

CS 629/729 GPU Programming

Lab 2

Spring 2024

Learning Outcomes

- Understand how to use a GPU cluster server with job scheduler
- Understand how to write a CUDA program from scratch
- Understand how to use shared memory to improve performance of a memory bound algorithm (matrix multiply)
- Understand how to use tiling in shared memory for matrix multiply
- Understand how to design computing and data copy pipeline (CS 729 only)

Task 0: Set up GPU cluster server account

1. Follow the instruction provided by the TA to set up your account with DUO MFA
2. Log into asax.asc.edu using ssh with your username and temporary password using command:

```
ssh <username>@asax.asc.edu
```

3. The first time you login, you will have to enter the password, authenticate via DUO, then set a new password (entered twice). Nothing is displayed when you type these passwords.
4. Create a test.cu file in your home directory with the following testing code

```
#include <iostream>

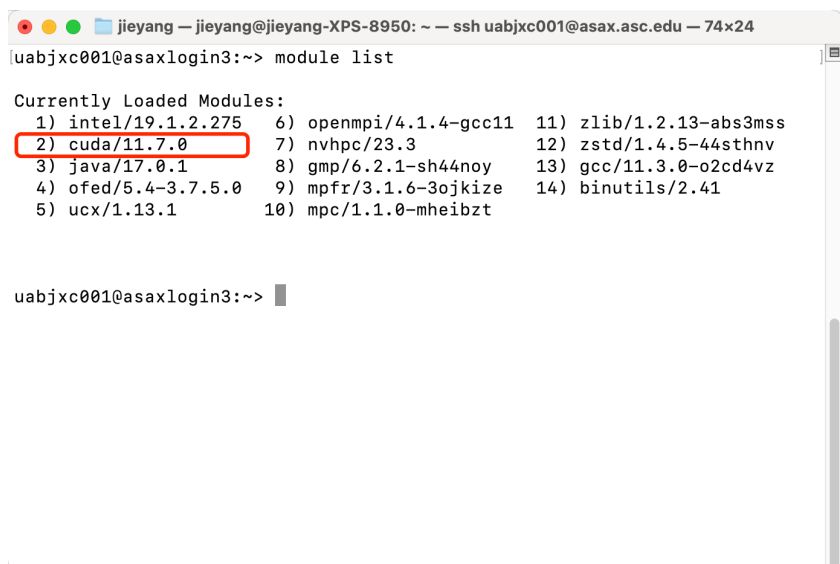
int main(){
    std::cout << "hello world!\n";
    return 0;
}
```

5. Load cuda module using command

```
module load cuda
```

6. Confirm the cuda module is loaded using command:

```
module list
```



```
uabjxc001@asaxlogin3:~> module list

Currently Loaded Modules:
 1) intel/19.1.2.275   6) openmpi/4.1.4-gcc11 11) zlib/1.2.13-abs3mss
 2) cuda/11.7.0       7) nvhpc/23.3          12) zstd/1.4.5-44sthnv
 3) java/17.0.1       8) gmp/6.2.1-sh44noy   13) gcc/11.3.0-o2cd4vz
 4) ofed/5.4-3.7.5.0  9) mpfr/3.1.6-3ojkize  14) binutils/2.41
 5) ucx/1.13.1        10) mpc/1.1.0-mheibzt
```

7. Compile your code using command to make sure the code is compiled for both A100 and H100.

```
nvcc test.cu -o test --generate-code arch=compute_80,code=sm_80
--generate-code arch=compute_90,code=sm_90
```

8. Create a run script file (my_run_script.sh) with the following content:

```
#!/bin/bash
module load cuda
./test
```

9. Make your run script executable by executing:

```
chmod +x my_run_script.sh
```

10. Submit your run script to the PBS job queue by executing:

```
run_gpu my_run_script.sh
```

11. You will be asking to configure your job by specifying

- Queue Name (type: gpu)
- Number of processor cores (CPU cores) (type: 1)
- Time limit (type: 1:00:00) minimum is 1 hour
- Memory limit (type: 1gb)
- Number of GPUs (type: 1)
- GPU architecture (type: any)
- Job name (type anything meaningful to you)



```
jiayang — jieyang@jieyang-XPS-8950: ~ — ssh uabjxc001@asax.asc.edu — 93x33

Enter Queue Name (default <cr>: gpu) gpu
Enter number of processor cores (default <cr>: 1 ) 1
Enter Time Limit (default <cr>: 01:00:00 HH:MM:SS) 1:00:00
Enter memory limit (default <cr>: 1gb ) 1gb
Enter the number of GPUs (default <cr>: 1) 1
Enter GPU architecture [ampere/hopper] (default <cr>: any) any
Enter a name for your job (default: runscriptGPU)
test

=====
==== Summary of your GPU job =====
=====
The script file is: run_script
The time limit is 1:00:00 HH:MM:SS.
The memory limit is: 1gb
The job will start running after: 202402061025.51
Job Name: test
Virtual queue: -q gpu
Queue: -q gpu
Constraints:
Queue submit command:
qsub -q gpu -j oe -N test -a 202402061025.51 -r n -M jchen3@uab.edu -l walltime=1:00:00 -l s
elect=1:ngpus=1:ncpus=1:mpiprocs=1:mem=1000mb

11126.asax-pbs1
uabjxc001@asaxlogin3:~>
```

12. You job will be submitted to the gpu job queue waiting to be executed. Use this command to check its status: qstat

```
asndcy@hpcbastion: ~
asndcy@asaxlogin1:sine_example_pytorch> qstat
Job id      Name      User      Time Use S Queue
-----
1625.asaxpbs1  pytorch_test  asndcy    00:00:00 R gpu

asndcy@asaxlogin1:sine_example_pytorch>
asndcy@asaxlogin1:sine_example_pytorch>
```

Job states

Running	R
Queued (waiting)	Q
Waiting (-a option)	W
Held	H
Exiting	E
Completed	C
Finished	F

If the job is not listed, it is done

13. There may be a longer waiting time if a lot of users are using the GPUs.
14. If you notice something is wrong with your program after submitted the job, you can delete you job by executing:

```
qdel <job id>
```
15. If your job finished execution, it would disappear for the output of `qstat`. Any output of your job will be saved in a file in the current directory named: `<job name>.o<job id>`

For task 1-3:

- You may develop and debug your code using the T4 GPU on Google Colab. You should do performance tests and report the final performance you got using the GPU cluster server (A100 or H100).
- Because of the memory limit on the T4 GPUs on Google Colab, you may choose smaller matrix sizes for debugging.
- For performance test on the GPU cluster server, choose (m=k=n=10000) as the input sizes. *You need to adjust the host memory limit to fit the input matrices when submitting your job to the cluster.*
- Submit task1.cu, task2.cu, and task3.cu (CS 729 only) and a report (Word/PDF) showing the time and performance you got for each task.

Task 1

Task 1 requires that we develop a matrix-matrix multiplication kernel that compute:

$$C = A \cdot B$$

- a) A (input), B (input), and C (output) are matrices
- b) All values are in random single-precision floating point (i.e., float type) in range (0-1)
- c) The dimensions of the matrices are A (m*k), B(k*n), and C (m*n).
- d) Use total m*n threads to parallelly compute total m*n elements in output matrix C.
- e) Try the following thread block sizes (8*8), (16*16), and (32, 32) and select the fastest configuration.
- f) Use CUDA event to time your kernel execution time for input size (m=k=n=10000)
- g) Compute the GFlops performance of your kernel using: $m*k*n*2 / 1e9 / \text{<exec time in second>}$
- h) Your kernel should work with any valid input sizes. This means you need to handle the cases where the dimensions are not the multiply of the thread block sizes.
- i) Implement a CPU version matrix-matrix multiplication function to verify your results
- j) You need to create a new source code file named: task1.cu

Task 2

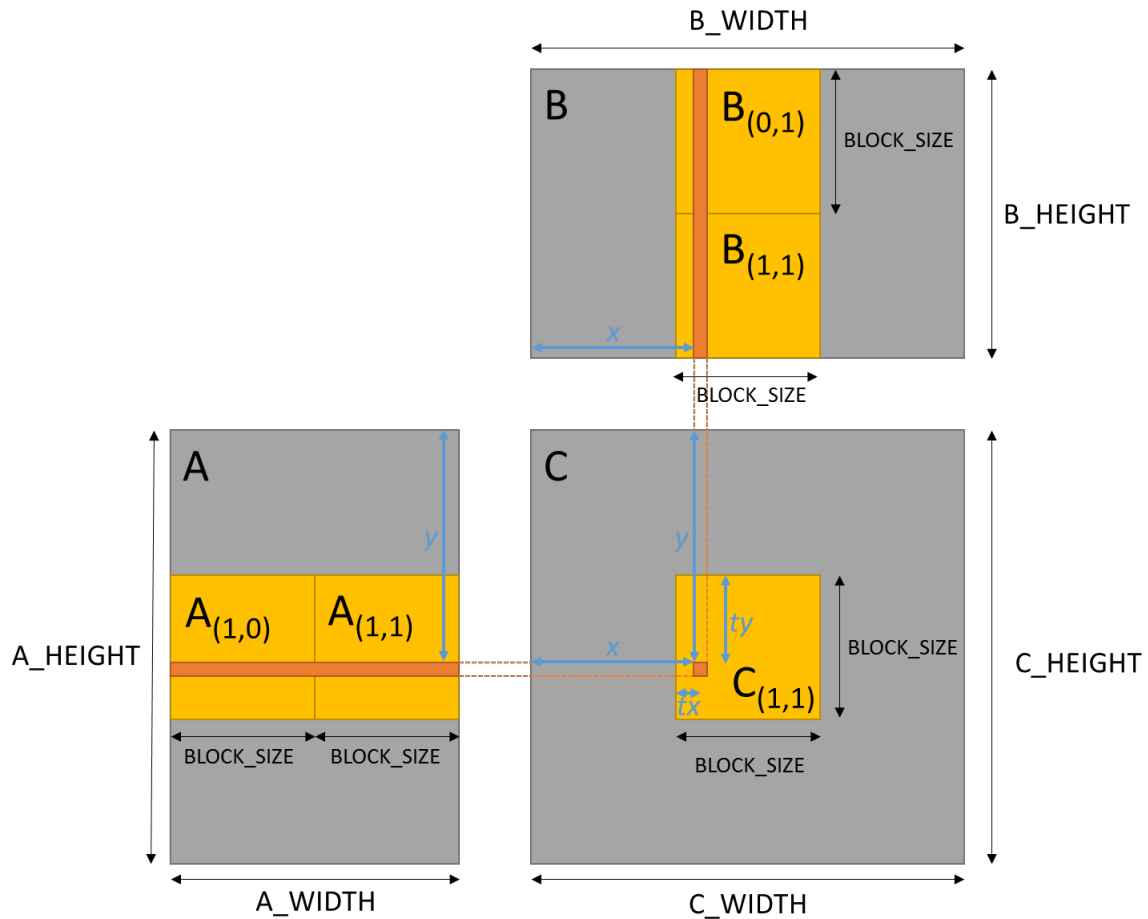


Figure 1 - Blocked matrix multiply (See text for description). Note: indices of sub block are given as (y, x)

For task 2, we are going to optimize the matrix-matrix multiplication kernel using shared memory based on the code of Task 1. You need to create a new source code file named: task2.cu. The implementation is currently very inefficient as it performs $A_WIDTH \times B_HEIGHT$ memory loads to compute each product (value in matrix C). To improve this we will implement a blocked matrix multiply which uses CUDA's shared memory to reduce the number of memory reads by a factor of $BLOCK_SIZE$.

To implement a blocked matrix multiply will require that we load NUM_SUBS square sub matrices of the matrix A and B into shared memory so that we can compute the intermediate result of the sub matrix products. In the example Figure 1 (above) the sub matrix $C_{(1,1)}$ can be calculated by a square thread block of $BLOCK_SIZE$ by $BLOCK_SIZE$ where each thread (with location tx, ty in the square thread block) performs the following steps which requires two stages of loading matrix tiles into shared memory.

1. Load the two sub-matrices $A_{(1,0)}$ and $B_{(0,1)}$ into shared memory. For these sub matrices each thread should
 - a. Load an element of the sub matrix $A_{(1,0)}$ into shared memory from the matrix A at position $(ty + BLOCK_SIZE, tx)$

- b. Each thread should load an element of the sub matrix $B_{(0,1)}$ into shared memory from the matrix B at position $(t_y, t_x + \text{BLOCK_SIZE})$
 - c. Synchronize to ensure all threads have completed loading sub matrix values to shared memory
2. Compute the dot product of each row in sub-matrix $A_{(1,0)}$ with each column in the sub-matrix $B_{(0,1)}$, storing the result in a local variable. Achieved through the following steps.
 - a. Iterate from 0 to BLOCK_SIZE to multiply row t_y of $A_{(1,0)}$ (from shared memory) by column t_x of $B_{(0,1)}$ (from shared memory) to calculate the sub matrix product value.
 - b. Store the sub matrix product value in a thread local variable.
 - c. Synchronize to ensure that all threads have finished computation
3. Repeat steps 1 & 2 for the next sub-matrix (or matrices in the general case), adding each new dot product result to the previous result. For the example in the figure, there is only one more iteration required to load the final 2 sub matrices. E.g.
 - a. Load an element of the sub matrix $A_{(1,1)}$ into shared memory from Matrix A at position $(t_y + \text{BLOCK_SIZE}, t_x + \text{BLOCK_SIZE})$
 - b. Load an element of the sub matrix $B_{(1,1)}$ into shared memory from matrix B at position $(t_y + \text{BLOCK_SIZE}, t_x + \text{BLOCK_SIZE})$
 - c. Synchronize to ensure all threads have completed loading sub matrix values to shared memory
 - d. Iterate from 0 to BLOCK_SIZE to multiply row t_x of $A_{(1,1)}$ (from shared memory) by column t_y of $B_{(1,1)}$ (from shared memory) to calculate the sub matrix product value.
 - e. Add this sub matrix product value to the one calculated for the previous sub matrices.
4. Store the sum of the sub-matrix dot products into global memory at location x, y of Matrix C.
5. Same as Task 1, you should find the best block size and report time and performance you achieved and use CPU matrix-matrix multiplication to verify correctness

Task 3 (CS 729 only)

- Based on the code of Task 2, implement batched matrix-matrix multiplication, which performs a series of individual matrix-matrix multiplication operation with different sets of input matrices that share the same dimensions.
- You need to create a new source code file named: task3.cu.
- For batch size = K, your program should logically does

```
For (int i = 0; i < K; i++) {  
     $C[i] = A[i] \cdot B[i]$   
}
```

- A[i] (input), B[i] (input), and C[i] (input/output) are matrices of batch i.
- The input matrices should be on host memory in the beginning. A[i], B[i], and C[i] are copied to device memory before computation and output matrix C[i] is copied back the host memory after computation.
- Use CUDA concurrent execution feature (i.e., CUDA streams) to build pre-fetch pipeline that overlaps computations with memory copies

```
// pre-fetch inputs for the first batch  
H2D(A[0])  
H2D(B[0])  
For (int i = 0; i < K; i++) {  
    If (i+1 < K) { // pre-fetch inputs for next batch  
        H2D(A[i+1])  
        H2D(B[i+1])  
    }  
     $C[i] = A[i] \cdot B[i]$   
    D2H(C[i])  
}
```

- Report the time and performance of your batched matrix-matrix multiplication with and without pre-fetch pipeline when K=10 and m=k=n=10000.
- Same as Task 1 and 2, you should use your CPU matrix-matrix multiplication to verify correctness.