CUDA Streams

UNIVERSITY OF CALIFORNIA
UC RIVERSIDE

Slide credit: Slides adapted from

Midterm Question 1

You are the head architect for a new open source GPU project. In your first design meeting, layout how YOU believe the architecture should be designed. As it is an open sourced project programmability and ease of use are important considerations. Explain why you designed it in that way. Defend your design with any reasoning you feel is valid along with a use case

- Goal is to understand the connection between why GPUs are designed the way they are and the motivation behind them
- People gave motivations such as it need to be; data parallel, throughput orientated, easy to program, thousands of threads, etc..
- But did not provide how the architecture satisfies those requirements
- Some gave hardware design; alu -> simd -> sm, reg files, memory system, etc…
- But did not provide any reasoning for why they decided to design in this way
- Answer needed to link the two together with solid reasoning

Midterm Question 2

A member of your group suggests a Nvidia style GPU. Cost is a concern, so the total number of SIMD units is limited 32. You are presented with three options, 32 SMs with 1 SIMD unit each, 16 SMs with 2 SIMD units, or 8 SMs with 4 SIMD units. Evaluate each option, giving pros and cons for each. Justify your decision with any reasoning you feel is valid.

- All option have the same theoretical computation because, they all have a total of 32 SIMD units
- So one option is not necessarily faster than any other one
- The difference comes from how you program it
- Some gave arguments for more SIMD per SM to utilize shared memory more, better tiling perhaps
- Less SIMD units forces smaller thread block, so synchronizing within a thread block is less overhead
- It really depends on how you think the hardware will be use, some use cases fit better on other hardware

Midterm Question 3

Later in the project, someone suggests integrating a couple of CPU cores within the same chip as your GPU, instead of the typical connection over PCIe. Do you think this is a good idea? How would this affect programmability? Or the design of the GPU cores? What are the drawbacks? Justify your decision with any reasoning you feel is valid

- A lot of confusion that an integrated chip would be programmed differently than a discrete system
- This is not the case
- Complexity of a system doesn't necessarily mean more complexity to program
- Main difference is that there is no pcie to connect so CPU and GPU share memory systems; memory and caches.
- No need to copy any data
- The GPU cores can be the same, but some drawback are reduced space for GPU cores

Midterm Question 4

You and your buddy have developed a GPU program for an imaginary GPU, the G1000. The G1000 has 16 SMs with a maximum of 1024 threads/SM. You developed your program to have a block size of 1024 and a grid size of 16 to fully utilize the G1000. The program is work efficient, but each thread does a significant amount of work. The day after you finish coding, a new GPU comes, the G2000, with 32 SM and a maximum of 2048 threads/SM. Your friend suggests buying the G2000 to speed up your new program but realizes that changes to your program will be needed. They suggest all you need to do is half the block size and double the grid size, then the G2000 would be fully utilized. Do you agree or disagree with this modification? Why or why not? Show by example. Whether or not the G2000 is fully utilized would you expect any speedup in you program? Give any reason you feel is valid.

- Almost everyone got this one
- G2000 would not be fully utilized since each thread block will have 512 threads which underutilize the 2048 threads/ SM
- So you should not expect much speedup do to the same number of threads being used
- Or maybe some speedup because we might have more parallelism with more SMs

Midterm Question 5

After some debate, your friend then asks how you would modify your program to fully utilize the G2000? Would those changes affect the amount of work done per thread? If so how? Justify your modification with any reasoning you feel is valid.

- Almost everyone got this one
- To fully utilize the hardware you need to have a grid size of 32 and a block size of 2048
- If you do this you would need to modify the program so threads do ¼ of the work

Midterm Question 6

The debate ends when you both realize you do not have any money to buy the new card. Using the G1000, your program was only designed o run with a fixed data size and breaks when using a larger dataset. Your friend proposes two options, scale the grid size to fit the dataset or tile the algorithm. Which do you choose? Give the pros and cons for both

- Scaling grid size is easy enough to do and it works well, however performance won't scale if we are already fully utilizing the gpu
- Tiling requires more changes, but it could potentially increase performance if shared memory is used or other localities are taken advantage of

Midterm Question 7

Game of Life kernel

- Where __syncthreads() was placed was the issue
- Synchronization cannot be placed inside if else statements because it might be possible that not all threads in the block go down that path so this would stall the entire program
- ALSO, the iteration loop should be on the outside of the kernel. This is because ALL cells in the grid must be updated before you move onto the next iteration. This requires a device level synchronization to synchronize all thread blocks. This can only be done with cudaDeviceSynchronzie at the host level.
- I gave the most leeway with these next two questions since it may be confusing and it was the end of a really long test

Midterm Question 8

Game of Life kernel

- Some improvements included reducing the number of if else cases so there is less control flow
- Using shared memory since cells are used across many threads
- Any other improvements that you provided I accepted, but I did not accept just stating that we could improve memory coalescing, etc…
- I'm looking for the actual changes that will improve those behaviors

Midterm Overview

- Average was around 80%
- I'm sorry for the length of the test and I will fix that for the final, but expect the same style of questions
- I think y'all did an excellent job and you should be proud of yourselves
- I hope this shows that you are capable of understanding the material and you should have confidence in that
- Points were taken off either if no reasoning was provided, reasoning was unrelated to the question, or if your reasoning was flawed or incorrect
- Points were NOT taken off for choice one implementation over another, if you gave solid reasoning than you got the question correct
- I am really looking for why you believe in the answer that you provided
- If you have any questions on feedback or grade, please email me and we can work something out

CUDA Streams

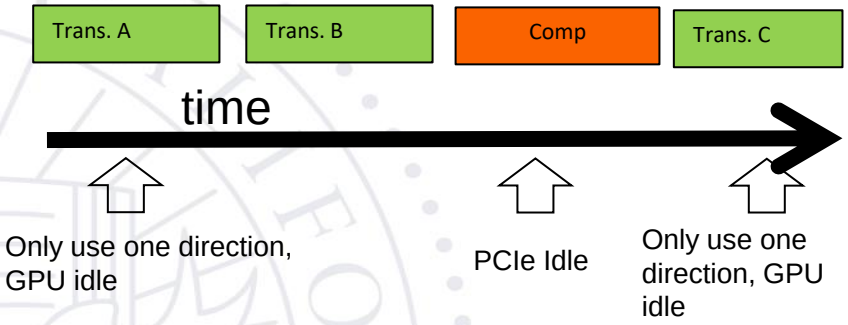
UNIVERSITY OF CALIFORNIA
UC RIVERSIDE

Slide credit: Slides adapted from

Serialized Data Transfer and Computation

UC RIVERSIDE UNIVERSITY OF CALIFORNIA

– So far, the way we use `cudaMemcpy` serializes data transfer and GPU computation for `VecAddKernel()`



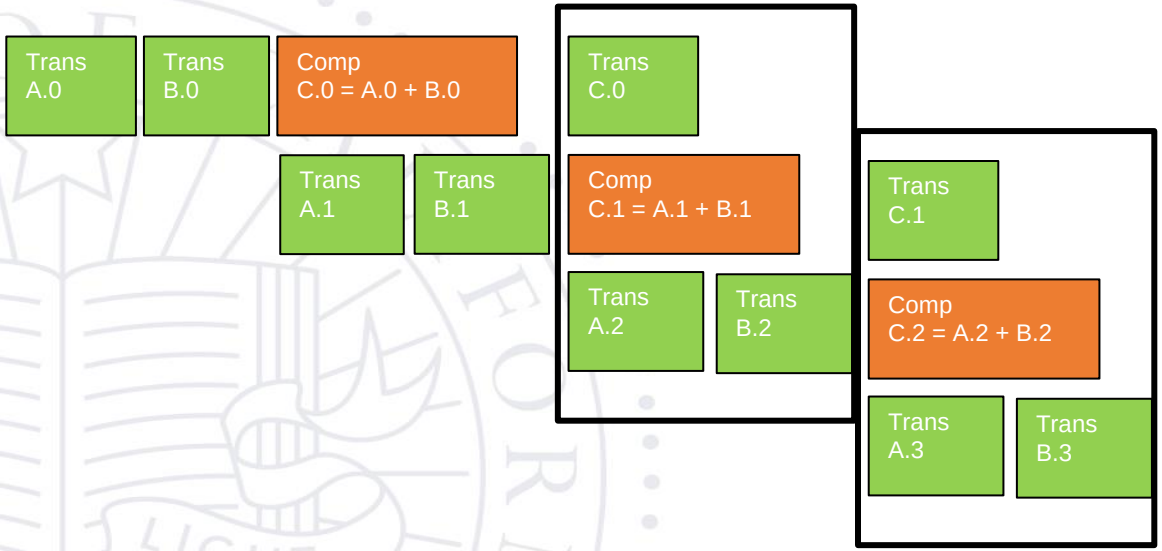
Device Overlap

- Some CUDA devices support device overlap
  - Simultaneously execute a kernel while copying data between device and host memory

```
int dev_count;
cudaDeviceProp prop;

cudaGetDeviceCount( &dev_count);
for (int i = 0; i < dev_count; i++) {
  cudaGetDeviceProperties(&prop, i);
  if (prop.deviceOverlap) …
```

Ideal, Pipelined Timing

- Divide large vectors into segments
- Overlap transfer and compute of adjacent segments

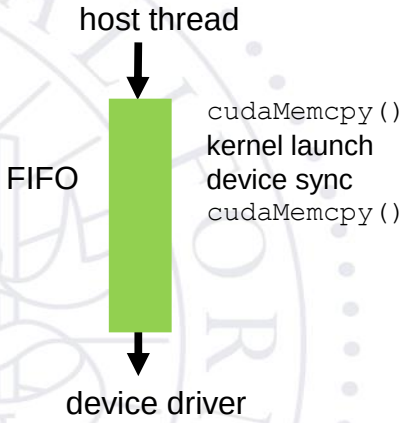


CUDA Streams

- CUDA supports parallel execution of kernels and `cudaMemcpy()` with “Streams”
- Each stream is a queue of operations (kernel launches and `cudaMemcpy()` calls)
- Operations (tasks) in different streams can go in parallel
  - “Task parallelism”

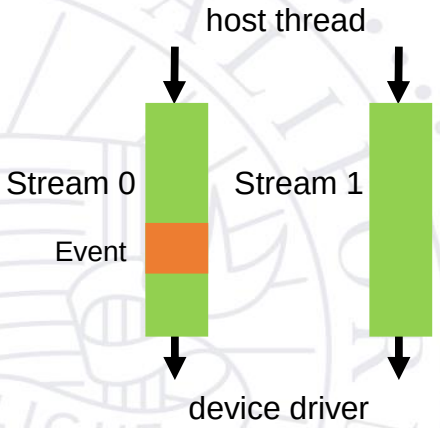
Streams

- Requests made from the host code are put into First-In-First-Out queues
  - Queues are read and processed asynchronously by the driver and device
  - Driver ensures that commands in a queue are processed in sequence. E.g., Memory copies end before kernel launch, etc.

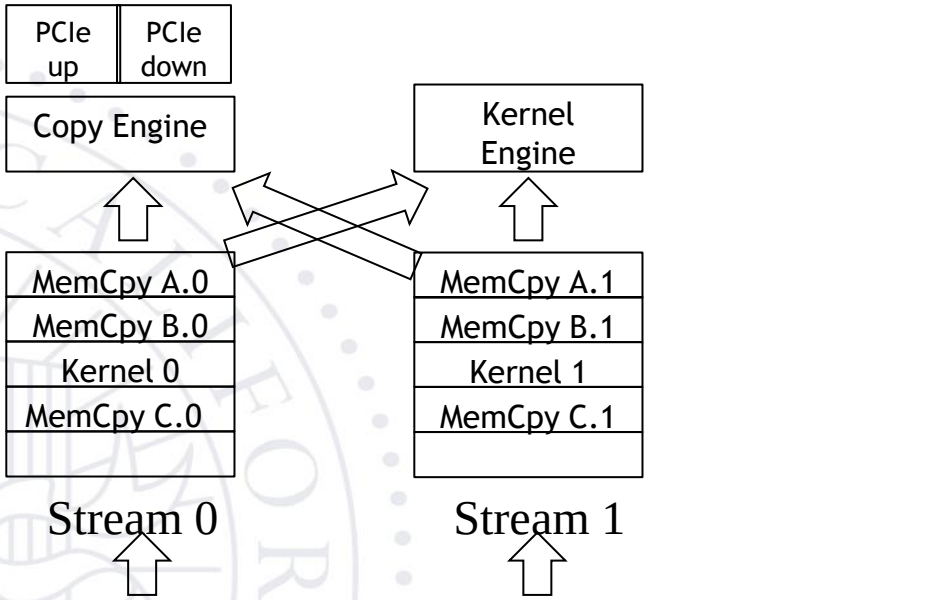


Streams cont.

- To allow concurrent copying and kernel execution, use multiple queues, called “streams”
  - CUDA “events” allow the host thread to query and synchronize with individual queues (i.e. streams).



Conceptual View of Streams



UNIVERSITY OF CALIFORNIA
UC RIVERSIDE

Overlapping data transfer w/ computation

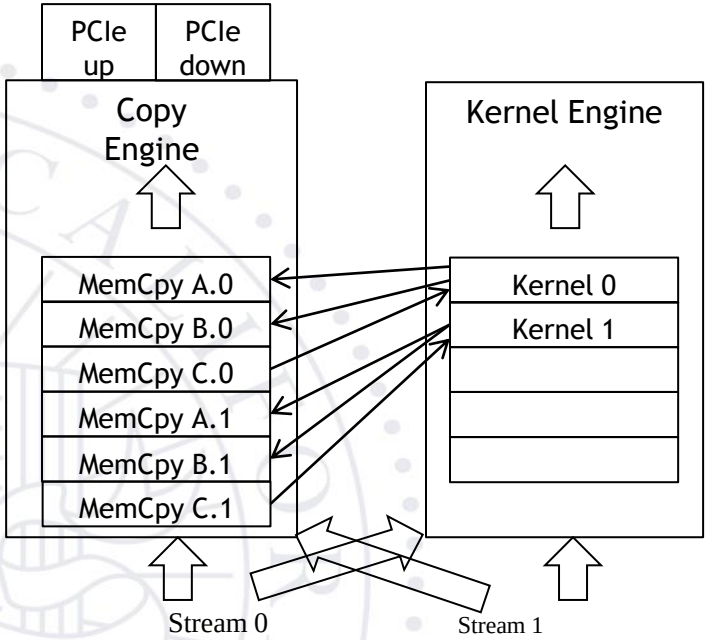
Simple Multi-Stream Host Code


```
cudaStream_t stream0, stream1;
cudaStreamCreate(&stream0);
cudaStreamCreate(&stream1);

float *d_A0, *d_B0, *d_C0; // device memory for stream 0
float *d_A1, *d_B1, *d_C1; // device memory for stream 1

// cudaMalloc() calls for d_A0, d_B0, d_C0, d_A1, d_B1, d_C1 go here
```

Simple Multi-Stream Host Code (Cont.)


```
for (int i=0; i<n; i+=SegSize*2) {
    cudaMemcpyAsync(d_A0, h_A+i, SegSize*sizeof(float),…, stream0);
    cudaMemcpyAsync(d_B0, h_B+i, SegSize*sizeof(float),…, stream0);
    vecAdd<<<SegSize/256, 256, 0, stream0>>>(d_A0, d_B0,…);
    cudaMemcpyAsync(h_C+i, d_C0, SegSize*sizeof(float),…, stream0);
    cudaMemcpyAsync(d_A1, h_A+i+SegSize, SegSize*sizeof(float),…, stream1);
    cudaMemcpyAsync(d_B1, h_B+i+SegSize, SegSize*sizeof(float),…,    stream1);
    vecAdd<<<SegSize/256, 256, 0, stream1>>>(d_A1, d_B1, …);
    cudaMemcpyAsync(d_C1, h_C+i+SegSize, SegSize*sizeof(float),…,    stream1);
}
```

A View Closer to Reality in Previous GPUs

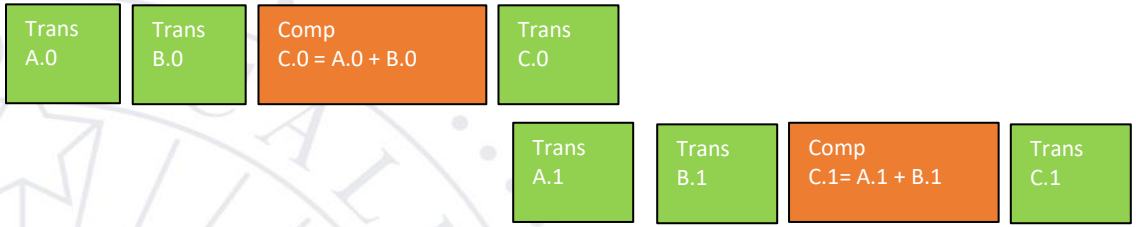


Operations (Kernel launches, `cudaMemcpy()` calls)

Not quite the overlap we want in some GPUs

UC RIVERSIDE UNIVERSITY OF CALIFORNIA

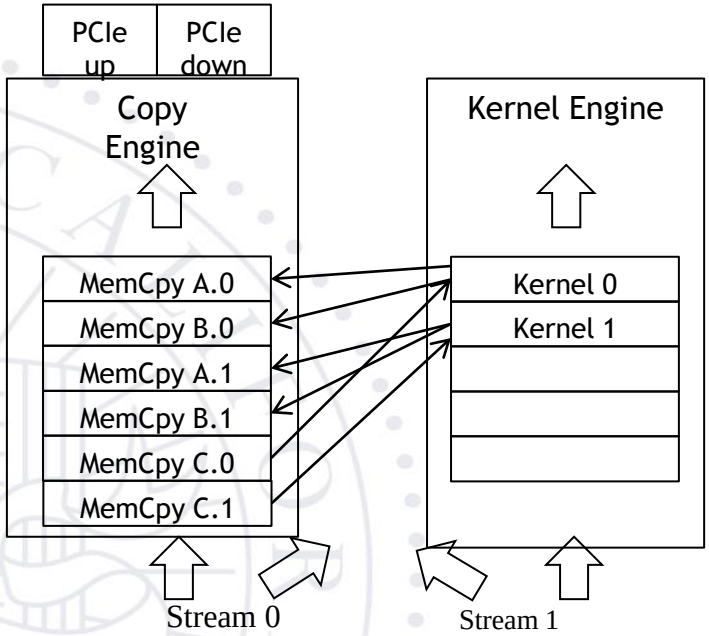
– C.0 blocks A.1 and B.1 in the copy engine queue



Better Multi-Stream Host Code

```
for (int i=0; i<n; i+=SegSize*2) {
  cudaMemcpyAsync(d_A0, h_A+i, SegSize*sizeof(float),…, stream0);
  cudaMemcpyAsync(d_B0, h_B+i, SegSize*sizeof(float),…, stream0);
  cudaMemcpyAsync(d_A1, h_A+i+SegSize, SegSize*sizeof(float),…, stream1);
  cudaMemcpyAsync(d_B1, h_B+i+SegSize, SegSize*sizeof(float),…, stream1);

  vecAdd<<<SegSize/256, 256, 0, stream0>>>(d_A0, d_B0, …);
  vecAdd<<<SegSize/256, 256, 0, stream1>>>(d_A1, d_B1, …);

  cudaMemcpyAsync(h_C+i, d_C0, SegSize*sizeof(float),…, stream0);
  cudaMemcpyAsync(h_C+i+SegSize, d_C1, SegSize*sizeof(float),…, stream1);
}
```

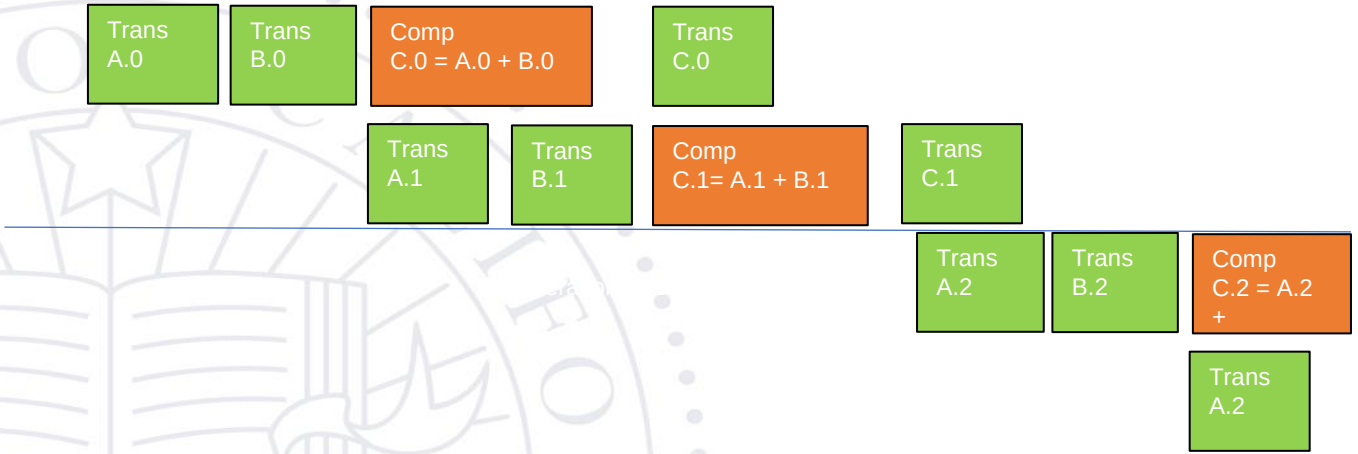
C.0 no longer blocks A.1 and B.1



Operations (Kernel launches, cudaMemcpy() calls)

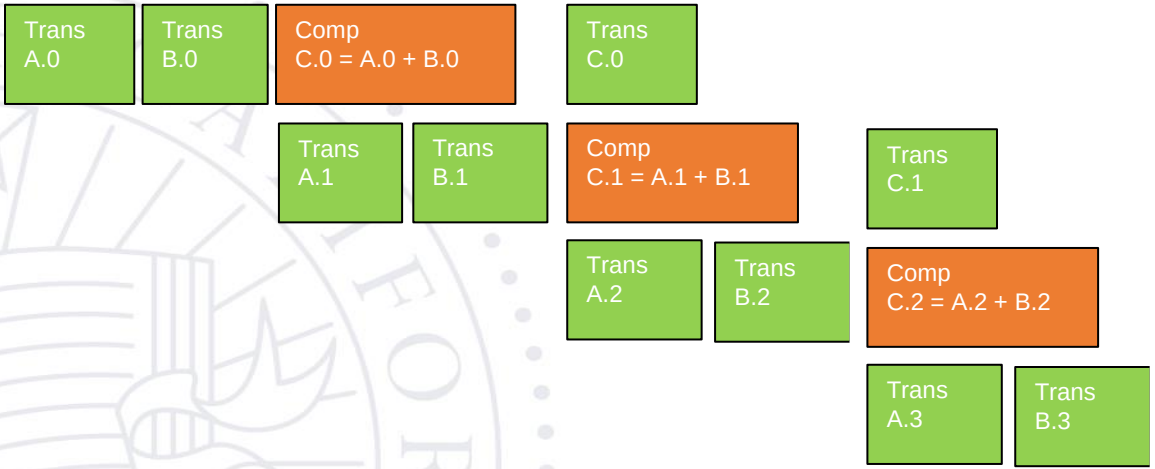
Better, not quite the best overlap

– C.1 blocks next iteration A.0 and B.0 in the copy engine queue



Ideal, Pipelined Timing

– Will need at least three buffers for each original A, B, and C, code is more complicated



Wait until all tasks have completed

UC RIVERSIDE UNIVERSITY OF CALIFORNIA

- `cudaStreamSynchronize(stream_id)`
  - Used in host code
  - Takes one parameter – stream identifier
  - Wait until all tasks in a stream have completed
  - `E.g., cudaStreamSynchronize(stream0)` in host code ensures that all tasks in the queues of stream0 have completed

- This is different from `cudaDeviceSynchronize()`
  - Also used in host code
  - No parameter
  - `cudaDeviceSynchronize()` waits until all tasks in all streams have completed for the current device