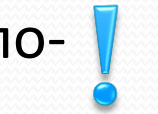
- Последовательность команд для GPU (запуски ядер, копирования памяти и т.д.), исполняемая строго последовательно
  - следующая команда выполняется после полного завершения предыдущей

cudaStream

- Пользователь сам создает потоки и распределяет команды по ним
- По умолчанию, все команды помещаются в «Default Stream», равный нулю

cudaStream

- Только команды из разных потоков, отличных от потока по-умолчанию, могут выполняться параллельно



- Пользователь сам задет необходимую синхронизацию между командами из разных потоков (при наличии зависимостей)
  - В общем случае, порядок выполнения команд из разных потоков неопределен

Создание и уничтожение

```
cudaStream_t stream;
cudaStreamCreate(&stream);
…
cudaStreamDestroy(stream);
```

- Поток привязывается к текущему активному устройству
  - Перед отправлением команды нужно переключаться на устройство, к которому привязан поток
  - Если попробовать отправить в него команду при другом активном устройстве, будет ошибка

Создание и уничтожение

```
cudaStream_t stream;
cudaStreamCreate(&stream);
…
cudaStreamDestroy(stream);
```

- `cudaStreamDestroy` не выполняет синхронизацию
  - Управление возвращается хостовому процессу сразу, реальное освобождение ресурсов произойдет после завершения всех команд потока

Асинхронное копирование

```
cudaMemcpyAsync ( void* dst, const void* src,
          size_t count,
          cudaMemcpyKind kind,
          cudaStream_t stream = 0)

cudaMemcpy2DAsync ( void* dst, size_t dpitch,
                const void* src, size_t spitch,
                size_t width, size_t height,
                cudaMemcpyKind kind,
                cudaStream_t stream = 0 )
```

Асинхронное копирование

Когда возвращается управление хостовой нити

	Host->device		Device->host		host-host	dev-dev
	pageable	pinned	pageable	pinned		
memcpy	После копирования в буфер*	После полного завершения	После полного завершения	После полного завершения	После полного завершения	сразу
memcpyAsync	После копирования в буфер	сразу	После полного завершения	сразу	После полного завершения	сразу

*В начале работы неявно вызывается cudaDeviceSynchronize

Асинхронное копирование

Когда возвращается управление хостовой нити

	Host->device		Device->host		host-host	dev-dev
	pageable	**pinned**	pageable	**pinned**		
memcpy	После копирования в буфер*	После полного завершения	После полного завершения	После полного завершения	После полного завершения	сразу
memcpyAsync	После копирования в буфер	**сразу**	После полного завершения	**сразу**	После полного завершения	сразу

*В начале работы неявно вызывается cudaDeviceSynchronize

Параллельное выполнение команд

- Команды из разных потоков, отличных от потока по-умолчанию, могут исполняться параллельно
  - В зависимости от аппаратных возможностей
- Возможные случаи:
  - Параллельные копирование и выполнение ядра
  - Параллельные выполнение ядер
  - Параллельные копирования с хоста на устройство и с устройства на хост

Копирование & выполнение ядра

- Если `cudaDeviceProp::asyncEngineCount` > 0 устройство может выполнять параллельно копирование и счет ядра
  - Хостовая память долна быть **page-locked**

```
cudaMallocHost(&aHost, size);
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
cudaMemcpyAsync( aDev, aHost, size,
                 cudaMemcpyHostToDevice, stream1);
kernel<<<grid, block, 0, stream2>>>(…);
```

Параллельное выполнение ядер

- Если `cudaDeviceProp::concurrentKernels` > 0 устройство может выполнять ядра параллельно

```
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
kernel1<<<grid, block, 0, stream1>>>(data_1);
kernel2<<<grid, block, 0, stream2>>>(data_2);
```

Копирование в обе стороны & выполнение ядра

- Если cudaDeviceProp::asyncEngineCount== 2 устройство может выполнять параллельно копирование в обе стороны и счет ядра

```
cudaMallocHost(&aHost, size);
cudaMallocHost(&bHost, size);
// создать потоки
cudaMemcpyAsync( aDev, aHost, size,
         cudaMemcpyHostToDevice, stream1);
cudaMemcpyAsync( bHost, bDev, size,
         cudaMemcpyDeviceToHost, stream2);
kernel<<<grid, block, 0, stream3>>>(…);
```

Средства синхронизации CUDA-потоков

Неявная синхронизация

- Неявная синхронизация (ожидание завершения всех команд на устройтве) выполняется перед:
  - Выделением page-locked памяти / памяти на устройстве
  - `cudaMemSet`
  - Копированием между пересекающимися областями памяти на устройстве
  - Отправкой команды в поток по-умолчанию
  - Переключением режима кеша L1

- Если между отправкой двух команд в разные потоки стоит что-то из этого списка – параллельного выполнения не будет

События (cudaEvent)

- Маркеры, приписываемые «точкам программы»
  - ✓ Можно проверить произошло событие или нет
  - ✓ Можно замерить время между двумя произошедшими событиями
  - ✓ Можно синхронизоваться по событию, т.е. заблокировать CPU-поток до момента его наступления
- «Точки программы» расположены между отправками команд на GPU

Запись события

```
cudaError_t cudaEventRecord (
                            cudaEvent_t event,
                            cudaStream_t stream = 0)
```

- Приписывает событие к точке программы в потоке **stream**, в которой вызывается

```
kernel<<<…, stream>>> (…);

…;//нет запусков команд в потоке stream
cudaEventRecord(event, stream);
…;//нет запусков команд в потоке stream

cudaMemcpyAsync(…, stream);
```

Точка программы в потоке **stream** между вызовом ядра и асинхронным копированием

Совершение события

- Событие происходит, когда выполнение команд на GPU реально доходит до точки, к которой в последний раз было приписано событие

Совершение события

- Событие происходит когда завершаются все команды, помещённые в поток, к которому приписано событие, до последнего вызова cudaEventRecord для него

- Если событие приписано потоку по умолчанию (stream = 0), то оно происходит в момент завершения всех команд, помещённых во все потоки до последнего вызова cudaEventRecord для него

Синхронизация по событию

```
cudaError_t cudaEventQuery(cudaEvent_t event)
```

- Возвращает `cudaSuccess`, если событие уже произошло (вся работа до последнего `cudaEventRecord` выполнена): иначе **`cudaErrorNotReady`**

```
cudaError_t cudaEventSynchronize
                        (cudaEvent_t  event)
```

- Возвращает управление хостовой нити только после наступления события

Синхронизация на GPU

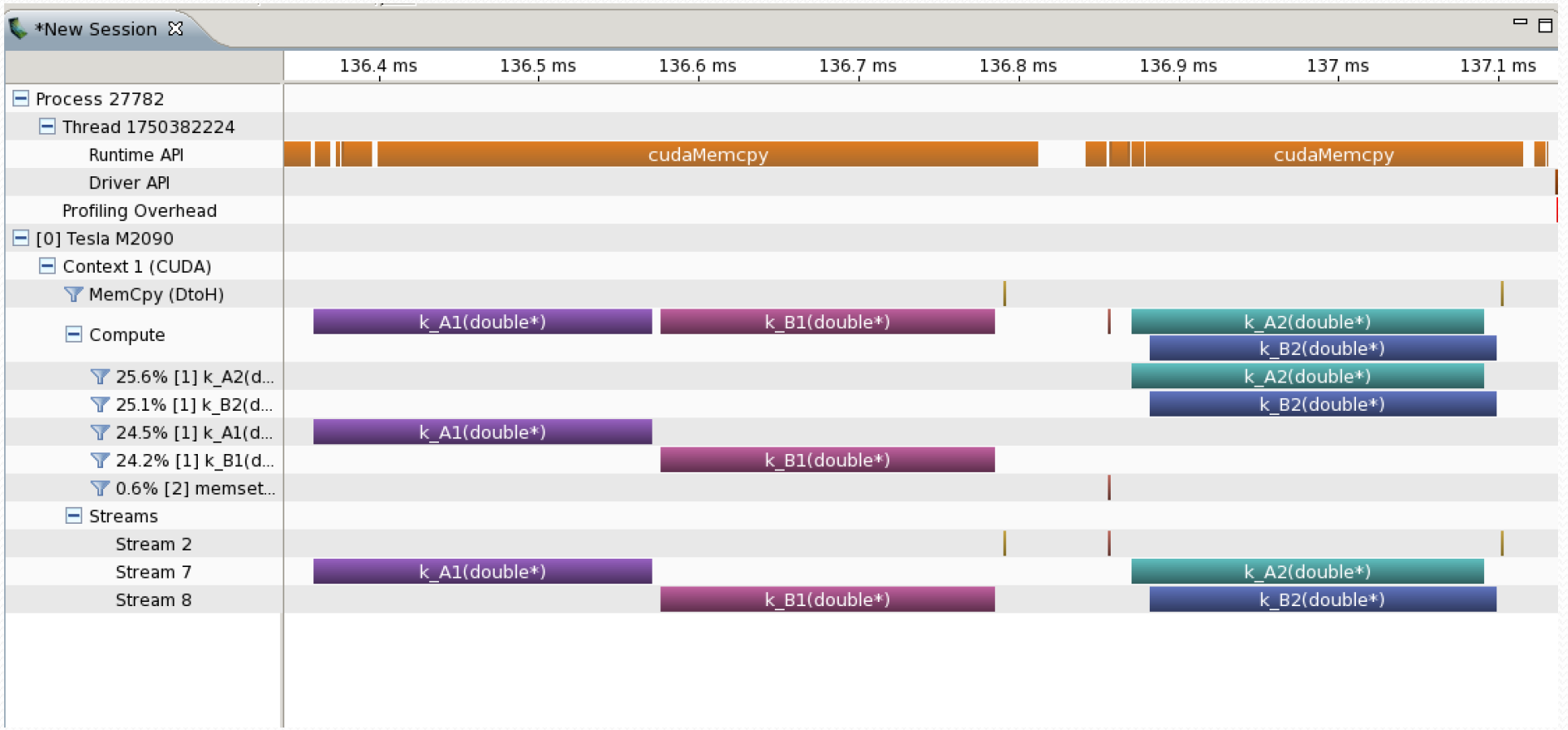
```
cudaError_t cudaStreamWaitEvent
      (cudaStream_t stream, cudaEvent_t event,
            unsigned int  flags )
```

- Команды, отправленные в **stream**, начнут выполняться после наступления события **event**
  - Синхронизация будет эффективно выполнена на GPU
  - При stream == NULL будут отложены все команды всех потоков
- Событие **event** может быть записано на другом GPU
  - Синхронизация между GPU

Синхронизация на GPU

```
A1<<<1, 1 , 0, streamA>>>(d); // A1
cudaEventRecord(halfA, streamA);
cudaStreamWaitEvent(streamB, halfA, 0);
B1<<<1, 1, 0, streamB>>>(d); // B1 начнется после
                                завершения A1
cudaEventRecord(halfB, streamB);
cudaStreamWaitEvent(streamA, halfB, 0);
A2<<<1, 1, 0, streamA>>>(d); // A2 начнется после
                                завершения B1
B2<<<1, 1, 0, streamB>>>(d); // B2
```

Синхронизация на GPU



Синхронизация по потоку

```
cudaError_t cudaStreamQuery (cudaStream_t stream);
```

- Возвращает `cudaSuccess`, если выполнены все команды в потоке stream, иначе **`cudaErrorNotReady`**

```
cudaError_t cudaStreamSynchronize (cudaStream_t stream);
```

- Возвращает управление хостовой нити, когда завершится выполнение всех команд, отправленных в поток stream

cudaStreamCallback

```
typedef void (*cudaStreamCallback_t)(cudaStream_t stream,
            cudaError_t status, void *userData );
```

```
cudaError_t cudaStreamAddCallback (
      cudaStream_t stream, cudaStreamCallback_t callback,
      void *userData, unsigned int flags );
```

- **callback** будет вызван, когда выполнятся все предшествующие команды, отправленные в поток
- В callback запрещены обращения к CUDA API

Выводы

Выводы

В следующий раз, после детального разбора случаев применения потоков

The end