Атомарные операции. Потоки. Параллельное копирование и выполнение ядра. Интероперабельность с OpenGL Библиотека thrust

Лекторы:

Боресков А.В. (ВМиК МГУ)

План

- Атомарные операции
- Потоки, их использование
- Реализация одновременного копирования данных и выполнения ядра
- Интероперабельность с OpenGL
- thrust

Атомарность операции

Рассмотрим традиционную операцию инкремента

```
x++
```

На практике она переводится в следующие операции

```
r = x; load into register
inc r; increment value in register
x = r; store incremented value
```

Атомарность операции

Теперь рассмотрим ситуацию когда две нити пытаются одновременно выполнить операцию инкремента над одной и той же переменной

```
; Thread 1          ; Thread 2
r1 = x;             r2 = x;
inc r1;             inc r2;
x = r1;             x = r2;
```

Тут уже возможны варианты наложения операций друг на друга

Атомарность операции. Конфликт

```
; Thread 1
; x = 0
r1 = x; r1 = 0
inc r1; r1 = 1
x = r1; x = 1
```

```
; Thread 2
; x = 0
r2 = x; r2 = 0
inc r2; r2 = 1
x = r2; x = 1
```

Операция прошла некорректно - значение переменной было инкрементировано всего один раз

Атомарные операции

CUDA поддерживает специальные операции, гарантирующие атомарность

- Они выполняются медленнее
- CC 1.1 поддерживает целочисленные атомарные операции в глобальной памяти
- CC 1.1 поддерживает целочисленные атомарные операции в разделяемрй памяти

Атомарные операции

- Над 64-битовыми целыми с CC 2.0
- atomicAdd для float – CC 2.0

Атомарные операции

```
  // возвращают старое значение
int atomicAnd ( int * addr, int value );
uint atomicAnd ( uint * addr, uint value );
unsigned long long atomicAdd ( unsigned long long * addr, unsigned long long value );
float atomicAdd ( float * addr, float value );

int  atomicSub ( int * address, int value );
uint atomicSub ( uint * address, uint value );

  // записывает значение по адресу, возарщает старое значение
int atomicExch ( int * addr, int value );
uint atomicExch ( uint * addr, uint value );
unsigned long long atomicExch ( unsigned long long * addr, unsigned long long value );

  // записывает результат операции, возвращает старое значение
int  atomicMin ( int  * addr, int value );
uint atomicMin ( uint * addr, uint value );
int  atomicMax ( int  * addr, int value );
uint atomicMax ( uint * addr, uint value );

uint atomicInc ( uint * addr, uint value );
uint atomicDec ( uint * addr, uint value );

int  atomicAnd ( int  * addr, int  value );
uint atomicAnd ( uint * addr, uint value );
int  atomicOr  ( int  * addr, int  value );
uint atomicOr  ( uint * addr, uint value );
int  atomicXor ( int  * addr, int  value );
uint atomicXor ( uint * addr, uint value );
```

Потоки (Streams)

- GPU умеют выполнять многие вещи параллельно
  - Выполнение ядер и копирование памяти между CPU и GPU может выполняться параллельно
  - GPU с CC 2.x умеют выполнять до 16 ядер одновременно

Потоки

- Поток (stream) в CUDA представляет собой очередь запросов, которые должны быть выполнены а заданном порядке
- По умолчанию используется всегда существующий поток 0
- Однако можно создать несколько потоков, тогда операции из разных потоков могут выполняться параллельно

Потоки

```
cudaStream_t stream;

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

Потоки (пример)

Рассмотрим следующую задачу – есть операция, берущая на вход два массива и по каждой паре элементов из соответствующих массивов строящая элемент третьего (выходного) массива

c[i] = foo(a[i], b[i])

Потоки (пример)

Традиционный способ

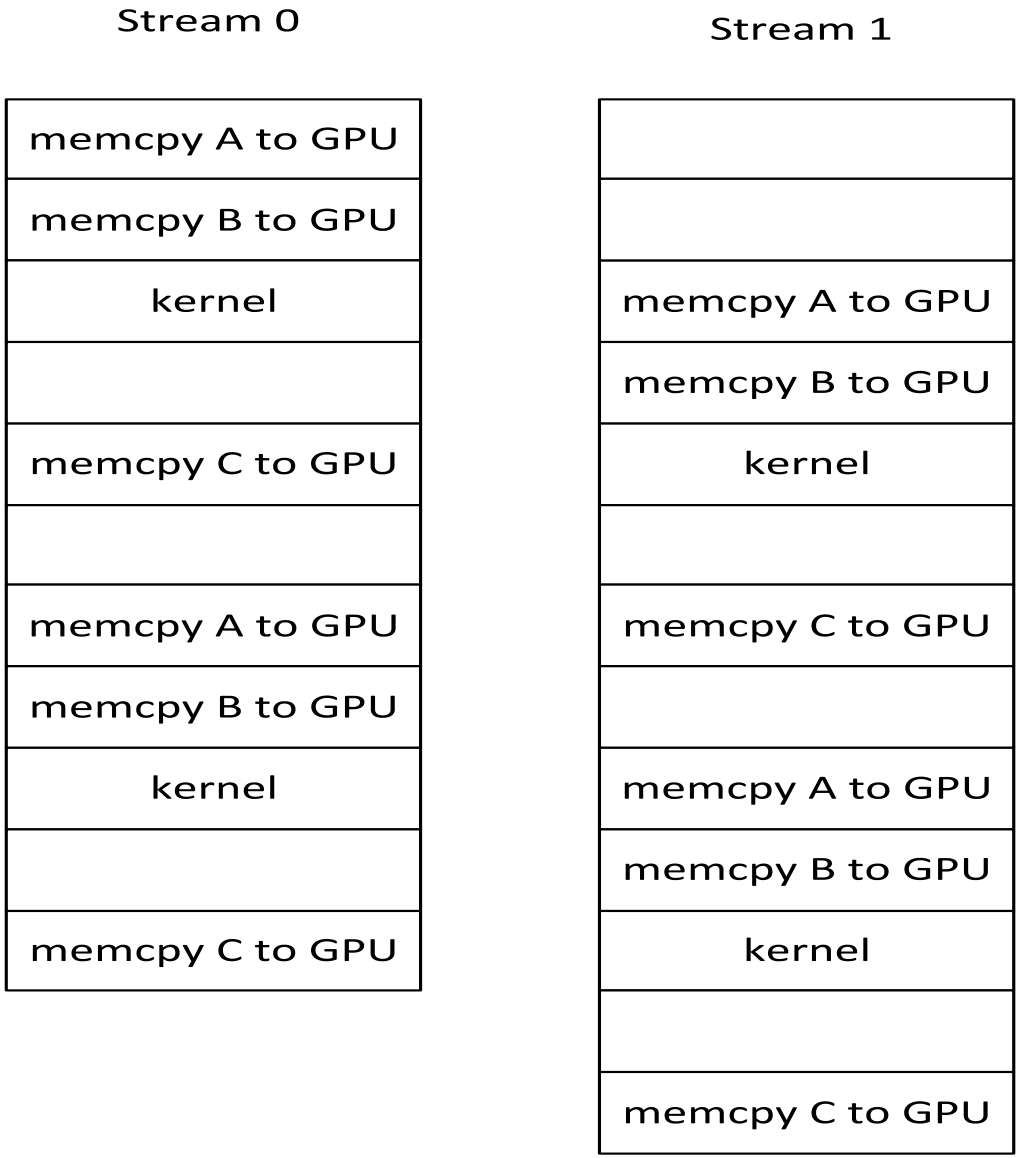
- Сперва целиком копируем оба массива CPU->GPU
- Выполняем ядро
- Целиком копируем выходной массив GPU->CPU

Потоки (пример)

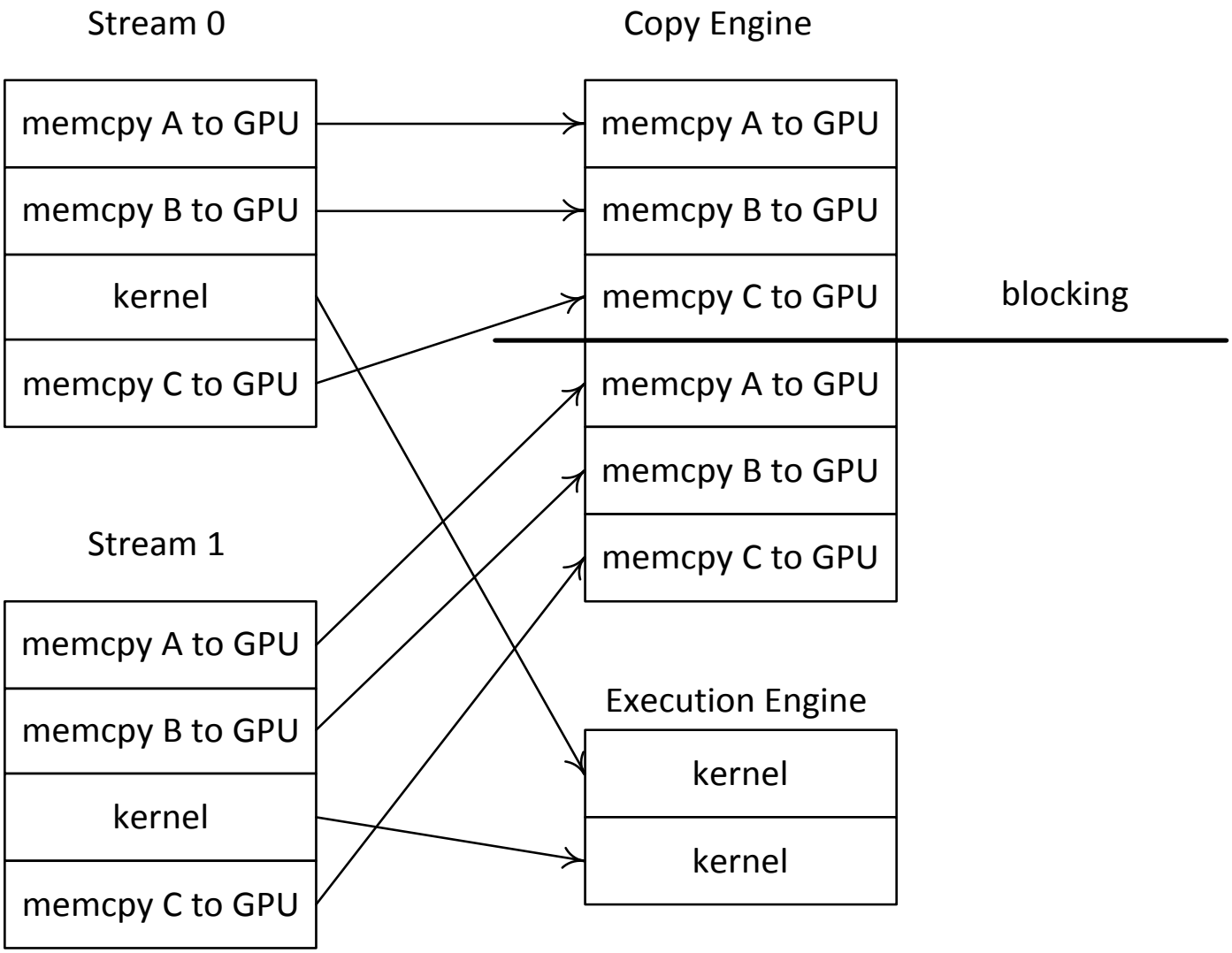
Для того, чтобы воспользоваться возможностью параллельного копирования и выполнения

- Выделим pinned-память
- Разобьем массивы на блоки
- Одновременно будем копировать два входных блока, выполнять ядро и копировать результат обратно

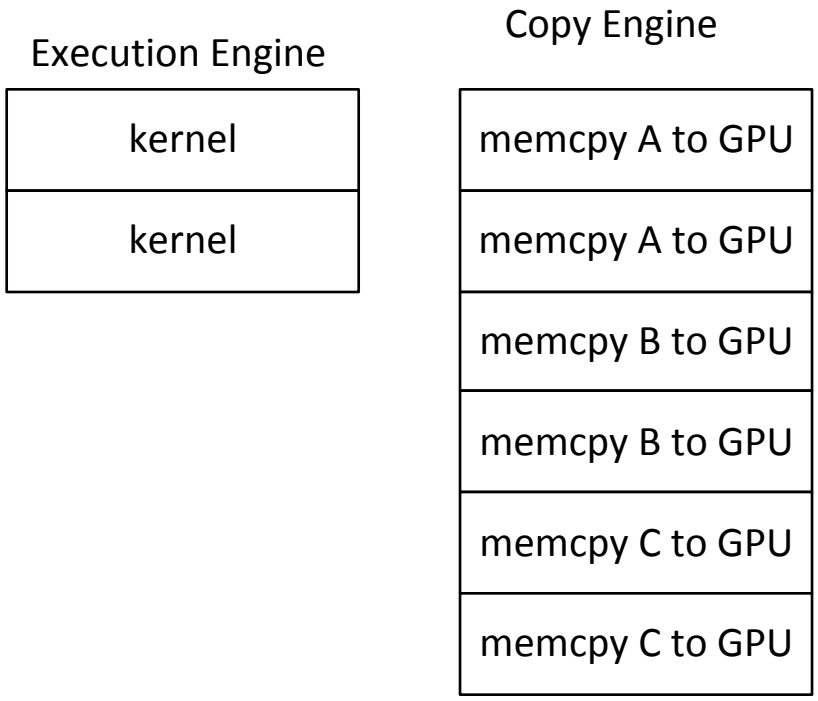
Потоки


Stream 0
Stream 1
memcpy A to GPU
memcpy B to GPU
kernel
memcpy C to GPU
memcpy A to GPU
memcpy B to GPU
kernel
memcpy C to GPU
memcpy A to GPU
memcpy B to GPU
kernel
memcpy C to GPU
memcpy A to GPU
memcpy B to GPU
kernel
memcpy C to GPU

Потоки - как все работает



Потоки - как нужно



Потоки - как нужно

```
for ( int i = 0; i < DATA_SIZE; i += N*2)
{
  cudaMemcpyAsync( devA0, hostA + i, N * sizeof (int),
                   cudaMemcpyHostToDevice, stream0 );
  cudaMemcpyAsync( devA1, hostA + i + N, N * sizeof (int),
                   cudaMemcpyHostToDevice, stream1 );

  cudaMemcpyAsync( devB0, hostB + i, N * sizeof (int),
                   cudaMemcpyHostToDevice, stream0 );
  cudaMemcpyAsync( devB1, hostB + i + N, N * sizeof (int),
                   cudaMemcpyHostToDevice, stream1 );

  kernel<<<blocks, threads, 0, stream0>>> ( devA0, devB0, devC0 );
  kernel<<<blocks, threads, 0, stream0>>> ( devA1, devB1, devC1 );

  cudaMemcpyAsync( hostC + i, devC0, N * sizeof (int),
                   cudaMemcpyDeviceToHost, stream0 );
  cudaMemcpyAsync( hostC + i + N, devC1, N * sizeof (int),
                   cudaMemcpyDeviceToHost, stream1 );
}
```

Интероперабельность с OpenGL

- Возможность непосредственно в CUDA использовать данные OpenGL без необходимости их копировать
  - Поддерживаются текстуры
  - Поддерживаются VBO
  - Соответствующий ресурс необходимо зарегистрировать вначале
  - Для отображения ресурса в адресное пространство CUDA используется соответствующие функции отображения

Интероперабельность с OpenGL

- При этом ресурс не может одновременно использоваться и CUDA и OpenGL – мы отображаем его в CUDA, работаем, потом закрываем отображение и OpenGL может снова его использовать
- В CUDA 8 есть некоторые изменения в этом механизме

Интероперабельность с OpenGL - VBO

```
class     CudaGlBuffer                                          // VBO
{
  cudaGraphicsResource * resource;
  VertexBuffer         * buffer;
  GLenum                 target;

public:
  CudaGlBuffer ( VertexBuffer * buf, GLenum theTarget,
                 unsigned int flags = cudaGraphicsMapFlagsWriteDiscard )
  {
          buffer = buf;
          target = theTarget;

          buffer -> bind ( target );
          cudaGraphicsGLRegisterBuffer ( &resource, buffer -> getId (), flags );
          buffer -> unbind ();
  }
  ~CudaGlBuffer ()
  {
          cudaGraphicsUnregisterResource ( resource );
  }
  bool    mapResource ( cudaStream_t stream = 0 )
  {
          return cudaGraphicsMapResources ( 1, &resource, stream ) == cudaSuccess;
  }
  bool    unmapResource ( cudaStream_t stream = 0 )
  {
          return cudaGraphicsUnmapResources ( 1, &resource, stream ) == cudaSuccess;
  }
```

Интероперабельность с OpenGL - VBO

```
  void * mappedPointer ( size_t& numBytes ) const
  {
          void * ptr;

          if ( cudaGraphicsResourceGetMappedPointer ( &ptr, &numBytes, resource )
               != cudaSuccess )
                    return NULL;

          return ptr;
  }

  GLuint  getId () const
  {
          return buffer -> getId ();
  }

  GLenum getTarget () const
  {
          return target;
  }

  cudaGraphicsResource * getResource () const
  {
          return resource;
  }
};
```

Интероперабельность с OpenGL - Текстуры

```
class     CudaGlImage
{
  GLuint                                    image;
  GLenum                                    target;
  cudaGraphicsResource * resource;

public:
  CudaGlImage ( GLuint theImage, GLenum theTarget,
    unsigned int  flags = cudaGraphicsMapFlagsWriteDiscard )
  {
          image  = theImage;
          target = theTarget;
          cudaGraphicsGLRegisterImage ( &resource, image, target, flags );
  }
  ~CudaGlImage ()
  {
          cudaGraphicsUnregisterResource ( resource );
  }

  bool    mapResource ( cudaStream_t stream = 0 )
  {
          return cudaGraphicsMapResources ( 1, &resource, stream ) == cudaSuccess;
  }

  bool    unmapResource ( cudaStream_t stream = 0 )
  {
          return cudaGraphicsUnmapResources ( 1, &resource, stream ) == cudaSuccess;
  }
```

Интероперабельность с OpenGL - Текстуры

```
cudaArray * mappedArray ( unsigned int index = 0, unsigned int mipLevel = 0 ) const
{
        cudaArray * array;

        if ( cudaGraphicsSubResourceGetMappedArray  ( &array, resource, index,
                                                      mipLevel ) != cudaSuccess )
                return NULL;

        return array;
}
```

Библиотека thrust

- Полностью на шаблонах
- Напоминает STL
- Свой namespace thrust
- Контейнеры, итераторы, алгоритмы (почти как STL)

Библиотека thrust

Все два типа контейнеров (самые эффективные с точки зрения доступа к памяти) – thrust::host_vector, thrust_device_vector

```
thrust::host_vector<int> hv ( 1000 );
thrust::device_vector<int> dv ( 1000 );
thrust::generate ( hv.begin (), hv.end (), sequence );
dv = hv;
int sum = thrust::reduce ( dv.begin (), dv.end () );
```

Библиотека thrust

Получение указателя по контейнеру

```
thrust::device_vector<int> v (1000);
int * rawPtr = thrust::raw_pointer_cast(v.data));
myKernel<<<threads, blocks>>> ( rawPtr, v.size () );
```

Можно в любой момент обращаться к любому элементу контейнера, не важно в какой памяти (скрытый cudaMemcpy)

Библиотека thrust

Итераторы

- begin(), end(), ++, --, арифместика
- Тип итератора несет в себе информацию о том, в чьей памяти
- Обычный указатель - тоже итератор, но только в память CPU
- Перевод device-указателя в итератор

```
thrust::device_ptr<int> ptr ( rawPtr );
```

Библиотека thrust

Простейшие операции, работают с памятью CPU и GPU

```
thrust::fill ( d.begin (), d.end (), 77 );
thrust::sequence ( d.begin (), d.end (), 7, -2 );
thrust::generate ( d.begin (), d.end (), rand );
thrust::copy ( src.begin (), src.end (), dst.begin () );
```

Библиотека thrust

Алгоритмы

- Принимают на вход функторы
- Есть набор стандартных функторов

```
thrust::transform ( a.begin (), a.end(),
  output.end (), operation );

thrust::transform ( a.begin (), a.end (), b.begin
  (), output.begin (), thrust::multiplies<float>
  () );

int sum = thrust::reduce( a.begin (), a.end (), 0,
  thrust::plus<int> () );
```

Библиотека thrust

```
template<typename T> struct square
{ __device__ __host__ T operator ()( const T& x )
  const { return x*x; }
};
float s = thrust::transform_reduce ( v.begin (),
 v.end (), square<float>(), 0,
  thrust::plus<float>() );
thrust::inclusive_scan ( x.begin (), x.end (),
  x.begin () );
thrust::exclusive_scan ( a.begin (), a.end (),
  b.begin () );
thrust::sort ( v.begin (), v.end () );
thrust::sort_by_key ( keys.begin (), keys.end (),
  data.begin () );
```

Библиотека thrust

Аналог лямбд

```
#include <thrust/functional.h>
using namespace thrust::placeholders;
thrust::for_each(x.begin(), x.end(), _1++);

int a = 42;
thrust::transform(x.begin(), x.end(), y.begin(),
                  x.begin(), a * _1 + _2);
```

Библиотека thrust

Есть еще много различных алгоритмов

Есть много специальных итераторов, например transform_iterator или zip_iterator