CS410 Assignment 4
Matrix Multiplication on GPUs

Date released: Apr/15/2022.
Due date: Apr/25/2022 (11:59 pm IST).

In this assignment, you will implement (yet again) the widely used **matrix multiplication** algorithm for execution on GPUs using CUDA-C**.** You will implement and experiment with the algorithm based on the inner product (`ijk` loop). The following code shows the `'ijk'` loop for computing `C=C+A*B`:

```
C = zeros(n,n);
for i = 1 to n
      for j = 1 to n
            for k = 1 to n
                  C(i,j) = C(i,j) + A(i,k) * B(k,j);
```

The outer two loops can be imagined to be executed by a 2D grid of processors computing the C matrix in parallel. You can assume that the input matrix sizes are `nxn`.

0) You will have to implement the parallel matrix multiplication algorithm using CUDA-C. Call this `ver0.cu`. Your parallelization strategy should have each thread compute one element of the C matrix. Each thread should load a row of matrix A from memory, a column of matrix B from memory, compute an element of matrix C and store the result in memory. As part of this implementation, you will have to:
    a. Allocate and free global memory on GPU
    b. Copy data from CPU to GPU global memory and from GPU global memory to CPU.
    c. Launch the kernel on the GPU
    d. Free the device and host memory

   Your program should accept one argument, n, which indicates the square matrix size.
   Initialize matrices with random 32-bit integers.

1) You will have to improve ver0.cu by adopting a parallelization strategy, where the matrix C is divided into tiles based on the TILE_SIZE parameter and each tile of C is computed by a thread block from a grid of thread blocks. Each block computes a tile of C through multiple iterations (there will be n/TILE_SIZE iterations): in each iteration, **i)** a tile of A and a tile of B are loaded from global memory to shared memory. Recall the solution to SUMMA that we discussed in class: to compute a tile of C located at ($i\hat{}$, $j\hat{}$) (, i.e. at (`row, column`) with `row` as the index of the tile along the y-axis and `column` as the index of the tile along the x-axis.), you need all tiles along row $i\hat{}$ from A and all tiles from column $j\hat{}$ of B. *Note that CUDA thread with ID (`row, column`) indicates a thread at `row` along the x-axis and at `column` along the y-axis in a 2D grid.* **b)** Every thread within the block computes an inner product TileA[i][k] * TileB[k][j] to compute the result in a thread-local variable. The

value of the inner product accumulated over n/TILE_SIZE iterations is then written back to global memory. To ensure that all threads within a block have access to the same data within the shared memory, you should synchronize (i.e. insert a barrier) after **a)** and **b)**. Call this `ver1.cu`. As part of this implementation, you will have to (in addition to CUDA features used in `ver0.cu`):

a. Use __shared__
b. Use __host__
c. Use __syncthreads
d. Measure time for steps a and b separately to understand the time for memory related operations and compute related operations.

You may assume that TILE_SIZE divides n

You will use the IITDH DGX system to compile and run your code.
You may do the project alone or in a group of two students.
You may discuss ideas but do not share the code with another team.

Distribution and collection of assignments will be managed using **GitHub Classroom**. You can accept the assignment at the URL shared with you.

When you accept the assignment, it will provide you with a clone of a GitHub repository that contains just this problem statement.

Experiments

When you compile your code using nvcc, make sure to use -O3 optimization. All of the execution measurements described below should be performed with the executable produced using this optimization.

0) With `ver0.cu`, vary the size of your matrices from n=1024, 2048, 4096, 8192. Measure the runtime (separately for data transfer time and device computation time). Plot runtime vs. n.
1) With `ver1.cu`, Set TILE_SIZE=32 and for each matrix size of n=1024, 2048, 4096, 8192 Measure the runtime (separately for data transfer time and device computation time). Plot the speedup w.r.t. corresponding `ver0.cu` runs. Provide an explanation.

   Along with each version, implement a function for computing matrix multiplication on the host (you can use the 'ijk' ordering or any other ordering.). This function must be used to validate your results obtained from the device.

Written report

Write a report that describes how your program works. This report should NOT include your program, though it may include one or more figures containing *pseudo-code* that sketches key elements of the parallelization strategy.

0) *For each version of the implementation, explain how your program exploits parallelism. Your response must touch upon SIMD parallelism in addition to other strategies that you adopt.*
1) *After showing the speedup vs. n plot as per experiment 1, explain why you see the speedup that you see. Your response must touch upon the communication costs in addition to other strategies that you adopt.*

Submitting your assignment

Your assignment should be submitted in three parts.

- One or more source files containing your different versions of the program. Your submission must contain a Makefile with the following targets:
  - `team`: prints the team members' names on the terminal.
  - `exp0`: builds related code and runs experiments mentioned in Experiment 0).
  - `exp1`: builds related code and runs experiments mentioned in Experiment 1).
- A report about your program in PDF format. Guidelines for your report: Make sure that you address all the requirements specified in the "Experiments" section. Don't forget to add Axis headers, legends, figure title, discussion corresponding to a figure, and averaging from multiple runs.
- Data from your experiments. Include 2 files ver0.log and ver1.log that contain consolidated data from experiment 0 and experiment 1 respectively. This is the output that is written to a file after your job completes.

You will submit your assignment by committing your source files, Makefile, and report to your GitHub Classroom repository. If you have trouble using GitHub and come up against the deadline for submitting your assignment, email the instructor a tar file containing your source code and your report.

Click on the link shared above. This will create a repository in your GitHub account.

1. Clone the repository into your local development environment (on DGX master node) using the **git clone** command.
2. Add all assignment related files that you want to submit to your GitHub local repository using the **git add** command.
3. Save the changes using the **git commit** command
4. Upload the changes to GitHub using the **git push** command
5. Release your changes by first tagging your commit on the local environment using the **git tag -a cs410assignment4 -m "submitting assignment 4"**.
6. Next, push the tag to GitHub with the help of the following command: **git push --tags** \\
   If you want to make changes after you have submitted (repeat the above steps from 1 to 5 and apply modified commands shown below in place of step 4):
   git tag -a -f git tag -a cs410assignment4 -m "submitting assignment 4"

git push -f --tags

Grading criteria

- 40% for conducting experiments: split as 20% ver0 and 20% for ver1 (the log files are used for evaluating this.)
- 20% for correctness and completeness of code: 15% for GPU kernels and supporting code, 4% for writing the validate code that runs on CPU, 1% for correct makefile.
- 10% for performance:
- 30% Report. Your grade on the report will consider the quality of the writing (grammar, sentence structure), whether all of the required elements are included, and the clarity of your explanations. Make sure that you have performed all of the experiments requested above and provided the information requested about them in your report. If you haven't, you will lose points!

Using the DGX machine

1. You must be connected to the VPN to access the DGX master node.
    a. Follow these instructions to connect to VPN.
2. `ssh <your id>@10.250.101.55`
3. Log onto 10.250.101.55 with your id and password.

Using SLURM on DGX to submit jobs

The difference from earlier scripts that you used in assignments 1 and 2 is that you must use the a100 queue and request for a GPU device. See sample script below to know how to do this.

```
#!/bin/bash
#SBATCH --job-name=GPU_JOB            # Job name
#SBATCH --partition=a100
#SBATCH --mail-user=email@iitdh.ac.in # Where to send mail
#SBATCH --nodes=1
#SBATCH --time=00:01:00
#SBATCH --gres=gpu:1
#set -x
#SBATCH --output=helloworld_%j.log   # Standard output and error
log
#SBATCH --error=err_helloworld%j.out

srun /iitdh/guest/a.out
```

The default queue for the DGX machine is `v100`. However, this queue is not used for this assignment. Instead, you must use the queue `a100`.