Multi-GPU Stencil using MPI

PUMPS 2015

OBJECTIVE

The purpose of this lab is to understand how CUDA manages multiple GPUs and how it interacts with MPI in order to run programs across several nodes. You will optimize a naïve implementation of a 7-point stencil computation using MPI and CUDA, by using streams and page-locked memory to overlap communication and computation.

BACKGROUND

The 7-point stencil application is an example of nearest neighbor computations on an 3D input volume. Every element of the output volume is described as a weighted linear combination of the corresponding element and its 6 neighboring values in the input column, as shown in the following figure:

In the implementation provided in this lab, each thread block processes a `BLOCK_SIZE` by `BLOCK_SIZE` tile within a for-loop that iterates in the Z-direction. To simplify boundary conditions, the outer planes of all dimensions are initialized to zeros, and the thread blocks process the elements within this boundary. Thus, some of the threads in the blocks that process the outer boundary of the X-Y plane are idle, and the iteration along the Z-direction starts at plane `1` and ends at plane `nz-2`.

In the multi-GPU implementation of this computation, domain decomposition is used to partition data and computation across different GPUs. Since each output point is computed using the nearest neighbors, a data dependency is created between neighboring domains: in order to compute the points in the boundary of a domain, some extra points from the neighboring domains are needed (i.e. halos). This scheme is shown in the following figure:

This application executes the stencil kernel several times, using the output of the previous execution as input for the current one. Before starting one interation (i.e. time-step), the halo points must be updated with the new data computed by the neighboring domains in the previous iteration. This can lead to a performance loss since the next iteration cannot proceed until the halos are updated. In order to solve this problem, the points to be transferred can be computed first and the memory transfer can be overlapped with the computation of the rest of the points, as pictured in the following figure.

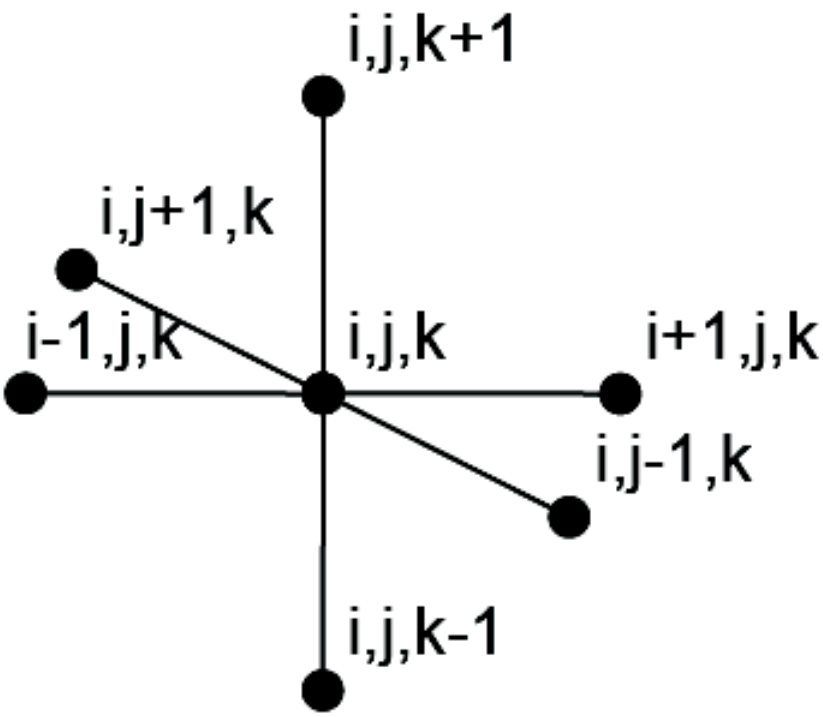

Figure 1: Stencil Shape

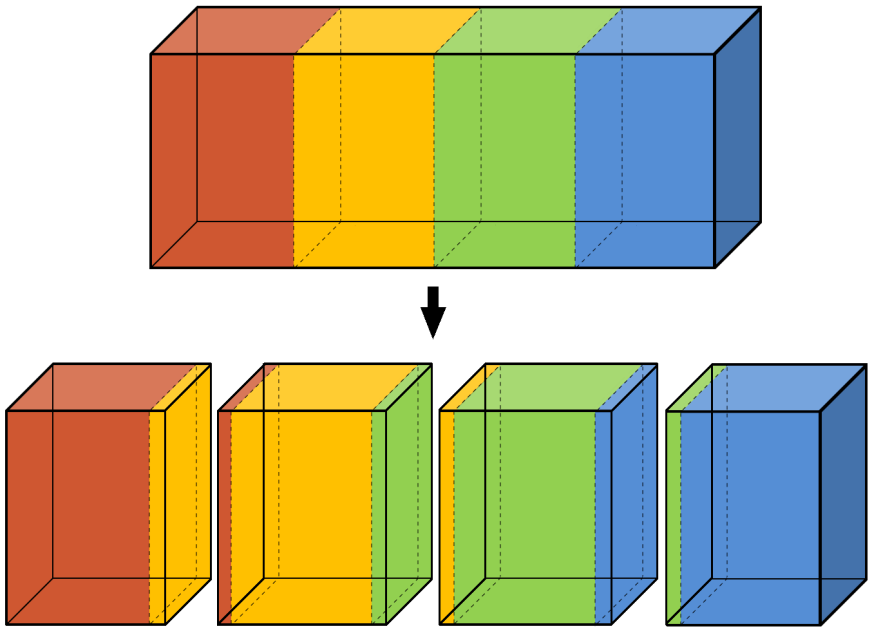

Figure 2: Domain decomposition

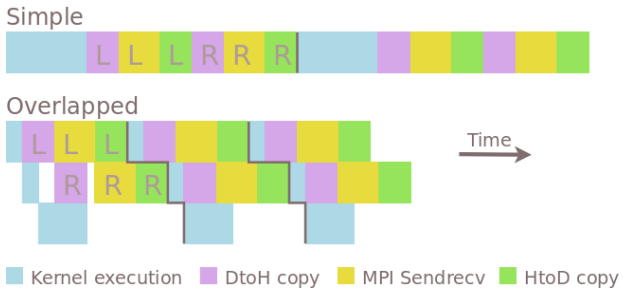

Figure 3: Timeline pictorization

This computation scheme requires the utilization of asynchronous data transfers and kernel execution, which in turn require the utilization of CUDA streams and page-locked memory. The main CUDA functions needed to manage these abstractions are:

- `cudaStreamCreate/cudaStreamDestroy`: creates/destroys a stream.
- `cudaStreamSynchronize`: blocks until all the operations launched on the stream have finished.
- `cudaMemcpyAsync`: copies data asynchronously (takes an stream as a parameter).
- `cudaHostAlloc/cudaFreeHost`: allocates/frees page-locked host memory (neededed to perform asynchronous data transfers between host and device).

INSTRUCTIONS

In the provided source code, you will find a function named `block2D_stencil`. This function implements the 7-point stencil computation by calling the `block2D_stencil_kernel` kernel. You don't have to modify this code, just call it when necessary.

An initial MPI implementation of the host code is provided as a reference (function `do_stencil`). In this version, first the GPU kernel computes all the points for a domain, and then boundary points are sent and halos are updated with the points received from the neighboring MPI processes. Then, the next iteration can proceed.

You have implement the body of the `do_stencil_overlap` function. It has to do the same as `do_stencil` but using the aforementioned communication/-computation overlap scheme to improve the application performance:

- You have to perform as many kernel calls as boundaries in the domain, plus another kernel call to compute the rest of the points. Add offsets to the input/output volumes pointers to control the part of the volume that is processed.
- You have to use CUDA streams and page-locked memory in order to perform asynchronous memory transfers. Use as many streams as needed, remember that operations must be executed on different streams to be overlapped.

- There are 2 datasets available for this lab. Both use the same volume as input data, but dataset 0 executes the provided reference implementation and dataset 1 executes the overlapped implementation that you have to write.

QUESTIONS

- Comment on the relative behavior of each implementation.
- Explain why you think certain implementations perform better than others for various input combinations.

SUGGESTIONS

- The system's autosave feature is not an excuse to not backup your code and answers to your questions regularly.
- If you have not done so already, read the tutorial
- Do not modify the template code provided – only insert code where the //@@ demarcation is placed
- Develop your solution incrementally and test each version thoroughly before moving on to the next version
- Do not wait until the last minute to attempt the lab.
- If you get stuck with boundary conditions, grab a pen and paper. It is much easier to figure out the boundary conditions there.
- Implement the serial CPU version first, this will give you an understanding of the loops
- Get the first dataset working first. The datasets are ordered so the first one is the easiest to handle
- Make sure that your algorithm handles non-regular dimensional inputs (not square or multiples of 2). The slides may present the algorithm with nice inputs, since it minimizes the conditions. The datasets reflect different sizes of input that you are expected to handle
- Make sure that you test your program using all the datasets provided (the datasets can be selected using the dropdown next to the submission button)
- Check for errors: for example, when developing CUDA code, one can check for if the function call succeeded and print an error if not via the following macro:

```
#define wbCheck(stmt) do {                                          \
        cudaError_t err = stmt;                                     \
        if (err != cudaSuccess) {                                   \
            wbLog(ERROR, "Failed to run stmt ", #stmt);             \
            wbLog(ERROR, "CUDA error ...  ", cudaGetErrorString(err));\
            return -1;                                              \
```

```
        }                                                                \
    } while(0)
```

An example usage is `wbCheck(cudaMalloc(...))`. A similar macro can be developed while programming OpenCL code.

LOCAL DEVELOPMENT

While not required, the library used throughout the course can be downloaded from Github. The library does not depend on any external library (and should be cross platform), you can use `make` to generate the shared object file (further instructions are found on the Github page). Linking against the library would allow you to get similar behavior to the web interface (minus the imposed limitations). Once linked against the library, you can launch your program as follows:

```
./program -e <expected_output_file> \
    -i <input_file_1>,<input_file_2> -o <output_file> -t <type>
```

The `<expected_output_file>` and `<input_file_n>` are the input and output files provided in the dataset. The `<output_file>` is the location you'd like to place the output from your program. The `<type>` is the output file type: `vector`, `matrix`, or `image`. If an MP does not expect an input or output, then pass `none` as the parameter.

In case of issues or suggestions with the library, please report them through the issue tracker on Github.