

ICNPG 2023

Clase 12: Streams, eventos, nvprof, nvvp



Clases de C. Bederian (FaMAF)

Motivación

- La GPU queda muy “lejos” del resto del sistema
 - PCI Express 3.0 16X: **15.75 GB/s, bidireccional**
- ⇒ Necesidad de ocultar esta latencia
- La GPU tiene poca memoria
 - ⇒ Necesidad de subir → procesar → bajar partes de sistemas grandes
 - Si el kernel es computacionalmente intensivo (ej: MM), puede operarse sin pérdida de performance

Motivación (cont.)

- ④ Necesidad de concurrencia entre kernels dentro de una GPU
 - A veces una GPU tiene demasiados recursos de ejecución para correr un único kernel
 - Queremos poder correr múltiples kernels en paralelo

Hasta ahora todo lo que hicimos fue mandar en serie un kernel, pero podríamos haber mandado más de uno

Kernels concurrentes

- Secuencial (así trabajamos hasta ahora)



- Cuatro en paralelo



El nro de kernels que se pueden hacer en paralelo depende de la GPU, para las nuevas es mucho mayor

Hasta ahora...

```
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
```

```
some_kernel<<<1,N>>>(d_a)
```

```
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

CPU - **NON** Blocking

CPU - GPU Blocking

Todos los Memcpy son puntos de sincronización entre la CPU y la GPU

Uno quisiera tener copias asincrónicas

Overlapping CPU - GPU

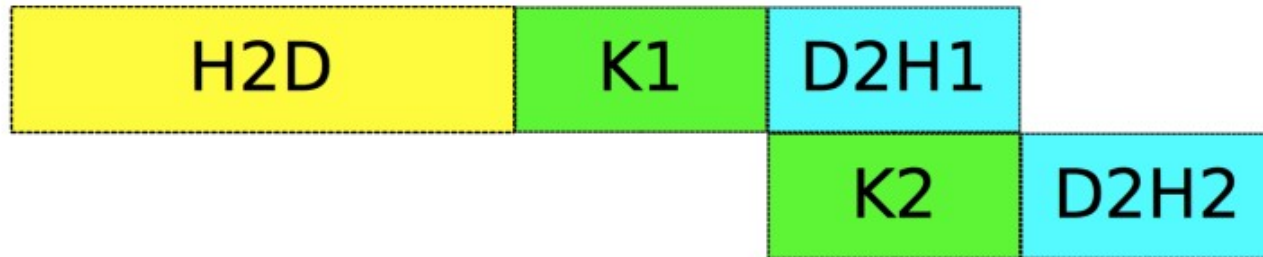
```
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);  
some_kernel<<<1,N>>>(d_a)  
some_cpu_work(a)  
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

Ocultamiento de latencia

Secuencial

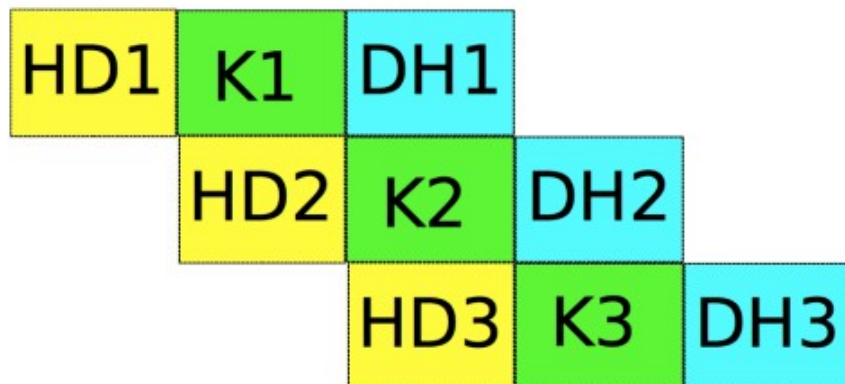


Copia y cómputo (GeForce)



Distintas
generaciones
de GPUs

Doble copia y cómputo (>Tesla)



Streams

son un

▣ Modelo de concurrencia dentro de la GPU

Cada stream son

- Colas de tareas que se ejecutan secuencialmente en el orden que se despachan
- Todos los streams se ejecutan concurrentemente (si pueden)

Streams – Creación/Destrucción

En programas grandes uno debería poner esto en todo (incluso en los Memcpy)

 `cudaError_t cudaStreamCreate(cudaStream_t* s)`

- Crea un nuevo stream y lo devuelve en *s
- Devuelve un `cudaError_t` como todo CUDA
 - ¡Verificar el resultado! (nunca está de más recordarlo)

`cudaStreamDestroy(cudaStream_t stream)`

- Espera que stream termine su trabajo y lo destruye.

Para liberar memoria

Streams - Ejecución

Encolar un kernel en un stream:

`kernel <<<grid, block, shared, stream>>> (...)`

¡Parámetros de configuración nuevos!

shared: Memoria compartida definida en tiempo de ejecución

Si usamos memoria shared, hay que hacer la

- Declaración en el kernel: `extern __shared__ T buf[];`
- Por defecto: 0 bytes

stream: Stream donde encolar el kernel.

- Por defecto: Stream 0 ("null stream")
 - Creado automáticamente
 - Semántica distinta a la del resto de los streams

Acá fue a parar todo lo que hicimos hasta ahora. Siempre estuvimos usando stream, el Stream 0

Streams – Memoria

Copias asincrónicas! No se bloquea la GPU

- Encolar un memset:

`cudaMemset*Async(... , cudaStream_t stream)`

- Encolar una transferencia:

`cudaMemcpy*Async(... , cudaStream_t stream)`

OJO: La memoria del host debe ser **pinned** para que funcione

La memoria alojada en el Host no puede ser la misma de siempre, hay que usar una memoria llamada "anclada" o "pinned". Esto se hace en la reserva de memoria al crear los arrays. ¿Por qué es necesario hacer esto? Porque el sistema operativo tiene la capacidad de mover cosas en la memoria RAM, lo cual generaría problemas a la hora de hacer copias asincrónicas. Pierde esa capacidad en las memorias pinned

Pinned memory

Las copias asíncronas son manejadas por un DMA engine en la GPU

¡Pero el sistema operativo puede mudar las páginas de memoria!

CUDA provee funciones para usar memoria no paginable:

`cudaMallocHost (void ** ptr, size_t size)`

`cudaFreeHost (void * ptr)`

- Uso similar a `cudaMalloc/cudaFree` pero para memoria del host
- Copias más rápidas

porque se "ahorran pasos" respecto a la copia normal `cudaMemcpy`

¡Usar con precaución!

porque uno le está poniendo trabas al sistema operativo

Streams – Sincronización

Esperar que termine un stream (como `cudaDeviceSynchronize`):

`cudaStreamSynchronize(cudaStream_t s)`

El stream particular se sincroniza con la CPU

Consultar si un stream terminó (sin bloquear):

`cudaError_t` `cudaStreamQuery(cudaStream_t s)`

Todas las funciones dan un mensaje de error si sale algo mal. Uno decide dónde capturarlo. Si no lo captura, el error puede saltar o no al momento de la ejecución

Streams - Eventos

- Esperar que termine un stream o todo el trabajo encolado en la GPU no es suficientemente fino
- Los eventos se introducen para cubrir esta necesidad
 - Marcadores que se insertan entre operaciones de un stream
 - Funciones de sincronización de eventos

Se torna compleja la programación, pero se vuelve más eficiente

Streams – Eventos (cont.)

Creación

```
cudaEvent_t event;  
cudaEventCreate(&event);
```

Destrucción

```
cudaEventDestroy(event);
```

Inserción en un stream

```
cudaEventRecord(event, stream);
```


Streams – Eventos (cont.)

Ver si ocurrió un evento

```
cudaError_t status = cudaEventQuery(event);
```

Esperar que ocurra un evento

```
cudaEventSynchronize(event);
```

Hacer que un stream espere que se dé un evento antes de seguir:

```
cudaStreamWaitEvent(stream, event, 0);
```

- **Esto ocurre sin intervención del host**

Streams – Cosas a evitar

Stream nulo

Sincroniza el device después de cada operación

Sincroniza el host después de cada operación, salvo algunas excepciones

Sincronización implícita

`cudaMallocHost / cudaHostAlloc`

`cudaMalloc`

`cudaMemcpy* / cudaMemcpyAsync` (no Async)

...

OJO: Estamos acostumbrados a esta falta de concurrencia

```
cudaStream_t stream[3];
```

```
for (int i = 0; i < 3; ++i)
```

```
    cudaStreamCreate(&stream[i]);
```

```
for (int i = 0; i < 3; ++i)
```

Se pasan distintos datos

```
    kernel_A <<<grid, block, 0, stream[i]>>> (...);
```

```
for (int i = 0; i < 3; ++i)
```

```
    kernel_B <<<grid, block, 0, stream[i]>>> (...);
```

```
for (int i = 0; i < 3; ++i)
```

```
    kernel_C <<<grid, block, 0, stream[i]>>> (...);
```

```
cudaDeviceSynchronize(); ~ "Barrera mayor": se espera a que terminen todos los streams
```

```
for (int i = 0; i < 3; ++i)
```

```
    cudaStreamDestroy(stream[i]);
```

KA1	KB1	KC1
KA2	KB2	KC2
KA3	KB3	KC3



Streams – Bibliotecas

Encolan sus kernels en el stream que se les configure

CUFFT

```
cufftResult cufftSetStream(cufftHandle plan,cudaStream_t stream);
```

CUBLAS

```
cublasStatus_t cublasSetStream(cublasHandle_t handle,cudaStream_t stream);
```

CUSPARSE

```
cusparseStatus_t cusparseSetStream(cusparseHandle_t h,cudaStream_t stream);
```

Thrust 1.8:

streams, new execution policies, dynamic parallelism, etc.



[Get Started](#) [Documentation](#) [Community](#) [Get Thrust](#)

Thrust v1.8.0 release

We are pleased to announce the release of [Thrust](#) v1.8, an open-source C++ library for developing high-performance parallel applications. Modeled after the C++ Standard Template Library, Thrust brings a familiar abstraction layer to the realm of parallel computing

Thrust 1.8.0 introduces support for algorithm invocation from CUDA `__device__` code, support for CUDA streams, and algorithm performance improvements. Users may now invoke Thrust algorithms from CUDA `__device__` code, providing a parallel algorithms library to CUDA programmers authoring custom kernels, as well as allowing Thrust programmers to nest their algorithm calls within functors. The `thrust::seq` execution policy allows users to require sequential algorithm execution in the calling thread and makes a sequential algorithms library available to individual CUDA threads. The `.on(stream)` syntax allows users to request a CUDA stream for kernels launched during algorithm execution. Finally, new CUDA algorithm implementations provide substantial performance improvements.

(desde Thrust 1.8.1 CUDA Toolkit 7.0 → Thrust 1.12.1 en CUDA Toolkit 11.4)

Streams en python CUDA

Low-level CUDA support

Device management

`cupy.cuda.Device([device])` Object that represents a CUDA device.

Memory management

`cupy.get_default_memory_pool()` Returns CuPy default memory pool for GPU memory.

`cupy.get_default_pinned_memory_pool()` Returns CuPy default memory pool for pinned memory.

`cupy.cuda.Memory(size_t size)` Memory allocation on a CUDA device.

`cupy.cuda.MemoryAsync(size_t size, ...)` Asynchronous memory allocation on a CUDA device.

`cupy.cuda.ManagedMemory(size_t size)` Managed memory (Unified memory) allocation on a CUDA device.

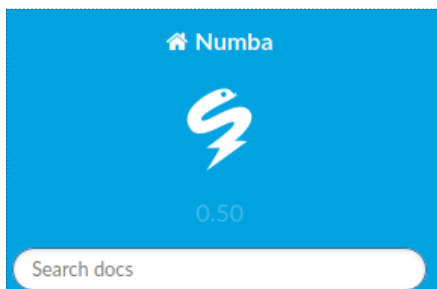
PyCUDA 2021.1
documentation

Q Search

Concurrency and Streams

`class pycuda.driver.Stream(flags=0)`

A handle for a queue of operations that will be carried out in order.



Stream Management

Streams allow concurrency of execution on a single device with items in different **streams** may execute concurrently. Most operations involve data transfers and kernel execution. For further details on stream management, see the [Numba documentation](#).

Streams are instances of `numba.cuda.cudadrv.driver.Stream`.

Memoria Anclada (Pinned Memory)

- *La función de la librería de C **malloc**, aloca “standard, pageable host memory”. El OS es libre de mudarla de aquí para allá...*
- *La función de Cuda C **cudaHostAlloc()** aloca “pinned memoria” en el host. Esencial: ¡El OS nos promete que no la mudará!*
- *Cuando copiamos H2D con memoria standard de host, el CUDA driver usa DMA para transferir primero a un buffer anclado, y de ahí a la GPU → **¡Usar memoria de host anclada debería ser más eficiente!***

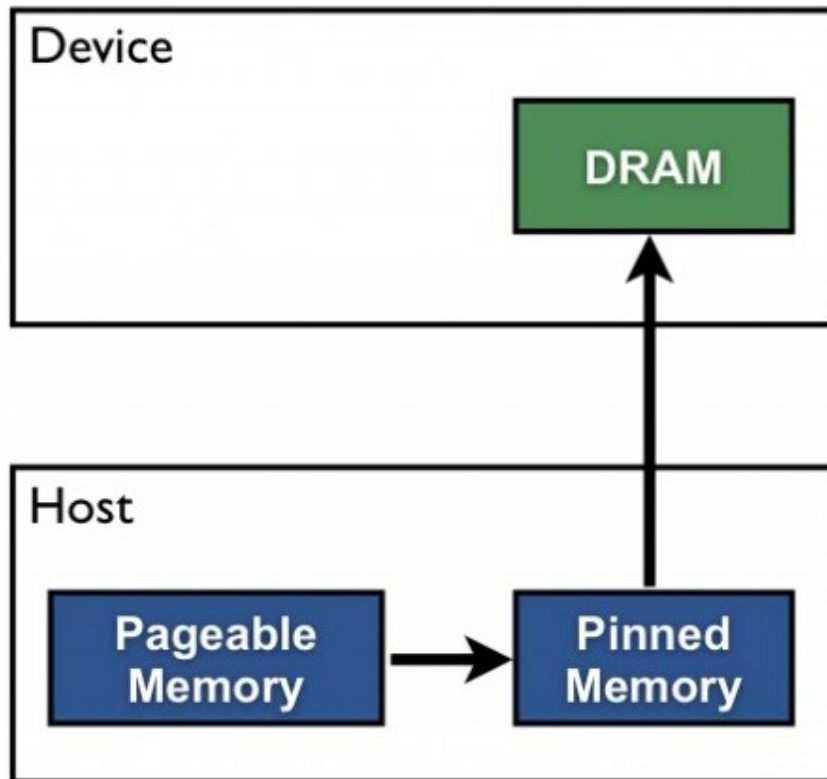
Se hace eso para asegurarse que en cualquier copia el OS no mueva ((los datos))

Hands on: Memoria Anclada (Pinned Memory)

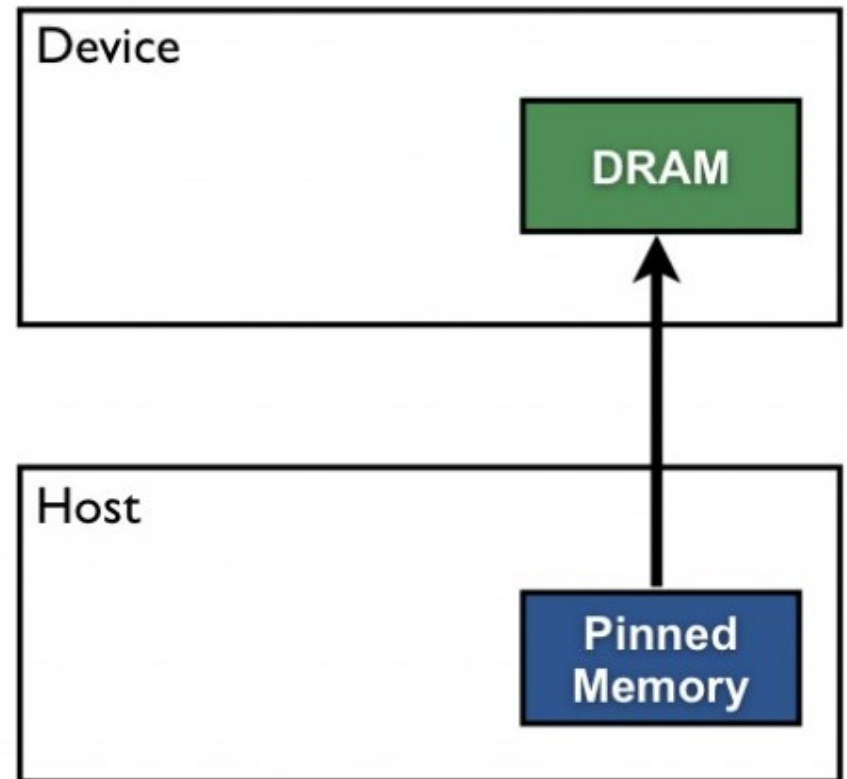
\$ CBE/copy_timed.cu

- Hace 100 copias de 64x1024x1024 enteros H2D/D2H.
- Performance: **cudaHostAlloc vs malloc**

Pageable Data Transfer



Pinned Data Transfer



Streams

BEST PRACTICES AND COMMON PITFALLS

Justin Luitjens - NVIDIA

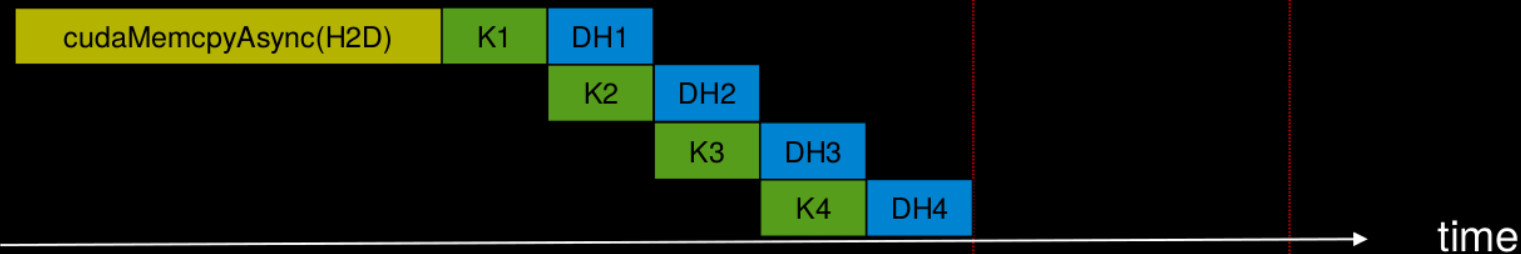
GPU TECHNOLOGY CONFERENCE

CONCURRENCY THROUGH PIPELINING

Serial



Concurrent- overlap kernel and D2H copy



$$N_{streams} = N/M$$

Ver `async.cu`, clase `streams...`

Para ejecutar el ejemplo hay que usar el modulo `cuda` en su versión 11 para que corra el profiler

$$a[N] \rightarrow \{a[0 : M - 1], a[M : 2M - 1], \dots, a[N - M : N - 1]\}$$

$$a[i] = a[i] + \sqrt{\sin^2(i) + \cos^2(i)}$$

Chunquificator: 1 stream, copias asincronicas

```
// now loop over full data, in bite-sized chunks
for (int i=0; i<FULL_DATA_SIZE; i+= N) {
    // copy the locked memory to the device, async
    HANDLE_ERROR( cudaMemcpyAsync( dev_a, host_a+i,
                                   N * sizeof(int),
                                   cudaMemcpyHostToDevice,
                                   stream ) );

    HANDLE_ERROR( cudaMemcpyAsync( dev_b, host_b+i,
                                   N * sizeof(int),
                                   cudaMemcpyHostToDevice,
                                   stream ) );

    kernel<<<N/256,256,0,stream>>>( dev_a, dev_b, dev_c );

    // copy the data from device to locked memory
    HANDLE_ERROR( cudaMemcpyAsync( host_c+i, dev_c,
                                   N * sizeof(int),
                                   cudaMemcpyDeviceToHost,
                                   stream ) );
}
```

\$ cbe/basic_single_stream.cu

Ejemplos de Streams

$$a[i] = a[i] + \sqrt{\sin^2(i) + \cos^2(i)}$$

$$a[N] \rightarrow \{a[0 : M - 1], a[M : 2M - 1], \dots, a[N - M : N - 1]\}$$

- `async.cu`, `async_thrust.cu`

Chunk0

Chunk1

Chunknstreams

```
for (int i = 0; i < nStreams; ++i) {
    int offset = i * streamSize;
    checkCuda( cudaMemcpyAsync(&d_a[offset], &a[offset],
                              streamBytes, cudaMemcpyHostToDevice,
                              stream[i]) );
    kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);
    checkCuda( cudaMemcpyAsync(&a[offset], &d_a[offset],
                              streamBytes, cudaMemcpyDeviceToHost,
                              stream[i]) );
}
```

(H2D-K-D2H)

(H2D-K-D2H)

(H2D-K-D2H)

Streams con Thrust

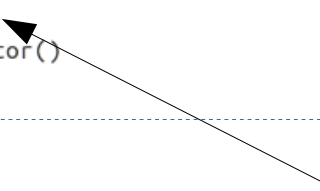
thrust::cuda::par is the parallel execution policy associated with Thrust's CUDA backend system.

Ahora puede tomar como argumento el stream!

Notación:

`thrust::cuda::par.on(stream[i]),`

```
// baseline case - sequential transfer and execute
memset(a, 0, bytes);
checkCuda( cudaEventRecord(startEvent,0) );
checkCuda( cudaMemcpy(d_a, a, bytes, cudaMemcpyHostToDevice) );
#ifdef THRUST18
kernel<<<n/blockSize, blockSize>>>(d_a, 0);
#else
    thrust::transform(
        thrust::cuda::par,
        thrust::make_counting_iterator(0),
        thrust::make_counting_iterator(n),
        d_a,
        d_a,
        functor()
    );
#endif
```



On the other hand, it's safe to use `thrust::cuda::par` with raw pointers allocated by `cudaMalloc`, even when the pointer isn't wrapped by `thrust::device_ptr`:

```
__global__ void kernel(float *a, int offset)
{
    int i = offset + threadIdx.x + blockIdx.x*blockDim.x;
    float x = (float)i;
    float s = sinf(x);
    float c = cosf(x);
    a[i] = a[i] + sqrtf(s*s+c*c);
}
//AGREGADO////////////////////////////////////
struct functor
{
    __device__
    float operator()(int i, float a)
    {
        float x = (float)(i);
        float s = sinf(x);
        float c = cosf(x);
        return a + sqrtf(s*s+c*c);
    }
};
```

```
// asynchronous version 1: loop over {copy, kernel, copy}
memset(a, 0, bytes);
checkCuda( cudaEventRecord(startEvent,0) );
for (int i = 0; i < nStreams; ++i) {
    int offset = i * streamSize;
    checkCuda( cudaMemcpyAsync(&d_a[offset], &a[offset],
                                streamBytes, cudaMemcpyHostToDevice,
                                stream[i]) );

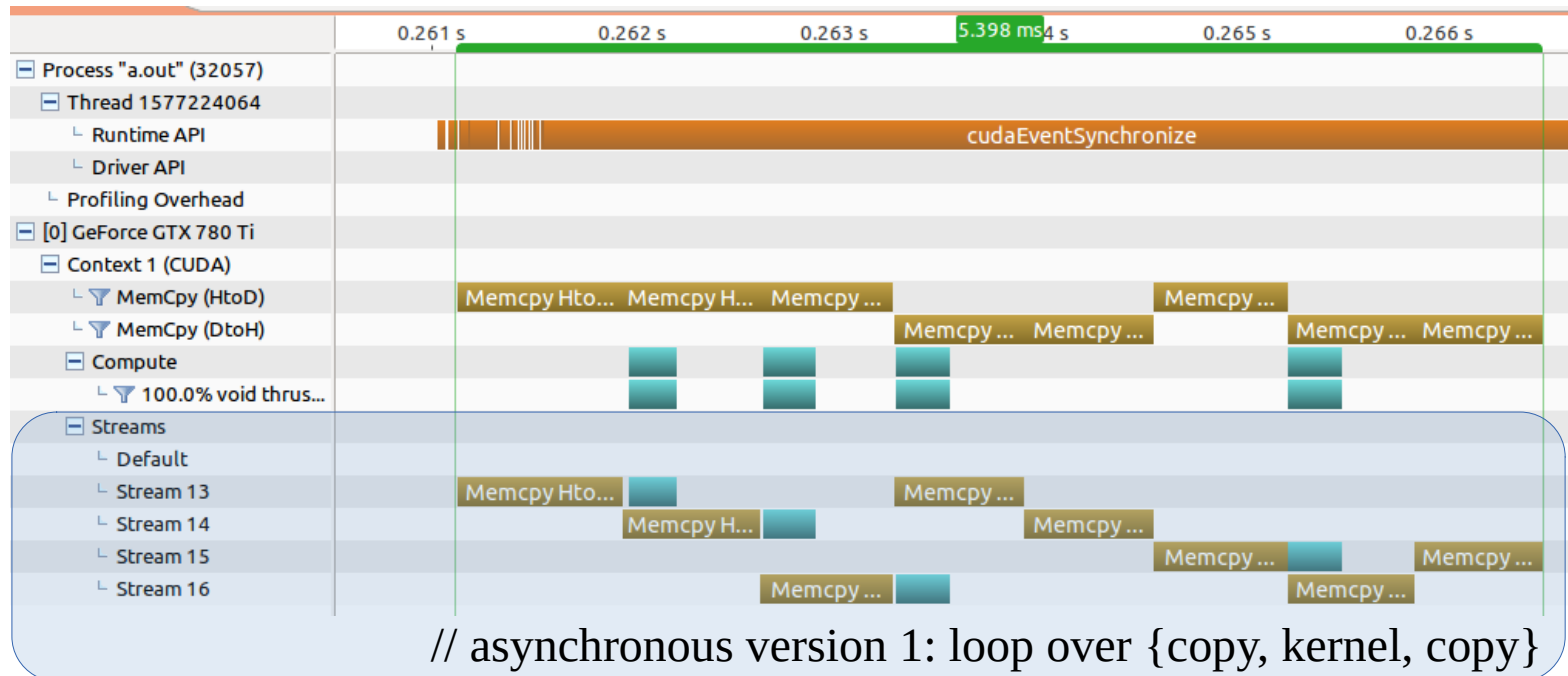
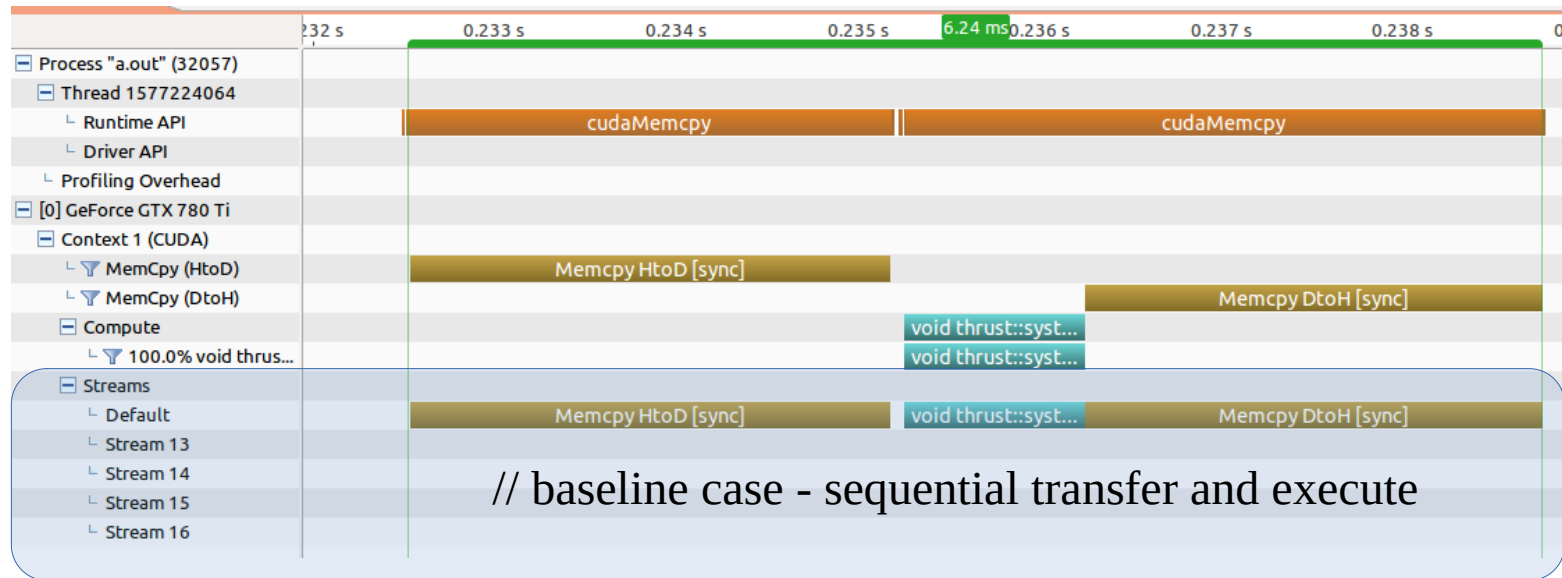
    #ifndef THRUST18
    kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);
    #else
    thrust::transform(
        thrust::cuda::par.on(stream[i]),
        thrust::make_counting_iterator(offset),
        thrust::make_counting_iterator(streamSize+offset),
        d_a+offset,
        d_a+offset,
        functor()
    );
    #endif
    checkCuda( cudaMemcpyAsync(&a[offset], &d_a[offset],
                                streamBytes, cudaMemcpyDeviceToHost,
                                stream[i]) );
}
```

NVIDIA Nsight Systems

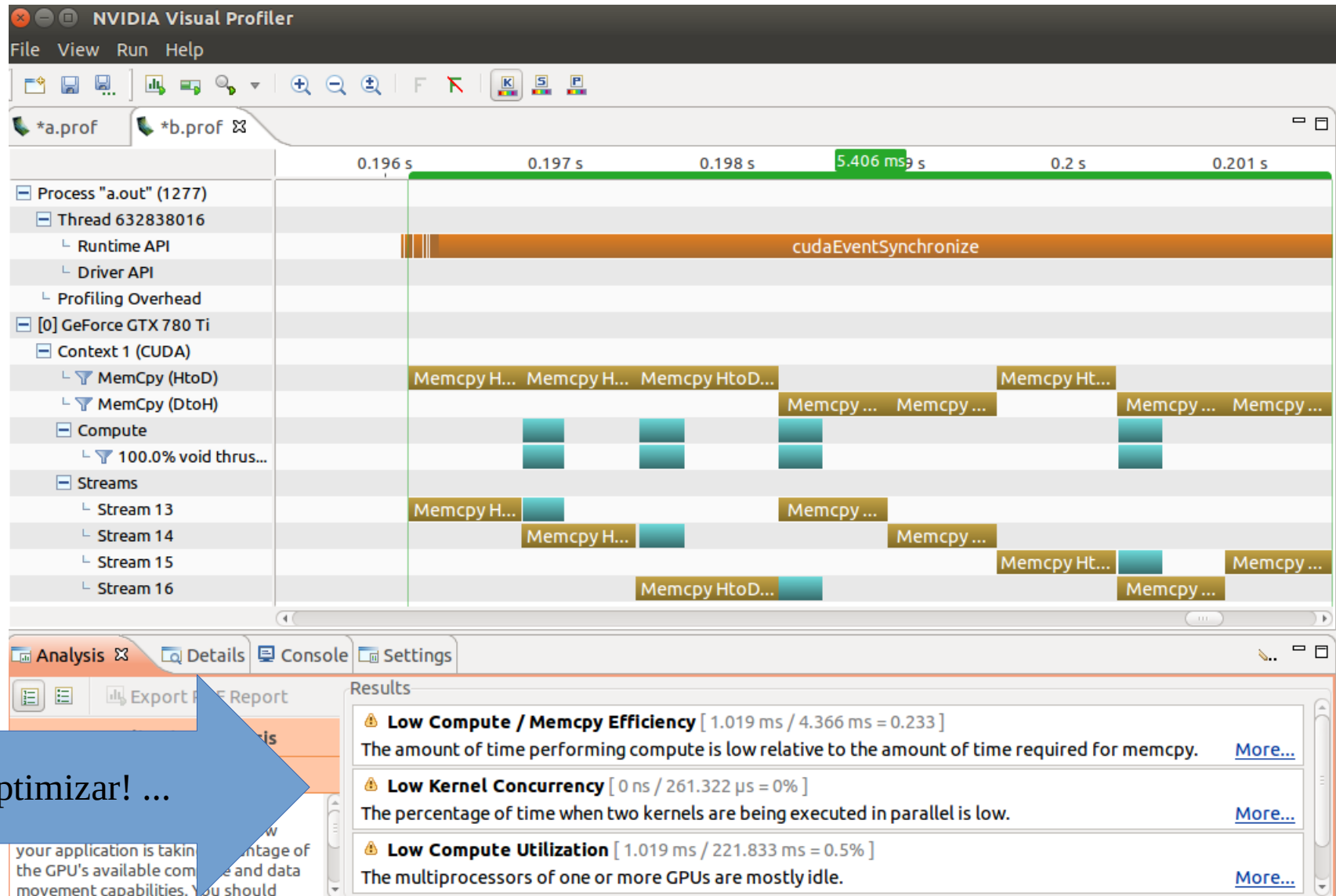
NVIDIA® Nsight™ Systems is a system-wide performance analysis tool designed to visualize and scale efficiently across any quantity or size of CPUs and GPUs; from large server to our smallest



Streams con Thrust



Streams con Thrust



Mas ejemplos de Streams

• *Hello World from stream...*

- thrust_streams_muy_simple.cu

Streams en librerias...

- cufft_streams.cu

*$y[i] = a * x[i] + y[i]$ (con $x[i] = i$, $y[i] = n - i$).*

- nvtxstreams.cu

ping pong entre dos streams

- ping_pong.cu

visual profiler!

- nvprof → nvvp → ejecutable.prof