

Атомарные операции. Поток. Параллельное копирование и выполнение ядра. Интероперабельность с OpenGL

Лекторы:

[Боресков А.В. \(ВМиК МГУ\)](#)

[Харламов А.А. \(NVidia\)](#)

План

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

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
- CС 1.1 поддерживает целочисленные атомарные операции в глобальной памяти
- CС 1.1 поддерживает целочисленные атомарные операции в разделяемрй памяти

Атомарные операции

- Над 64-битовыми целыми с CС 2.0
- `atomicAdd` для `float` - CС 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, возвращает 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 в 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

$$c[i] = \text{foo}(a[i], b[i])$$

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

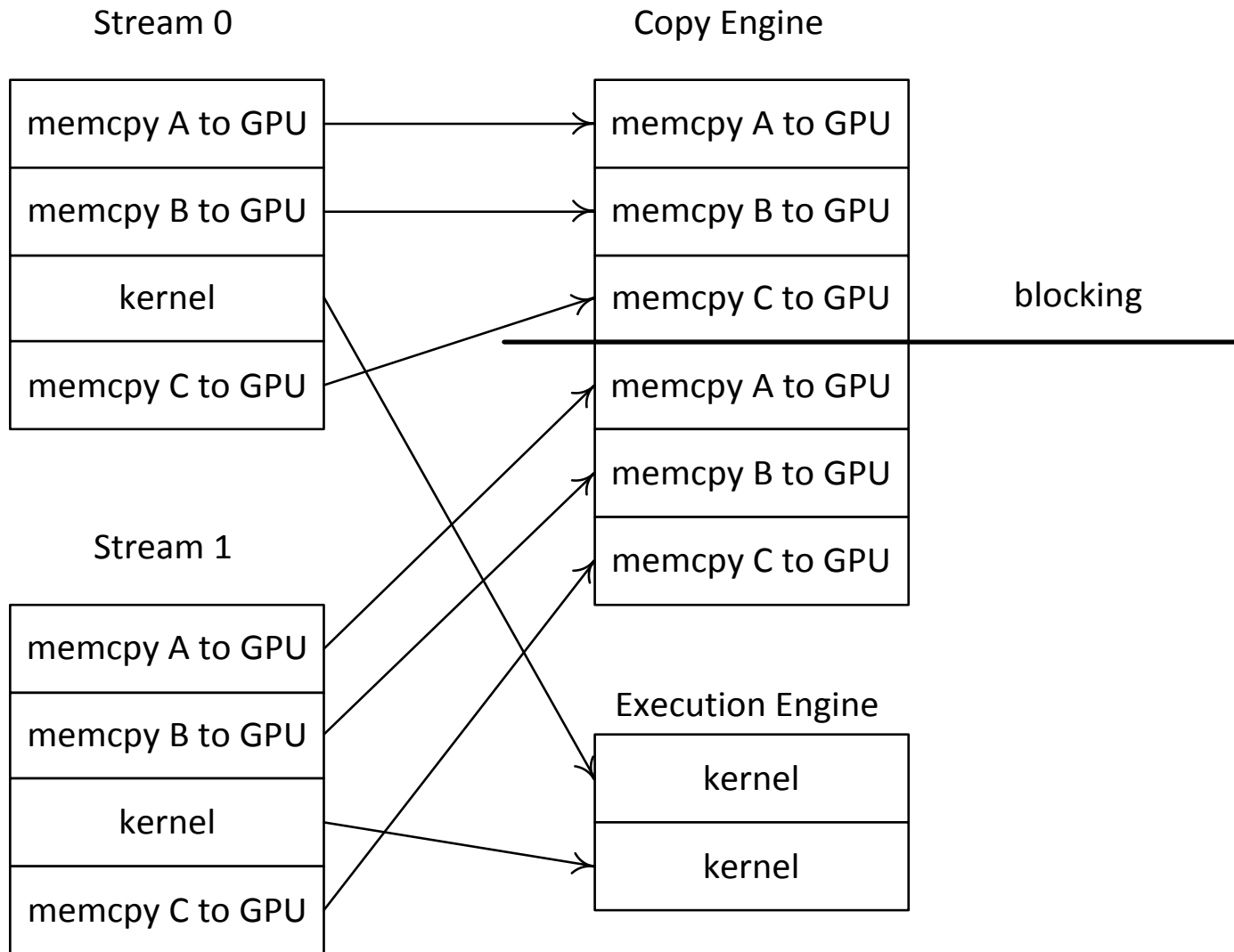
Stream 0

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

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

ПОТОКИ - как все работает



Потоки - как нужно

Execution Engine

kernel
kernel

Copy Engine

memcpy A to GPU
memcpy A to GPU
memcpy B to GPU
memcpy B to GPU
memcpy C to GPU
memcpy C to GPU

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 - 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;
}
```

Ресурсы нашего курса

- [Steps3d.Narod.Ru](#)
- [Google Site CUDA.CS.MSU.SU](#)
- [Google Group CUDA.CS.MSU.SU](#)
- [Google Mail CS.MSU.SU](#)
- [Google SVN](#)
- [Tesla.Parallel.Ru](#)
- [Twirpx.Com](#)
- [Nvidia.Ru](#)