Homework #1 (HW1)

Autumn 2022 - GPU Computing

DUE: 17 October 2022, 5:59 PM

Learning Objectives

- use the CUDA error checking macro
- implement kernels which use multidimensional indexing
- launch kernels from host using multidimensional execution configurations
- practice row-major access of 2D and 3D flattened/linearized datasets
- implement range bounds-checking inside kernels
- Perform result verification in the host application, between CPU baseline and GPU compute kernel versions, accounting for the possibility of numerical precision issues
- get experience implementing additional GPU kernels with CUDA

Overview

The HW1 includes two main parts: (i) answering some questions designed to reinforce your technical understanding of the material learned in the lectures and reading set, and (ii) a programming section where you will practice working with the CUDA host API and CUDA gpu device kernel programming.

Questions

Please turn in written (or typed) solutions to these problems, including showing your work and/or describing your reasoning/explanation for the selected answer.

1. If we need to use each thread to calculate one output element of a vector addition, what would be the expression for mapping the thread/block indices to data index:

    A. i=threadIdx.x + threadIdx.y;
    B. i=blockIdx.x + threadIdx.x;
    C. i=blockIdx.x*blockDim.x + threadIdx.x;
    D. i=blockIdx.x * threadIdx.x;

2. We want to use each thread to calculate two (adjacent) output elements of a vector addition. Assume that variable i should be the index for the first element to be processed by a thread. What would be the expression for mapping the thread/block indices to data index of the first element?

A. i=blockIdx.x*blockDim.x + threadIdx.x +2;
B. i=blockIdx.x*threadIdx.x*2
C. i=(blockIdx.x*blockDim.x + threadIdx.x)*2
D. i=blockIdx.x*blockDim.x*2 + threadIdx.x

3. We want to use each thread to calculate two output elements of a vector addition. Each thread block processes 2*blockDim.x consecutive elements that form two sections. All threads in each block will first process a section, each processing one element. They will then all move to the next section, again each processing one element. Assume that variable i should be the index for the first element to be processed by a thread. What would be the expression for mapping the thread/block indices to data index of the first element?

A. i=blockIdx.x*blockDim.x + threadIdx.x +2;
B. i=blockIdx.x*threadIdx.x*2
C. i=(blockIdx.x*blockDim.x + threadIdx.x)*2
D. i=blockIdx.x*blockDim.x*2 + threadIdx.x

4. For a vector addition, assume that the vector length is 8000, each thread calculates one output element, and the thread block size is 1024 threads. The programmer configures the kernel launch to have a minimal number of thread blocks to cover all output elements. How many threads will be in the grid?

A. 8000
B. 8196
C. 8192
D. 8200

Programming Exercises

Begin from a new default CUDA visual studio solution & project (like you created in EX1). Notice that the example defines a static arraySize of 5 elements - hardly a computationally hard problem!

Also it statically assigns and initializes the source arrays a, b - not practical for a real problem size! This also doesn't require any dynamic host-side memory allocation (which we did in EX1 with malloc( ) calls)
REMINDER: in C you have to manage your own memory, so be sure to pair a free( ) call for each malloc( ) to release the allocated heap memory when it is no longer used in the code!

```
int * pData = (int*)malloc(sizeData * sizeof(int);
…
free(pData);
```

Lastly the example launches a kernel composed of a single threadblock with size=5 threads. Again, not too awe inspiring!

```
// Launch a kernel on the GPU with one thread for each element.
    addKernel<<<1, size>>>(dev_c, dev_a, dev_b);
```

In this homework (HW1) you will implement several new kernels, of progressive sophistication, which will give you a change to practice with multidimensional assignment of CUDA grids/blocks/threads and determination of the corresponding thread indices inside the kernels, to allow accessing the data elements in a precise way.
You will also implement bounds checking in the kernels to avoid threads accessing data locations outside the desired N-dimensional computation range.

Error checking on host

Update the example main( ) and addWithCuda( ) functions to use the CUDA Runtime API error checking macro provided in Lecture 1 slides, for all CUDA functions which return `cudaError_t`.

- Put the error macro in a separate *.h header file which you then `#include` in the kernel.cu source code file
    - this error macro will be reused in many future projects

Note: To create the new header file, right click your project in the VSCode solution explorer (left panel), and go to `Add-> New Item -> Header File (.h)`

CUDA Kernels:

In all the kernels which implement a mathematical computation function: SAXPY, 2D matrix add, 3D grid add, you will need to allocate memory for the input & output datasets on the host and initialize the data elements to known values. The same set of input values can then be used for both the host-side CPU sequential version and the device-side GPU kernel computation - see section “Host-Side Result Verification”.
You will also need to add the execution configuration syntax <<< … >>> in the host application to launch each of the new kernels and pass in the kernel arguments.

printf kernel

1. Inside the kernel function, obtain the thread ID, block ID using the CUDA built-in variables, and use them to compute the globally unique thread IDentifier (GUTID).

2. Use printf( ) inside the kernel function body to print out the values of the thread ID, block ID, and GUTID on a single line (use the "\n" end-line character to include a carriage return).
3. in the host application, experiment with various values of the execution configuration (exec cfg) grid and block sizes, such as
    a. <<<1,16>>>
    b. <<<4,4>>>
    c. <<<16,1>>>
    d. Run each case several times (like 10 times) and observe the result behavior.
4. Some questions to consider:
    a. What do you observe about the results?
    b. Does the behavior of block and thread indices seem to differ?
    c. What does this imply about computations you perform in the kernels?

SAXPY kernel

SAXPY stands for *Single-Precision A times X plus Y*.
Thus all the terms are single-precision (32-bit) floats where

- *a* is a scalar float multiplier value
- **x** is a vector of length 1xN
- **y** is a vector of length 1xN

The expression to be computed is

y = *a***x** + **y**

SAXPY is one of the BLAS (Basic Linear Algebra Subroutines) Level-1 routines, and is known to take linear time *O*(n) [We'll cover algorithmic complexity and asymptotic notation officially in a coming lecture).
Include the integer vector dimension N in the kernel input arguments passed from the host so that the kernel can perform bounds-checking.

- Test cases:
    - N = 10 // this is a sanity check case to ensure it's working right
    - N = 1e3 // scaling things up
    - N = 1e7 // everything still passing?

2D matrix addition kernel

In this kernel you'll implement a 2D matrix addition. The expression to be computed is

C = **A** + **B**

where A, B are input MxN matrices and C is an output MxN matrix.
Include the integer matrix dimensions M and N in the kernel arguments so that the kernel can perform the 2D bounds-checking on each thread of execution.

- Test cases:

- MxN = 4x4 // this is a sanity check case to ensure it's working right
- 500x500 // this is a small matrix
- 3024x4032 // like a current iPhone picture!

3D grid addition kernel

In this kernel you'll implement a 3D grid addition. The expression to be computed is

C = **A** + **B**

where A, B are input MxNxP float data grids and C is an output MxNxP data grid
Include the integer grid dimensions M, N, P in the kernel input arguments so that the kernel can perform the 3D bounds-checking on each thread of execution.

- Test cases:
  - MxNxP = 3x3x3 // this is a sanity check case to ensure it's working right
  - 100^3 // total data points = 1e6
  - $1e3^3$?!? // total data points = 1e9

Host-Side Result Verification

1. As discussed in Lecture #2, include result verification routines in the host application code to VERIFY results of same the computations on CPU vs GPU
   a. Thus a sequential CPU host routine will need to be implemented for each calculation being performed by the GPU in the kernels implemented in this HW1.
   b. Use a tolerance threshold for result agreement to account for numerical precision differences, as discussed in Lecture 2.
   c. HW Q: At what tolerance value do verifications start to fail?
   d. HW Q: which of the 3 kernels exhibits the most severe result disagreements between CPU and GPU versions? Why?

SUBMIT

1. Submit answers to the questions, including shown work and explained reasoning.
2. Submit your CUDA host application source code
   a. .cu, .cpp and .h files only - do not submit all project & solution files and other build by-products!
3. Submit your 4 kernel functions