Homework 4 assigned. Due December 1, 2017 11:59pm MT.
Preliminaries:

1) Submission:

Send me by email to pouchet@colostate.edu, subject "CS560: assignment 4 -- your name" an archive (.zip or .tar.gz) containing all the files asked below, and a PDF document with your answers to the various questions.

Precisely, for each question you need to provide me either the ISCC script, the modified .c file, or some text for your answer. That is, for each and every question below you need to provide me something. Please ask on Piazza if you have any doubt about what to submit.

Make sure your files all have unique name, e.g., "gemm.exercise1.question1.iscc" or "gemm simp.exercise4.question1.cu", so that I can easily see which file corresponds to which question.

The hard deadline is Friday, December 1 2017, 11:59pm MT. NO EXTENSION WILL BE PROVIDED. To be sure, ABSOLUTELY NO EXTENSION WILL BE PROVIDED. ;-)

2) DISCLAIMER:

This assignment is currently "incomplete" as the very last exercise (worth 10% of the grade) requires you compile your program with a CUDA compiler, and evaluate it on a NVIDIA GPU. I will provide you with a procedure to compile and run a CUDA program in the coming days, but the entire assignment up to this last exercise can be completed without a CUDA compiler/GPU.

3) Grading:

Exercises 1-3 are worth 90%, Exercise 4 10%. An extra 20% is possible via Exercise 5 (optional, extra credit). It will not be graded/considered unless Exercises 1-4 are completed.

4) Advices:

- You will not be graded based on the performance of the program you obtained, but based on your completion of each of the stages. That is, there is no expectation of "minimal performance improvement" your solution must achieve to get 100% to a question.

- This assignment is meant to make you accelerate a simple computation (matrix-multiply) on GPUs. The objective is to make you walk through the various steps involved in producing such code. Providing me with a CUDA implementation for matrix-multiply you found on the internet is NOT satisfactory for this assignment, you must go through the steps yourself as asked.

- This assignment builds on previous assignments 1-3. Refer to those as needed for additional details.

- This assignment does not provide you with all the information you need. In particular, this is up to you to learn CUDA programming basics to complete several questions. To many extent, the objective of this assignment is making sure you have gone through these documentations, and are able to write a CUDA program on your own.

Assignment:

This assignment focuses on the exact same file/benchmark as Assignment 3:
polybench-c-4.2.1-beta/linear-algebra/blas/gemm.c from PolyBench/C 4.2.1, visit
http://polybench.sf.net
and download the file:

polybench-c-4.2.1-beta.tar.gz

to get the entire test suite.

The kernel of interest is:

```
#pragma scop
 for (i = 0; i < _PB_NI; i++) {
   for (j = 0; j < _PB_NJ; j++)
R:   C[i][j] *= beta;
   for (k = 0; k < _PB_NK; k++) {
      for (j = 0; j < _PB_NJ; j++)
S:      C[i][j] += alpha * A[i][k] * B[k][j];
    }
 }
#pragma endscop
```

Exercise 1:

This exercise aims at determining, prior to any work on generating a CUDA program, whether it could be beneficial to offload a computation from a CPU to a GPU.

Question 1:

Build the polyhedral representation (that is, write the ISCC script, as for assignment 2) of the kernel above, that is the iteration domain of each statement, and each access function. Count how many point are in each iteration domain.

Question 2:

Compute the data space of the entire program using ISCC. Count how many points are in the data space.

Question 3:

Assume the following characteristics/values:
- GPU: peak 1 TeraFlop/s, bandwidth RAM to/from GPU: 30GB/s
- CPU: peak 100 GigaFlop/s, bandwidth RAM to/from CPU: 30GB/s
- PB_NI = PB_NJ = PB_NK = 1000
- data is floating point double-precision.

Compute the minimal execution time of gemm on CPU and GPU using the roofline approach:
minimal_exec_time = max(computation_time, communication_time), where:

computation_time = (number of flops in the kernel) / (peak flop/s)
communication_time = (number of bytes in the data space) / (bandwidth)

Use the expressions from Questions 1 and 2 for the iteration domain sizes and data space size, multiplying them ("manually", outside of ISCC) as needed to compute correctly the number of flops / bytes in the data space, and minimal_exec_time for both CPU and GPU. What is the potential execution time reduction possible by offloading this code to a GPU?

Exercise 2:

This exercise aims at transforming the input program to expose enough parallelism to ease the job of porting the code to a GPU. You will use the pocc compiler, as in Assignment 3, to perform this task. Refer to the previous assignment for installation instructions.
Starting this exercise till the end of the assignment, we work on a simplified version of the gemm kernel. The new code is:

```
#pragma scop
 for (i = 0; i < _PB_NI; i++)
   for (k = 0; k < _PB_NK; k++)
      for (j = 0; j < _PB_NJ; j++)
S:      C[i][j] += alpha * A[i][k] * B[k][j];
#pragma endscop
```

This simplification to a single-statement perfectly-nested program will ease manual modifications later on.

Question 1:

Create the file gemm-simp.c by duplicating gemm.c, and changing the kernel to the simplified one above. Then, produce a transformed C program (e.g., gemm-simp.pocc.c) by compiling gemm-simp.c following the commands:

```
$> echo "4 4 4" > tile.sizes
$> ./bin/pocc --verbose --pluto-tile gemm-simp.c
$> rm tile.sizes
```

Question 2:

For each of the 6 loops in the gemm-simp.pocc.c kernel, say if the loop is parallel (it is not carrying any dependence) or if it is sequential. Beware that the above PoCC transformation has changed the order of loops compared to the original code, to put parallel loops outer-most as much as possible.

Exercise 3:

This exercise aims at generating a "naive" CUDA implementation from gemm-simp.pocc.c, where the outer-most two loops (c0 and c1) are translated into CUDA threads.

Question 1:

Compute how many times the loop nest made of loops {c2,c3,c4,c5} is executed in the program, using ISCC. Note you are now allowed to go beyond affine expressions, and use / (integer division) and ceil/floor operators in the description of ISCC structures.
Assuming NI=NJ=NK=1000, if we map each execution of {c2,c3,c4,c5} to a distinct CUDA thread, how many threads in total would be needed?

Question 2:

In CUDA programs, there is a hard limit on the number of threads per thread block, a thread block cannot have more than 1024 threads. Let us assume that we use loop c0 to model iterations over thread blocks, and loop c1 to model iterations over threads inside a block (i.e., for a particular value of c0, that is a particular thread block). What is the range of values of PB_NI, PB_NJ and PB_NK for gemm-simp.pocc.c which ensures we are not using more than 1024 threads per thread block?

Question 3:

We now map loop c0 to thread blocks, and loop c1 to threads inside a thread block, and we assume PB_NI, PB_NJ and PB_NK are such that no more than 1024 threads per thread block is needed/generated. Write the CUDA kernel function that executes {c2,c3,c4,c5,c6} in each thread, by manually modifying the code to replace references to c0 and c1 by the adequate expressions involving BlockDim.x, BlockIdx.x and ThreadId.x.

You can refer to:
http://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf
and also:
http://users.wfu.edu/choss/CUDA/docs/Lecture%205.pdf
which provides CUDA examples for matrix-multiply.

To be clear, this question requires you to manually edit gemm-simp.pocc.c as needed to create a CUDA kernel version of the function kernel_gemm(). You are not expected to use ISCC or PoCC to do this conversion.

Question 4:

As a follow-up to Question 3, write the host code for this computation. Implement the entire A, B and C matrices copy from host to GPU, and only the copy back of C from GPU to host. Refer to the documents above for details. Note that you need to generate in the host code the call to your GPU kernel. In this mapping strategy, we use 1D thread blocks. The number of blocks is given by the number of iterations of the loop c0, the number of threads in a block is given by the number of iterations of the loop c1. So you should end up with something similar to:

```
gemm_kernel_gpu<<<some_expression1, some_expression_2>>>(...)
```

where some_expression1, some_expression2 come from the loop bound of c0 and c1, respectively.

Exercise 4:

This exercise aims at testing your CUDA program. To complete this
exercise (worth 10% of the assignment) you need a CUDA compiler (nvcc
/ NVIDIA CUDA C++ SDK installed) as well as a CUDA-capable GPU.

Question 1:

Make all edits needed to obtain a stand-alone gemm-simp.pocc.cu file, which can compile with nvcc. Likely if Exercise 3 was completed correctly this is only about renaming the file with the .cu extension, but you may need to replace the various 'polybench_alloc_data' calls by a simple malloc() on some setups if you see a compilation error related to 'posix_memalign' function being unavailable.

Question 2:

We now assume PB_NI = PB_NJ = PB_NK = 1024. Compile and run the program for this problem size, and report the execution time of the CUDA kernel. You can simply add a timer around the cuda calls (from the first cudaMemcpy to the last one) or use nvprof. What is the GigaFlop/s achieved by your implementation?

Question 3:

Taking the best performing version you obtained for Assignment 3, compute and report its GigaFlop/s achieved, and compare it to what the GPU code achieved: is it faster?

Exercise 5 (optional, extra credits):

Note the optional exercise is hard, if you want to do it please discuss with me first for the best way to quickly implement what is needed. Two choices are offered:

Choice 1:
Implement a script that explores different tile sizes (instead of only 4x4x4 that we have set above) to find the best performing one. Limit to 20 or less tile sizes, and make sure each value is always a power of 2. It means you need to automate code generation, which for this situation is actually very simple by using some sed / text substitution. Please discuss with me before starting this question, so that I give you adequate material to ease automation.

Choice 2:
Implement the use of shared memory for the program, for one execution of the {c4,c5} loop nest. Follow assignment 3 to compute data spaces, copy-in/out codes and local arrays size. No automation is required here, but a ISCC script to compute the relevant structures/codes/sizes is needed.

Thanks,

++