

Lập trình song song trên GPU

HW3: Các loại bộ nhớ trong CUDA

Nên nhớ mục tiêu chính ở đây là **học, học một cách chân thật**. Bạn có thể thảo luận ý tưởng với bạn khác, nhưng **bài làm phải là của bạn, dựa trên sự hiểu thật sự của bạn**. **Nếu vi phạm thì sẽ bị 0 điểm cho toàn bộ môn học.**

Trong môn học, để thống nhất, tất cả các bạn (cho dù máy bạn có GPU) đều phải dùng Google Colab để biên dịch và chạy code (khi chấm Thầy cũng sẽ dùng Colab để chấm). Với mỗi bài tập, bạn thường sẽ phải nộp:

- 1) **File code** (file .cu)
- 2) **File báo cáo** là file notebook (file .ipynb) của Colab (nếu bạn nào biết Jupyter Notebook thì bạn thấy Jupyter Notebook và Colab khá tương tự nhau, nhưng 2 cái này hiện chưa tương thích 100% với nhau: file .ipynb viết bằng Jupyter Notebook có thể sẽ bị mất một số cell khi mở bằng Colab và ngược lại). File này sẽ chứa các kết quả chạy. Ngoài ra, một số bài tập có phần viết (ví dụ, yêu cầu bạn nhận xét về kết quả chạy), và bạn sẽ viết trong file notebook của Colab luôn. Colab có 2 loại cell: **code cell** và **text cell**. Ở code cell, bạn có thể chạy các câu lệnh giống như trên terminal của Linux bằng cách thêm dấu ! ở đầu. Ở text cell, bạn có thể soạn thảo văn bản theo cú pháp của Markdown (rất dễ học, bạn có thể xem [ở đây](#)); như vậy, bạn sẽ dùng text cell để làm phần viết trong các bài tập. Bạn có thể xem về cách thêm code/text cell và các thao tác cơ bản [ở đây](#), mục “Cells” (đừng đi qua mục “Working with Python”). Một phím tắt ưa thích của mình khi làm với Colab là ctrl+shift+p để có thể search các câu lệnh của Colab (nếu câu lệnh có phím tắt thì bên cạnh kết quả search sẽ có phím tắt). File notebook trên Colab sẽ được lưu vào Google Drive của bạn; bạn cũng có thể download trực tiếp xuống bằng cách ấn ctrl+shift+p, rồi gõ “download .ipynb”.

Đề bài

Câu 1 (7đ)

Áp dụng hiểu biết về các loại bộ nhớ trong CUDA để tối ưu hóa chương trình làm mờ ảnh RGB (cách làm mờ giống như ở HW1).

Các bạn có thể tham khảo file Demo/05-MemoryDemo.cu trong Drive để xem cách xử lý với Convolution1D, từ đó áp dụng lên Convolution2D trong bài này.

Code (5 đ)

Mình có đính kèm file ảnh đầu vào [in.pnm](#) (trong Windows, bạn có thể xem file *.pnm bằng chương trình [IrfanView](#)). Mình cũng đã viết sẵn khung chương trình trong file [HW3_P1.cu](#) đính kèm. Bạn sẽ cần phải viết code ở những chỗ mình để [// TODO](#):

- Định nghĩa hàm kernel [blurImgKernel1](#) – hàm kernel làm mờ ảnh ở HW1. Bạn nào đã làm HW1 rồi thì chỉ cần copy-paste (code lại một lần nữa cũng tốt); bạn nào chưa làm HW1 thì đầu tiên bạn phải hoàn

thành hàm kernel này. Đây là hàm kernel cơ bản nhất; nếu bạn chưa cài đặt được hàm kernel này thì không thể qua hàm kernel 2 và 3 (sẽ được trình bày ở dưới) được.

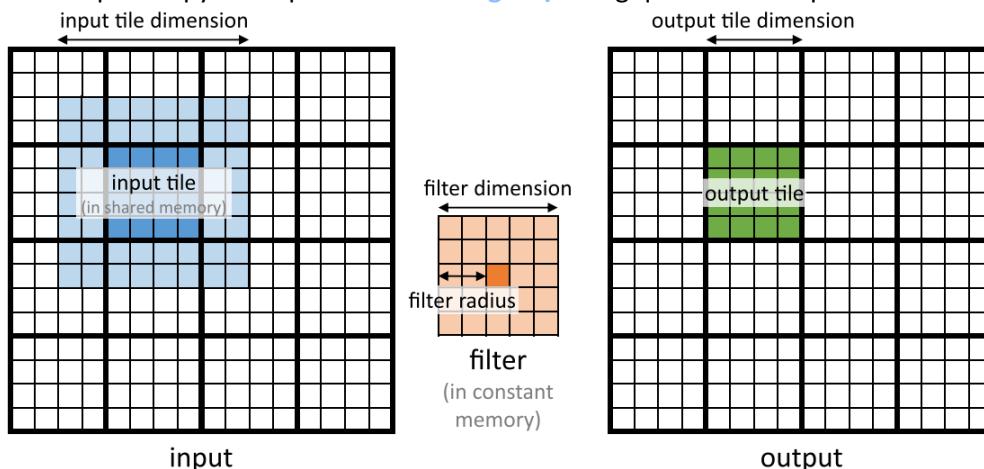
- Gọi hàm `blurImgKernel1`.
- Định nghĩa hàm kernel `blurImgKernel2` – hàm kernel làm mờ ảnh **có sử dụng SMEM**. Mỗi block sẽ đọc phần dữ liệu của mình từ `inPixels` ở GMEM vào SMEM, sau đó phần dữ liệu ở SMEM này sẽ được **dùng lại nhiều lần** cho các thread trong block. Bạn sẽ **cấp phát động** một mảng ở SMEM của mỗi block để cho phép kích thước của mảng này có thể thay đổi theo `filterWidth` và `blockSize`:
 - Trong hàm kernel `blurImgKernel2`, bạn khai báo mảng `s_inPixels` ở SMEM như sau:
`extern __shared__ uchar3 s_inPixels[];`
 - Khi gọi hàm kernel `blurImgKernel2`, trong cặp `<<...>>`, ngoài tham số `gridSize` và `blockSize`, bạn sẽ truyền vào **tham số thứ ba** cho biết kích thước (byte) của mảng `s_inPixels` trong SMEM của mỗi block (kích thước này được tính theo biến `filterWidth` và biến `blockSize`, ở đây ta cũng lưu mảng 2 chiều dưới dạng mảng 1 chiều).
- Gọi hàm `blurImgKernel2`.

Lưu ý:

Phần dữ liệu cần copy từ `inPixels` sang **SMEM** sẽ lớn hơn kích thước block, do phải copy thêm các phần tử xung quanh input để có thể thực hiện convolution tại các vùng gần biên.

Hình bên dưới thể hiện ý tưởng:

- Block output màu **xanh lá**
- Block Input màu xanh **dương đậm**.
- Cần phải copy thêm phần xanh **dương nhạt** xung quanh block input.



- Định nghĩa hàm kernel `blurImgKernel3` – hàm kernel làm mờ ảnh có sử dụng SMEM cho `inPixels` và **sử dụng CMEM cho filter**. Ở đầu file code, mình đã khai báo cho bạn mảng `dc_filter` ở CMEM. Do `dc_filter` ở tầm vực toàn cục nên trong hàm kernel `blurImgKernel3` bạn có thể sử dụng được `dc_filter` (do đó, không cần tham số đầu vào ứng với filter nữa). Nếu bạn đã viết xong hàm kernel `blurImgKernel2` thì bạn chỉ cần copy-paste và sửa `filter` thành `dc_filter`.
- Copy dữ liệu từ `filter` ở host sang `dc_filter` ở CMEM (dùng hàm `cudaMemcpyToSymbol`).
- Gọi hàm `blurImgKernel3`.

Hướng dẫn về các câu lệnh:

(Các câu lệnh dưới đây là chạy trên terminal của Linux, khi chạy ở code cell của Colab thì bạn thêm dấu ! ở đầu)

- Biên dịch file [HW3.cu](#): `nvcc HW3.cu -o HW3`
- Chạy file [HW3](#) với file ảnh đầu vào là [in.pnm](#), file ảnh đầu ra là [out.pnm](#):
`./HW3 in.pnm out.pnm`

Lúc này, chương trình sẽ: đọc file ảnh [in.pnm](#); làm mờ ảnh bằng host (để có kết quả đúng làm chuẩn), và làm mờ ảnh bằng device với 3 phiên bản của hàm kernel: [blurlImgKernel1](#), [blurlImgKernel2](#), và [blurlImgKernel3](#); các kết quả sẽ được ghi xuống 4 file: [out_host.pnm](#), [out_device1.pnm](#), [out_device2.pnm](#), và [out_device3.pnm](#). Với mỗi hàm kernel, chương trình sẽ in ra màn hình thời gian chạy và sự sai biệt so với kết quả của host (mình chạy thì thấy kết quả của device có sự sai biệt **nhỏ** so với kết quả của host, khoảng 0.000x; đó là do GPU tính toán số thực có thể hơi khác so với CPU, chứ không phải là do cài đặt sai).

Mặc định thì chương trình sẽ dùng block có kích thước 32×32 ; nếu bạn muốn chỉ định kích thước block thì truyền thêm vào câu lệnh 2 con số lần lượt ứng với kích thước theo chiều x và theo chiều y của block (ví dụ, [./HW3 in.pnm out.pnm 32 16](#)).

[Báo cáo \(2đ\)](#)

Trong file “[HW3.ipynb](#)” mà mình đính kèm:

- Bạn biên dịch và chạy file “[HW3.cu](#)”
- Giải thích tại sao kết quả lại như vậy (tại sao dùng SMEM lại chạy nhanh/chậm hơn so với không dùng, tại sao dùng CMEM lại chạy nhanh/chậm hơn so với không dùng). Nếu cần thì bạn có thể thực hiện thêm các thí nghiệm để kiểm chứng cho lý giải của bạn. Chỗ nào mà bạn không biết tại sao thì cứ nói là không biết tại sao.
- **Thử nghiệm với một số ảnh đầu vào đặc biệt và filter đặc biệt** (SV có thể sửa code một số chỗ khác, ngoài #TODO, để làm các thử nghiệm này)

[Câu 2 \(3đ\)](#)

Apply CUDA streams to optimize the program performing operations on vectors

Problem Description

You have **8 vectors**, each with 4 million elements. Each vector must undergo three sequential operations:

1. **Scale**: Multiply each element by 2.0
2. **Add**: Add 10.0 to each element
3. **Square**: Square each element

Sequential approach: Process one vector completely (all three operations), then move to the next vector.

Stream approach: Use CUDA streams to process multiple vectors concurrently, overlapping memory transfers and computation.

Part A: Implementation Tasks (2pts)

Task 1: Complete processSequential()

What to do:

1. Allocate device memory for one vector
2. For each vector:
 - o Copy input to device using cudaMemcpy
 - o Launch all three kernels in sequence
 - o Copy result back to host
 - o Use cudaDeviceSynchronize() after each vector

Task 2: Complete processBreadthFirst()

What to do:

1. Create an array of streams
2. Allocate device memory for each stream
3. Implement breadth-first pattern:
 - o **Phase 1:** All H2D transfers
 - o **Phase 2:** All scale kernels
 - o **Phase 3:** All add kernels
 - o **Phase 4:** All square kernels
 - o **Phase 5:** All D2H transfers
4. Synchronize and cleanup

Task 3: Complete processDepthFirst()

What to do:

1. Create streams and allocate memory (same as breadth-first)
2. Implement depth-first pattern:
 - o For each vector, issue ALL operations in its stream
 - o H2D → Scale → Add → Square → D2H
3. Synchronize and cleanup

Part B: Analysis Questions (1pts)

After completing the implementation, students should answer in notebook:

Question 1: Performance Analysis

Why is depth-first faster than breadth-first?

Question 2: Scalability

- What happens if you increase the number of vectors to 16? 32?
- Students should experiment and plot results

Question 3: Memory Transfer

How much time is spent on memory transfers vs computation?

Student can modify the code to get these time duration.

For example: 30-40% transfer, 60-70% compute. Streams help overlap these components

Question 4: GPU Utilization

How can you verify the GPU is actually doing concurrent work?

You also need to capture the results from NVIDIA Nsight Systems to show that using n streams results in overlap between tasks, while using 1 stream has no overlap. For example

1. **Single Stream Case:** Timeline showing sequential execution where:

- Memory transfers (H2D, D2H) and kernel executions happen one after another
- No overlap/parallelism visible
- Total execution time = sum of all individual operations

2. **Multiple Streams Case:** Timeline showing concurrent execution where:

- Multiple kernels executing simultaneously on the GPU
- Memory transfers overlapping with kernel execution (if using pinned memory)
- Operations from different streams running in parallel
- Total execution time < sum of all operations due to overlap

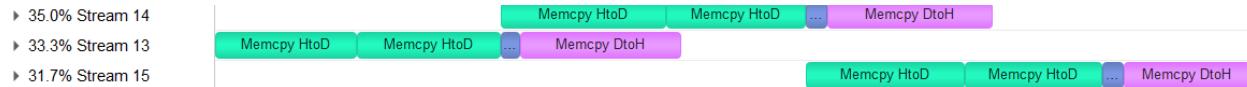
The visual timeline from Nsight Systems will clearly demonstrate the performance benefit of stream-based parallelism versus sequential execution.

Example

With 1 stream:



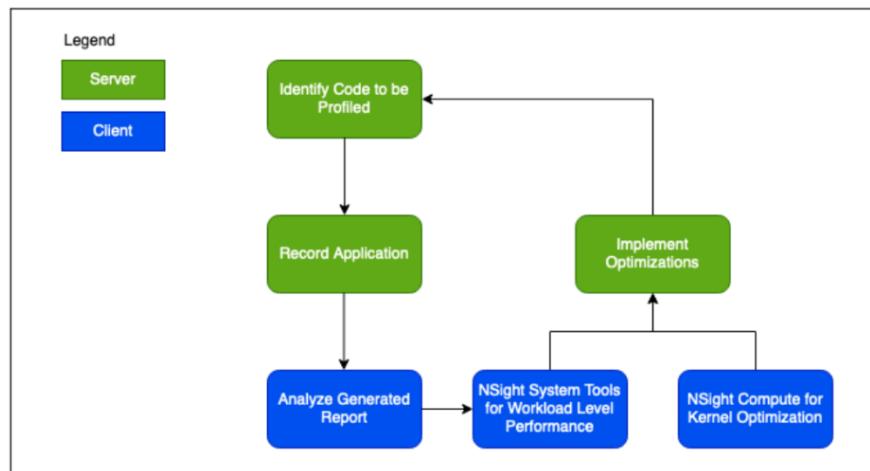
With 3 stream:



Guide to Using NVIDIA Nsight Systems:

NVIDIA provides us with tools like Nsight Systems and Nsight Compute. To learn more information about these tools, you can refer here:

<https://www2.cisl.ucar.edu/events/gpu-series-hands-session-nsight-systems-and-compute>



The image above describes the profiling process (viewing detailed execution information to optimize code). It consists of 2 parts:

- Server: where CUDA code runs and generates the report. Example: Google Colab.
- Client: where the report is viewed. Example: your personal computer.

Both need to have **Nsight Systems** installed. .

Installing Nsight Systems on Google Colab

- In the HW3.ipynb file, the installation method is already written. You just need to run the corresponding cell.
- After installation is complete, generate a report by entering the following command: `nsys profile ./a.out` (a.out is the executable file, previously compiled with nvcc, you can add command-line parameters after a.out if needed).
- The report file will be generated after the above command. Download it to your machine.

· Installing Nsight Systems on your personal computer:

- Install through NVIDIA's official website
- After installation is complete, the simplest way to open the report file on Windows is to double-click on this file

- For Nsight Systems on your personal computer to be able to open the report file created by Nsight Systems on Colab, the version of Nsight Systems on your personal computer must be equal to or higher than the version of Nsight Systems on Colab. You can check the version of Nsight Systems on Colab using the command `nsys --version`

When the report file is opened in Nsight Systems, you may need to zoom in a bit to see what you need to see (CTRL + Scroll, Select section -> Zoom into selection).

Submitting

You should organize your submission folder as follows:

Folder <Student_ID> (e.g., if you have Student ID 1234567, then name the folder 1234567)

- Code file "HW3_P1.cu"
- Code file "HW3_P2.cu"
- Report file "HW3.ipynb"

Then, compress this <Student_ID> folder and submit it at the link on Moodle.