Register Tiled Stencil

PUMPS 2015

OBJECTIVE

The purpose of this lab is to practice the thread coarsening and register tiling optimization techniques using 7-point stencil as an example.

INSTRUCTIONS

- Edit the code to implement a 7-point stencil with combined register tiling and x-y shared memory tiling, and thread coarsening along the y-dimension.
- Edit the code to launch the kernel you implemented. The function should launch 2D CUDA grid and blocks, where each thread is responsible for computing an entire column in the y-dimension.
- Answer the questions found in the questions tab.

ALGORITHM

You will be implementing a simple 7-point stencil without having to deal with boundary conditions. The result is clamped so the range is between 0 and 255.

```
for i from 1 to height-1:   # notice the ranges exclude the boundary
  for j from 1 to width-1:  # this is done for simplification
    for k from 1 to depth-1:# the output is set to 0 along the boundary
      res = in(i, j, k + 1) + in(i, j, k - 1) + in(i, j + 1, k) +
            in(i, j - 1, k) + in(i + 1, j, k) + in(i - 1, j, k) -
            6 * in(i, j, k)
      out(i, j, k) = Clamp(res, 0, 255)
```

With `Clamp` defined as

```
def Clamp(val, start, end):
  return Max(Min(val, end), start)
```

And `in(i, j, k)` and `out(i, j, k)` are helper functions defined as

```
#define value(arry, i, j, k) arry[(( i )*width + (j)) * depth + (k)]
#define in(i, j, k)   value(input_array, i, j, k)
#define out(i, j, k)  value(output_array, i, j, k)
```

QUESTIONS

- Briefly describe your implementation in one or two paragraphs as well as any difficulties you faced in developing and optimizing the kernel.
- Consider a 100x100x50 7-point stencil that does not use any thread-coarsening or tiling optimizations. How much real computation does each work-item perform? How many global memory loads does each work-item perform? What is the ratio of computation to global memory loads for each work-item? (Consider as useful work only additions, subtractions, and multiplications that operate directly on the loaded data. Do not consider index computations.)
- Consider a 100x100x50 7-point stencil that uses thread-coarsening and joint tiling optimizations such as the one you implemented. How much real computation does each work-item perform? How many global memory loads does each work-item perform? What is the ratio of computation to global memory loads for each work-item? (Consider as useful work only additions, subtractions, and multiplications that operate directly on the loaded data. Do not consider index computations.)
- Briefly comment on the difference in computation to global memory loads ratios for the two cases.

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.