M2 - Architecture et Programmation d'accélérateurs Matériels.

(APM 2024-2025)

TP2

Asynchronism – Use of Streams

`julien.jaeger@cea.fr`
`adrien.roussel@cea.fr`

The objectives of this practical work are:

- Use CUDA Streams for asynchronous computation.
- Overlap memory transfers with computation.

I GPU Porting (default Stream)

The main source code can be found in the CODE section of this practical work (TP). The program represents the calculation of an exponential function in 1D space.

Q.1: Port the "`func`" function to the GPU, setting up a 1D grid containing enough blocks and threads to cover the entire vector.

II Streams Initialization

A CUDA stream is a sequence of operations that are executed in the order in which they are called on the GPU. This programming model is used to introduce concurrency on the GPU:
- CUDA operations from different streams can execute concurrently.
- CUDA operations between different streams can be interchanged.

First, we will create and initialize various streams based on the code created in the previous question.

Q.2: Allocate a set of `NSTREAMS` CUDA Streams and initialize them.

Q.3: Without specifying a stream on which to run a CUDA kernel, which stream is used? What makes it special?

Q.4: Modify the CUDA kernel launch so that `NSTREAMS` instances of this kernel are run on different Streams.

Q.5: What do you observe in terms of performance? Why and how can they be resolved?

Q.6: Use the cudaMemcpyAsync function to make memory copies to and from the GPU. What effect does this function have?

Don't forget to synchronize the host and device before checking the result with cudaDeviceSynchronize to avoid memory corruption.

Q.7: What is the purpose of the cudaMallocHost function? Modify your code accordingly.

Q.8: Classify functions and kernel calls into two categories: synchronous and asynchronous.

Q.9: Setting `NSTREAMS` = 2, draw a diagram representing the various tasks on the execution streams and their dependencies.
We'll use the following notation:
A $\rightarrow$ **B** : B's execution cannot start until A's has finished.
A $\Rightarrow$ **B** : B's execution cannot end until A's has finished.

Tasks list:

S0 Creation	S1 Creation	Malloc Host	Thread Sync
cpy H→D [0,N[request	cpy H→D [N,2N[request	cpy D→H [0,N[request	cpy D→H [N,2N[request
cpy H→D [0,N[execution	cpy H→D [N,2N[execution	cpy D→H [0,N[execution	cpy D→H [N,2N[execution
K1 [0,N[request	K1 [N,2N[request	K1 [0,N[execution	K1 [N,2N[execution

III Time measurement

Q.10: Use the system function `gettimeofday` to measure the time spent executing the CUDA program.

Q.11: Use the `gettimeofday` and `cudaDeviceSynchronize` functions to measure the total execution time of kernels (not including memory transfers, etc.). Here, we're asking for the total time, not the individual kernels times.

Q.12: What is the impact (disadvantage) on overall program execution of this last way of measuring? You can use the previous dependency graph as a starting point, and complete it.

Q.13: How can we measure the execution time of each kernel using the previous technique?

Q.14: How can you use events (`cudaEvent*` functions) to measure the time of each kernel and the total time of all kernels? Comment on the previous approach?

IV Matrix multiplication

Q.15: Take the matrix multiplication exercise from TP1 and transform the code to make it multi-stream. In this case, a stream will handle a packet of lines to be processed. Try first with the “coalesced” kernel and then with the tiled multiplication kernel.

What are your performance gains over the synchronous version?