Cách thực thi song song trong CUDA (phần 4)

Trần Trung Kiên
ttkien@fit.hcmus.edu.vn

Cập nhật 13/11/2021

KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

fit@hcmus

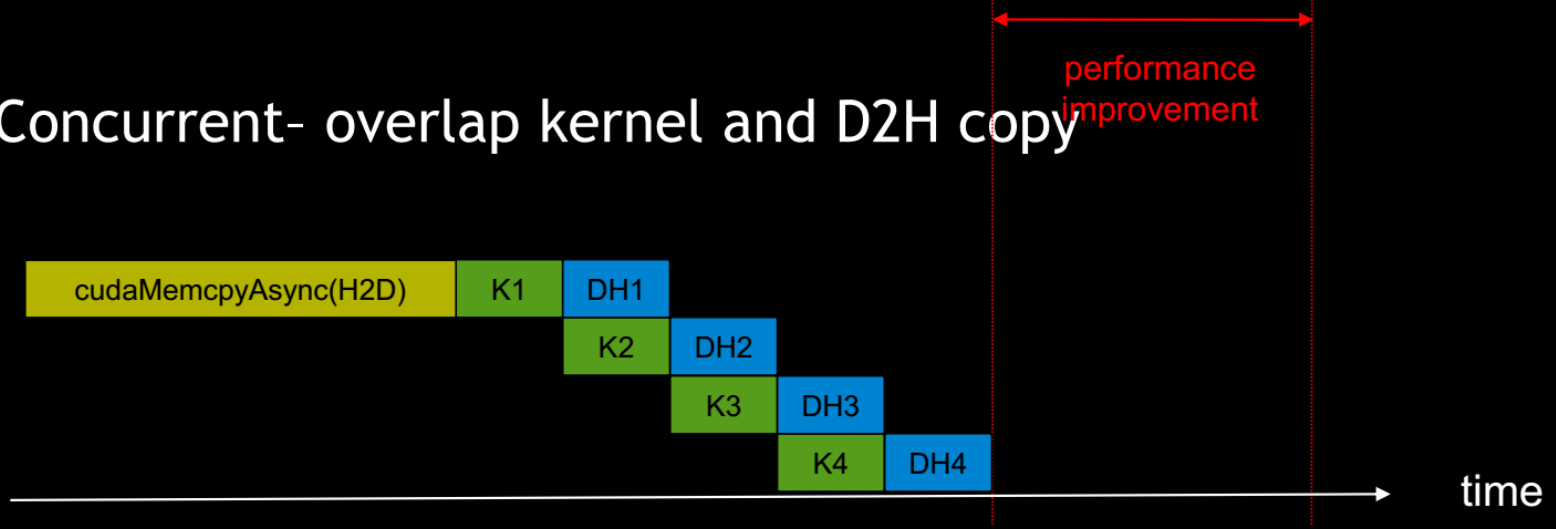
Dẫn nhập

- Tối ưu hóa: tận dụng tối đa các tài nguyên phần cứng, giữ cho chúng luôn "bận rộn", không để ai "ngồi chơi" một cách lãng phí
- Đến nay, mới chỉ nói về tối ưu hóa trong phạm vi một hàm kernel
  - Cần có đủ số block để tận dụng hết các SM
  - Trong mỗi SM, cần có đủ số câu lệnh độc lập (có thể đến từ cùng một warp hoặc đến từ các warp khác nhau) để tận dụng các execution pipeline, che độ trễ
  - Các thread trong cùng một warp nên làm cùng một việc (mỗi thread có dữ liệu riêng)
- Hôm nay, sẽ nói về tối ưu hóa ở phạm vi rộng hơn: ngoài một hàm kernel

CONCURRENCY THROUGH PIPELINING

- Serial

- Concurrent– overlap kernel and D2H copy

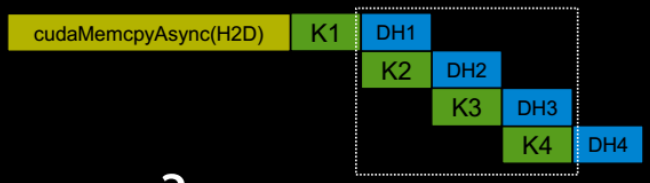


Nguồn: Justin Luitjens, CUDA Streams, GTC2014

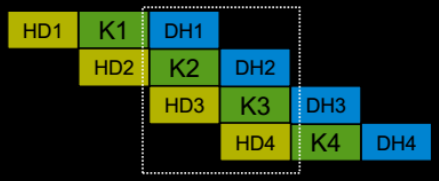
Càng "overlap" giữa các công việc → càng tận dụng các tài nguyên phần cứng

- Serial (1x)

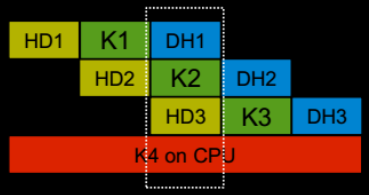
- 2-way concurrency (up to 2x)



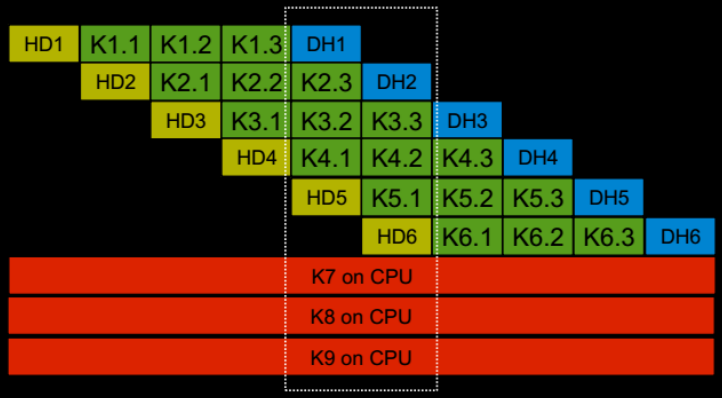
- 3-way concurrency (up to 3x)



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



- **4+ way concurrency**



Nguồn: Justin Luitjens, CUDA Streams, GTC2014

“Overlap” các công việc

- Với hầu hết các device hiện nay, ta có thể “overlap” tối đa như sau:

  nhiều tính toán trên host (nếu tận dụng các core của CPU)

  và nhiều tính toán trên device (nhiều hàm kernel)

  và một H2D (chép dữ liệu từ host sang device)

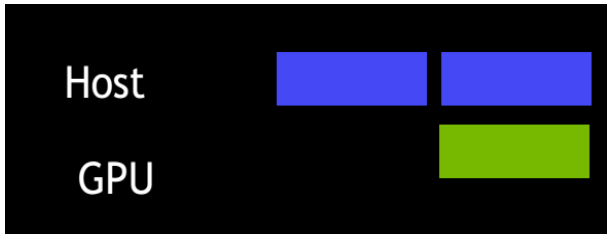
  và một D2H (chép dữ liệu từ device sang host)

- Điều kiện cơ bản để có thể “overlap” các công việc:
  - Các công việc này phải độc lập với nhau
  - Có đủ tài nguyên phần cứng cho các công việc

Trong CUDA, làm sao để "overlap" các công việc?

Khi host gọi các câu lệnh CUDA thì có thể xảy ra một trong hai trường hợp

- **Đồng bộ (synchronous)**: host đưa câu lệnh vào hàng đợi của device và chờ đến khi câu lệnh này được hoàn thành
  - Vd, `cudaMemcpy`
- **Không đồng bộ (asynchronous)**: host đưa câu lệnh vào hàng đợi của device và tiếp tục đi tiếp mà không chờ câu lệnh này hoàn thành
  - Vd, host gọi hàm kernel



Bình thường đã có thể "overlap" tính toán trên host và tính toán trên device
Làm sao để "overlap" các công việc khác (vd, kernel với kernel)?

Nguồn ảnh: Justin Luitjens, CUDA Streams, GTC2014

Trong CUDA, làm sao để "overlap" các công việc?

- Một luồng CUDA (CUDA stream) là một hàng đợi các công việc của device
  - Host đưa công việc vào hàng đợi của device
- Các công việc thuộc cùng một luồng sẽ được device thực thi một cách tuần tự theo thứ tự FIFO (do host đưa vào)
- Các công việc thuộc các luồng khác nhau sẽ không có thứ tự với nhau và có thể "overlap" với nhau

Các câu lệnh về luồng CUDA

- **Tạo luồng**

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

- **Hủy luồng**

```
cudaStreamDestroy(stream);
```

- **Đưa công việc vào luồng**
  -
```
kernel<<<gridSize, blockSize,
         0, // Tạm thời chưa đụng đến tham số này
         stream>>>(…);
```
  -
```
cudaMemcpyAsync(dst, src, size, dir,
                stream);
```

Luồng mặc định (luồng 0 / luồng NULL)

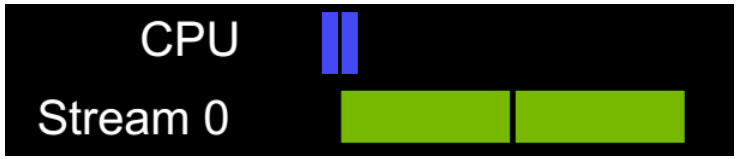
- Mặc định, các công việc sẽ được đưa vào luồng 0
- Lưu ý: luồng 0 đồng bộ hóa với các luồng khác
  - Vd, host đưa các công việc (CV) vào các luồng (L) theo thứ tự CV0-L0, CV-L1, CV-L2, CV1-L0 thì ở phía device:
    - Đầu tiên sẽ thực hiện CV0-L0
    - Xong CV0-L0 rồi mới thực hiện CV-L1, CV-L2 (CV-L1 và CV-L2 có thể "overlap")
    - Xong CV-L1 và CV-L2 rồi mới thực hiện CV1-L0
- Để có thể "overlap"
  - C1: thay luồng 0 thành luồng khác 0
  - C2: tạo các luồng khác 0 như sau

```
cudaStreamCreateWithFlags(&stream,
                          cudaStreamNonBlocking)
```

Ví dụ: “overlap” giữa các hàm kernel

- Giả sử hàm kernel `foo` chỉ dùng 50% tài nguyên device
- **Dùng luồng 0**

```
foo<<<blocks, threads>>>();
foo<<<blocks, threads>>>();
```



- **Dùng luồng 0 và luồng khác 0**

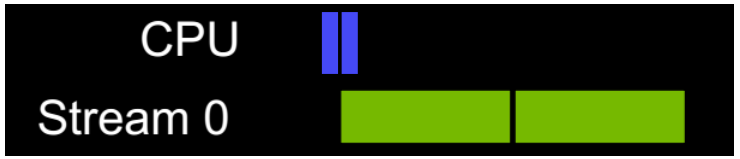
```
cudaStream_t stream1;
cudaStreamCreate(&stream1);
foo<<<blocks, threads>>>();
foo<<<blocks, threads,
        0, stream1>>>();
cudaStreamDestroy(stream1);
```

Nguồn ảnh: Justin Luitjens, CUDA Streams, GTC2014

Ví dụ: “overlap” giữa các hàm kernel

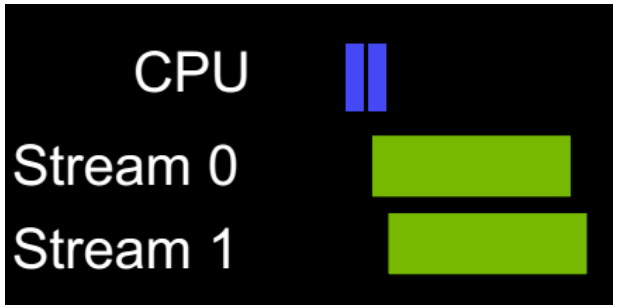
- Giả sử hàm kernel `foo` chỉ dùng 50% tài nguyên device
- **Dùng luồng 0**

```
foo<<<blocks, threads>>>();
foo<<<blocks, threads>>>();
```



- **Dùng luồng 0 và luồng khác 0**

```
cudaStream_t stream1;
cudaStreamCreateWithFlags(
   &stream1, cudaStreamNonBlocking);
foo<<<blocks, threads>>>();
foo<<<blocks, threads,
        0, stream1>>>();
cudaStreamDestroy(stream1);
```

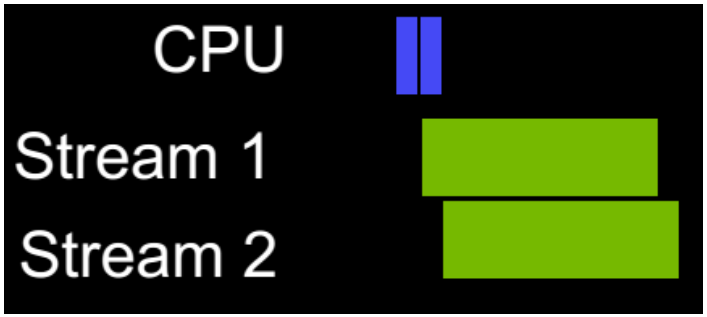


Nguồn ảnh: Justin Luitjens, CUDA Streams, GTC2014

Ví dụ: “overlap” giữa các hàm kernel

- Giả sử hàm kernel `foo` chỉ dùng 50% tài nguyên device
- **Dùng các luồng khác 0**

```
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
foo<<<blocks, threads,
       0, stream1>>>();
foo<<<blocks, threads,
       0, stream2>>>();
cudaStreamDestroy(stream1);
cudaStreamDestroy(stream2);
```



Nguồn ảnh: Justin Luitjens, CUDA Streams, GTC2014

Ví dụ: "overlap" việc chép dữ liệu với các công việc khác

- **Ví dụ 1**

```
cudaMemcpy(...);
foo<<<...>>>();
```

- **Ví dụ 2**

```
cudaMemcpyAsync(..., stream1);
foo<<<..., stream1>>>();
```

Cần "**pin**" vùng nhớ của host: thay `malloc` bằng `cudaMallocHost` (và `free` bằng `cudaFreeHost`)
Tại sao? Để host có thể tiếp tục làm việc của mình và để phần cứng của device có thể tự chép dữ liệu thì vùng nhớ vật lý chứa dữ liệu ở host phải được giữ nguyên - phải được "pin"; nếu không được "pin" thì dữ liệu ở vùng nhớ vật lý của host có thể bị OS thay đổi trong quá trình device chép dữ liệu ☹ (do cơ chế bộ nhớ ảo của host)

Nguồn ảnh: Justin Luitjens, CUDA Streams, GTC2014

Ví dụ: "overlap" việc chép dữ liệu với các công việc khác

- **Ví dụ 1**

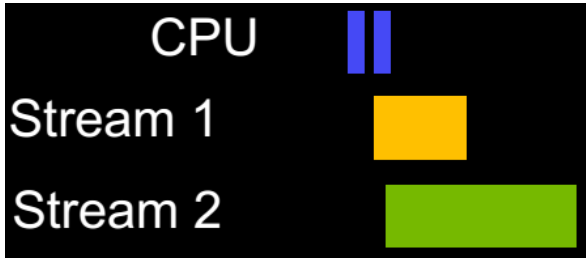
```
cudaMemcpy(...);
foo<<<...>>>();
```

- **Ví dụ 2**

```
cudaMemcpyAsync(..., stream1);
foo<<..., stream1>>>();
```

- **Ví dụ 3**

```
cudaMemcpyAsync(..., stream1);
foo<<..., stream2>>>();
```



Nguồn ảnh: Justin Luitjens, CUDA Streams, GTC2014

Đồng bộ hóa

Khi “thả” cho các công việc chạy bất đồng bộ, ta sẽ có nhu cầu đồng bộ hóa vào một lúc nào đó

Đồng bộ hóa

Đồng bộ hóa host với device

```
cudaDeviceSynchronize();
```

Đồng bộ hóa host với một luồng

- `cudaStreamSynchronize(stream);`
- `cudaStreamQuery(stream);`
  - Không bắt host phải chờ
  - Trả về: `cudaSuccess` nếu các công việc của luồng `stream` đã hoàn thành; `cudaErrorNotReady` nếu ngược lại

Đồng bộ hóa

Đồng bộ hóa host với một điểm trong luồng: dùng "event"

- Tạo event

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

- Đưa event vào luồng

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

- Đồng bộ hóa host với event
  - `cudaEventSynchronize(event);`
  - `cudaEventQuery(event);//Tương tự cudaStreamQuery`
- Hủy event

```
cudaEventDestroy(event);
```

Đồng bộ hóa

Đồng bộ hóa các luồng với nhau

```
cudaStreamWaitEvent(stream, event)
```

- Luồng `stream` sẽ chờ `event` (của một luồng khác) xảy ra rồi mới thực hiện tiếp các công việc được đưa vào luồng `stream` sau câu lệnh này
- Host không phải chờ!

Ví dụ

```
// Dummy computation, just to keep the kernel run long enough
__global__ void kernel1()
{
   double sum = 0.0;

   for(int i = 0; i < 300000; i++)
   {
      sum = sum + tan(0.1) * tan(0.1);
   }
}

// kernel2, kernel3, kernel 4: identical to kernel1
__global__ void kernel2() { ... }
__global__ void kernel3() { ... }
__global__ void kernel4() { ... }
```

```
// Create an array of non-null streams
int nStreams = 4;
cudaStream_t *streams = (cudaStream_t *) malloc(nStreams * sizeof(cudaStream_t));
for (int i = 0; i < nStreams; i++)
   cudaStreamCreate(&streams[i]);
// Create events
cudaEvent_t start, stop;
cudaEventCreate(&start); cudaEventCreate(&stop);
// Send start event to null stream
cudaEventRecord(start, 0);
// Send jobs to non-null streams in depth-first order
...
// Send stop event to null stream and wait for it
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
// Compute elapsed time
float time;
cudaEventElapsedTime(&time, start, stop);
// Destroy events
cudaEventDestroy(start); cudaEventDestroy(stop);
// Destroy streams
for (int i = 0; i < nStreams; i++)
   cudaStreamDestroy(streams[i]);
free(streams);
```

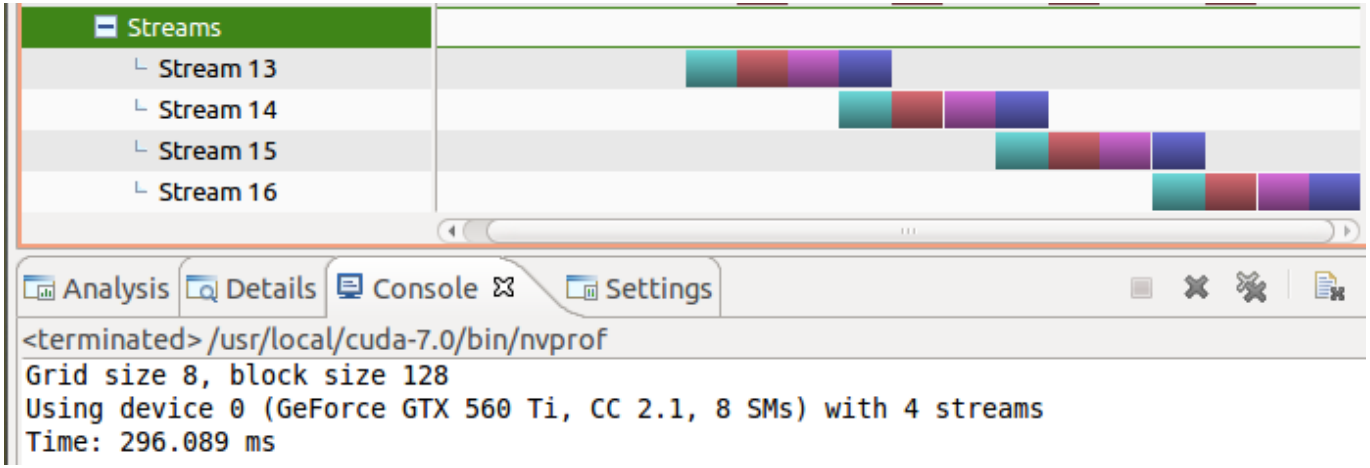
Thời gian (ms) giữa start (start được lấy ra khỏi luồng null) và stop (stop được lấy ra khỏi luồng null)

```
// Create an array of non-null streams
int nStreams = 4;
cudaStream_t *streams = (cudaStream_t *) malloc(nStreams * sizeof(cudaStream_t));
for (int i = 0; i < nStreams; i++)
   cudaStreamCreate(&streams[i]);
// Create events
…
// Send start event to null stream
…
// Send jobs to non-null streams in depth-first order
for (int i = 0; i < nStreams; i++)
{
   kernel1<<< 8, 128, 0, streams[i]>>>();
   kernel2<<< 8, 128, 0, streams[i]>>>();
   kernel3<<< 8, 128, 0, streams[i]>>>();
   kernel4<<< 8, 128, 0, streams[i]>>>();
}
// Send stop event to null stream and wait for it
…
// Compute elapsed time
…
// Destroy streams
…
// Destroy events
…
```

- Trong một streams[i]: kernel1, kernel2, kernel3, kernel4 có “overlap”?
- Các streams[i] khác nhau có “overlap”?

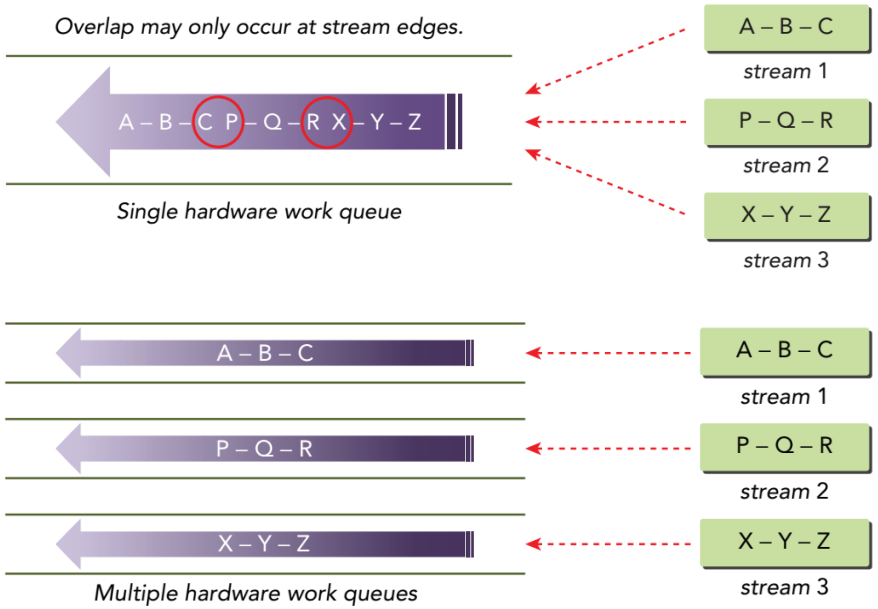
Vấn đề "false dependency" giữa các luồng của device có CC < 3.5

Kết quả chạy trên device CC 2.x với profiler của NVIDIA:



Nguyên nhân: device có CC < 3.5 chỉ có một queue vật lý

Device có CC >= 3.5 giải quyết vấn đề này bằng cách dùng **Hyper-Q**: nhiều queue vật lý



Nguồn ảnh: John Cheng et al. Professional CUDA C Programming. 2014

Giải quyết vấn đề “false dependency” của device có CC < 3.5

```
// Send jobs to non-null streams in breadth-first order
for (int i = 0; i < nStreams; i++)
  kernel1<<< 8, 128, 0, streams[i]>>>();
for (int i = 0; i < nStreams; i++)
  kernel2<<< 8, 128, 0, streams[i]>>>();
for (int i = 0; i < nStreams; i++)
  kernel3<<< 8, 128, 0, streams[i]>>>();
for (int i = 0; i < nStreams; i++)
  kernel4<<< 8, 128, 0, streams[i]>>>();
```

Kết quả chạy trên device CC 2.x với profiler của NVIDIA: