CUDA STREAMS. ИСПОЛЬЗОВАНИЕ CUDA STREAMS ДЛЯ РЕАЛИЗАЦИИ АСИНХРОННЫХ ОПЕРАЦИЙ.

АФАНАСЬЕВ ИЛЬЯ
AFANASIEV_ILYA@ICLOUD.COM

ПРЕДПОСЫЛКА

- Время работы GPU приложения = Время выполнения ядер + **время копирования данных**
- 2 подхода:

1. ускорение непосредственно копирований (Pinned memory)
2. **выполнение копирований параллельно с вычислениями (и не только, CUDA streams)**

ПЛАН ЛЕКЦИИ

- Определения и основные понятия, связанные с CUDA-потоками
- Примеры создания и исполнения CUDA-потоков
- Синхронизации CUDA-потоков
- Когда следует использовать CUDA-потоки?
- Выводы

CUDA STREAMS, ОПРЕДЕЛЕНИЕ

- **CUDA Stream** - последовательность команд для GPU (запуски ядер, копирования памяти и т.д.), исполняемая строго последовательно
- По-умолчанию, все команды помещаются в «Default Stream», равный нулю (именно в нём мы до этого и работали)
- Однако, пользователь может сам создавать новые потоки и распределять команды между ними
- Только команды из разных потоков, отличных от потока по-умолчанию, могут выполняться параллельно
- В общем случае порядок выполнения команд из различных потоков не определен (без явных синхронизаций, выполняемых пользователем)

СОЗДАНИЕ CUDA ПОТОКОВ

- 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
- Пример создания потоков:

```
cudaStream_t stream;

cudaStreamCreate(&stream);

// so some job

cudaStreamDestroy(stream);
```

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

ПАРАЛЛЕЛЬНАЯ РАБОТА НА СИСТЕМАХ С GPU

- Параллельная работа хоста и устройства
- Параллельная работа нескольких GPU
- Параллельное выполнение различных ядер на одном устройстве

ПАРАЛЛЕЛЬНАЯ РАБОТА ХОСТА И УСТРОЙСТВА

- Ядра выполняются асинхронно по умолчанию
- Копирование между pinned-памятью и памятью устройства при помощи cudaMemcpyAsync также выполняется асинхронно
- Добиться параллельной работы хоста и устройства несложно даже без использования потоков:

ПАРАЛЛЕЛЬНОЕ ВЫПОЛНЕНИЕ КОМАНД НА GPU

- Команды из разных потоков, отличных от потока по-умолчанию, могут исполняться параллельно
- Возможные случаи:
  1. Параллельные копирование и выполнение ядра
  2. Параллельные выполнение ядер
  3. Параллельные копирования с хоста на устройство и с устройства на хост
- Рассмотрим далее примеры для каждого из случаев

ПРИМЕР ПАРАЛЛЕЛЬНОГО КОПИРОВАНИЯ И ЗАПУСКА ЯДРА

- Пример:

```
cudaStream_t stream1 , stream2 ; // создаем переменные, соотвествующие каждому потоку

cudaStreamCreate(&stream1); // инициализируем потоки

cudaStreamCreate(&stream2); // инициализируем потоки

// начинаем копирование в потоке №1
cudaMemcpyAsync(data_d,data_h,size ,cudaMemcpyHostToDevice,stream1);

// начинаем вычсиления в потоке №2

kernel<<<grid , block , 0, stream2 >>>(...);

cudaStreamDestroy(stream1 );

cudaStreamDestroy(stream2 );
```

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

ПРИМЕР ПАРАЛЛЕЛЬНОГО ВЫПОЛНЕНИЯ ЯДЕР

- Пример:

```
cudaStreamCreate(&stream1);

cudaStreamCreate(&stream2);

kernel1<<<grid, block, 0, stream1>>>(data_1);

kernel2<<<grid, block, 0, stream2>>>(data_2);
```

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

ПРИМЕР ПАРАЛЛЕЛЬНОГО КОПИРОВАНИЯ В ОБЕ СТОРОНЫ

- Пример:

```
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>>>(...);
```

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

СИНХРОНИЗАЦИИ ПОТОКОВ

- Синхронизировать все потоки
  - при помощи функции cudaDeviceSynchronize()
  - блокирует хост пока все CUDA- операции не будут выполнены
- Синхронизировать конкретный поток
  - при помощи функции cudaStreamSynchronize(stream_id)
  - блокирует хост пока все CUDA-операции в указанном потоке не будут выполнены
- Синхронизация при помощи событий (CUDA events)
  - Создаются специальные CUDA-события внутри потоков, в дальнейшем используемые при синхронизации
  - cudaEventRecord (event, streamid)
  - cudaEventSynchronize (event)
  - cudaStreamWaitEvent (stream, event)
  - cudaEventQuery(event)

CUDA EVENTS

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

ФУНКЦИИ, СВЯЗАННЫЕ С СОБЫТИЯМИ

- Функция *cudaError_t cudaEventRecord (cudaEvent_t* event, *cudaStream_t stream = 0)* приписывает событие к точке программы в потоке stream, в которой вызывается

- Функция *cudaError_t cudaEventQuery(cudaEvent_t* event*)* возвращает cudaSuccess, если событие уже произошло (вся работа до последнего cudaEventRecord выполнена): иначе cudaErrorNotReady

- Функция cudaError_t cudaEventSynchronize (cudaEvent_t event) возвращает управление хостовой нити только после наступления события

- Функция *cudaError_t cudaStreamWaitEvent (cudaStream_t* stream, *cudaEvent_t* event, *unsigned int flags)* заставляет команды, Команды, отправленные в stream, начать выполняться после наступления события event

СИНХРОНИЗАЦИЯ ПО ПОТОКУ

- Функция cudaError_t cudaStreamQuery (cudaStream_t stream) возвращает cudaSuccess, если выполнены все команды в потоке stream, иначе cudaErrorNotReady

- Функция cudaError_t cudaStreamSynchronize (cudaStream_t stream) возвращает управление хостовой нити, когда завершится выполнение всех команд, отправленных в поток stream

ПРИМЕР СИНХРОНИЗАЦИИ ПОТОКОВ ПРИ ПОМОЩИ СОБЫТИЙ

```
cudaEvent_t event;

cudaEventCreate (&event);  // create event

cudaMemcpyAsync ( d_in, in, size, H2D, stream1 ); // 1) H2D copy of new input

cudaEventRecord (event, stream1); // record event

cudaMemcpyAsync ( out, d_out, size, D2H, stream2 ); // 2) D2H copy of previous result

cudaStreamWaitEvent ( stream2, event ); // wait for event in stream1

kernel <<< , , , stream2 >>> ( d_in, d_out );  // 3) must wait for 1 and 2

asynchronousCPUmethod ( ... ) // Async GPU method
```

ИСПОЛЬЗОВАНИЕ CUDA СОБЫТИЙ ДЛЯ ЗАМЕРА ВРЕМЕНИ ВЫПОЛНЕНИЯ ЯДЕР

```
float time_ms = 0.0f;

cudaEvent_t start , stop ;

cudaEventCreate( &start );

cudaEventCreate( &stop );

cudaEventRecord( start , 0 ); // записываем событие-начало

kernel<<< ... , ... , 0, 0 >>>( ... );

cudaEventRecord( stop , 0 );  // записываем событие-конец

cudaEventSynchronize(start );

cudaEventSynchronize( stop ); // не обязательно

cudaEventElapsedTime( &time_ms, start , stop );  // записываем разницу по времени между событиями в переменную time_ms

cudaEventDestroy( start ); cudaEventDestroy( stop ); // не забываем уничтожить созданные события
```

НЕЯВНЫЕ СИНХРОНИЗАЦИИ

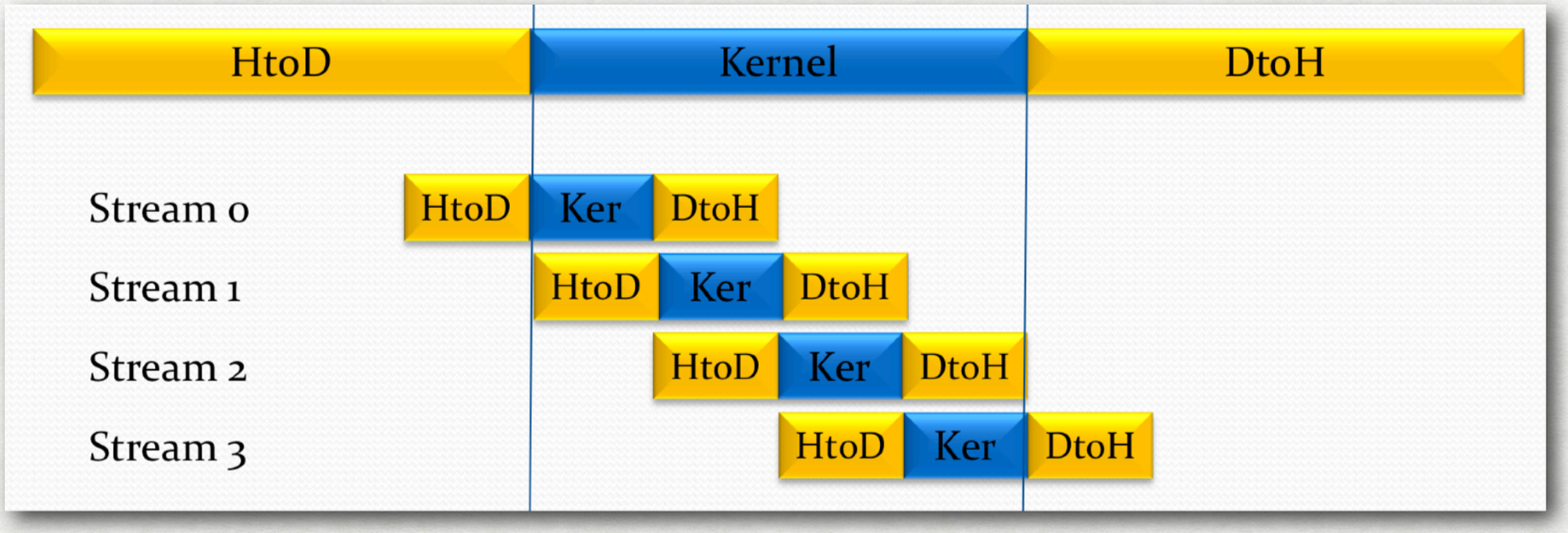
- Следующие операции неявно синхронизируют все другие CUDA-операции:
  - Page-locked выделения памяти - cudaMallocHost(), cudaHostAlloc()
  - Выделения памяти на устройстве - cudaMalloc ()
  - Синхронные операции копирований данных - cudaMemcpy* (без асинхронного суффикса) cudaMemset* (без асинхронного суффикса)
  - Изменение конфигурации L1 кэша и разделяемой памяти при помощи функции cudaDeviceSetCacheConfig ()

КОГДА СЛЕДУЕТ ПРИМЕНЯТЬ ПОТОКИ? ПРОСТОЙ СЛУЧАЙ

- Пусть в нашей задаче время копирований данных и время выполнения ядра сопоставимо
- Тогда можно разделить грид на части и запустим то же ядро на подгридах – результат не изменится
- Каждой подзадаче нужна только часть данных для старта
- Задачи

Было:

Стало:

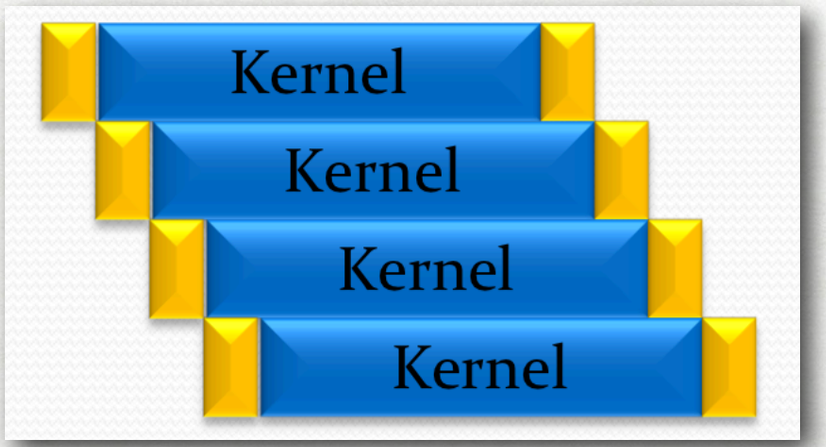


ПРИМЕР С НЕБОЛЬШИМИ КОПИРОВАНИЯМИ

- Ускорение в предыдущем случае достигается только за счет совмещения копирований и вычислений
- В случае, если вычислений значительно больше, чем копирований - ускорения получить не удасться, так как вычисления не станут производиться быстрее только за счет разбиения вычислительной части на подзадачи

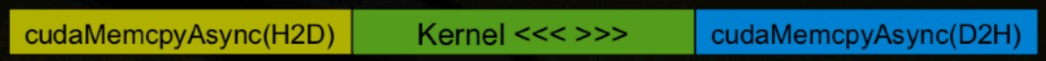
Было:

Стало:



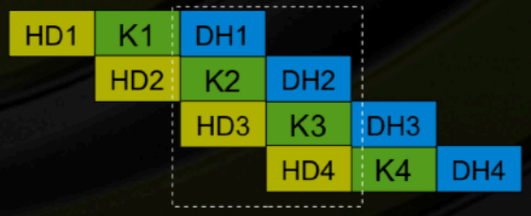
AMOUNT OF CONCURRENCY

- **Serial** (1x)

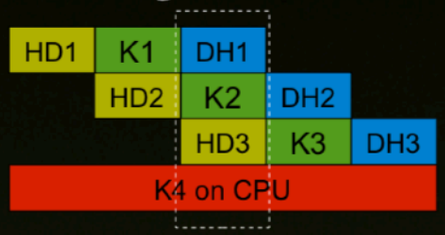


- **2-way concurrency** (up to 2x)

- **3-way concurrency** (up to 3x)



- **4-way concurrency** (3x+)



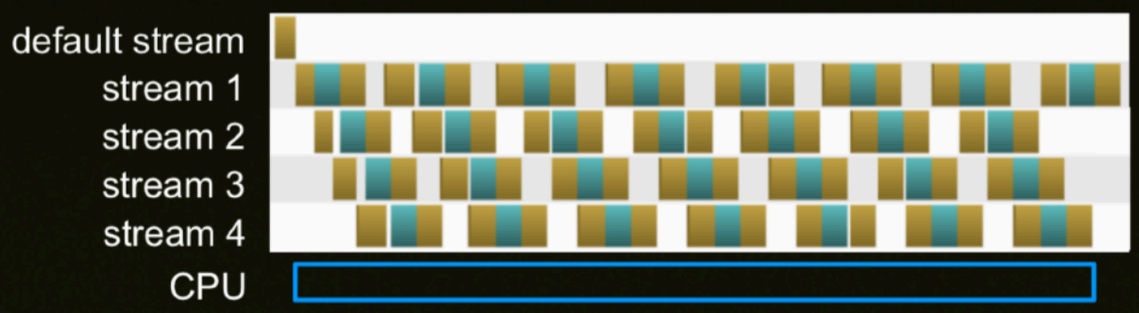
- **4+ way concurrency**

TILED DGEMM

- **CPU** (4core Westmere x5670 @2.93 GHz, MKL)
  - **43 Gflops**
- **GPU** (C2070)
  - **Serial : 125 Gflops (2.9x)**
  - **2-way : 177 Gflops (4.1x)**
  - **3-way : 262 Gfllops (6.1x)**
- **GPU + CPU**
  - **4-way con.: 282 Gflops (6.6x)**
  - **Up to 330 Gflops for larger rank**
- **Obtain maximum performance by leveraging concurrency**
- **All communication hidden – effectively removes device memory size limitation**

DGEMM: m=n=8192, k=288

Nvidia Visual Profiler (nvvp)



ВЫВОДЫ

- В случае, если в программе мало копирований с запусками ядер на больших гридах - не стоит пытаться ускорить программу за счет использования потоков