Hands-on Session – Part II

Compare *dist* on CPU and GPU

- This exercise still uses the Google Colab environment introduced in the 1st hands-on session
  - Using *cuda_Colab_part2.ipynb*
  - *Separate codes can be found in* `dist_gpu.cu`,`dist_cpu.c`
- First compile a CPU version of the 1D dist program introduced in Lecture 2
- Then compile the GPU version
- Run two batches of tests using CPU and GPU versions
- Plot their performance difference

1 compile the CPU version of the 1D dist program

cuda_Colab_part2.ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

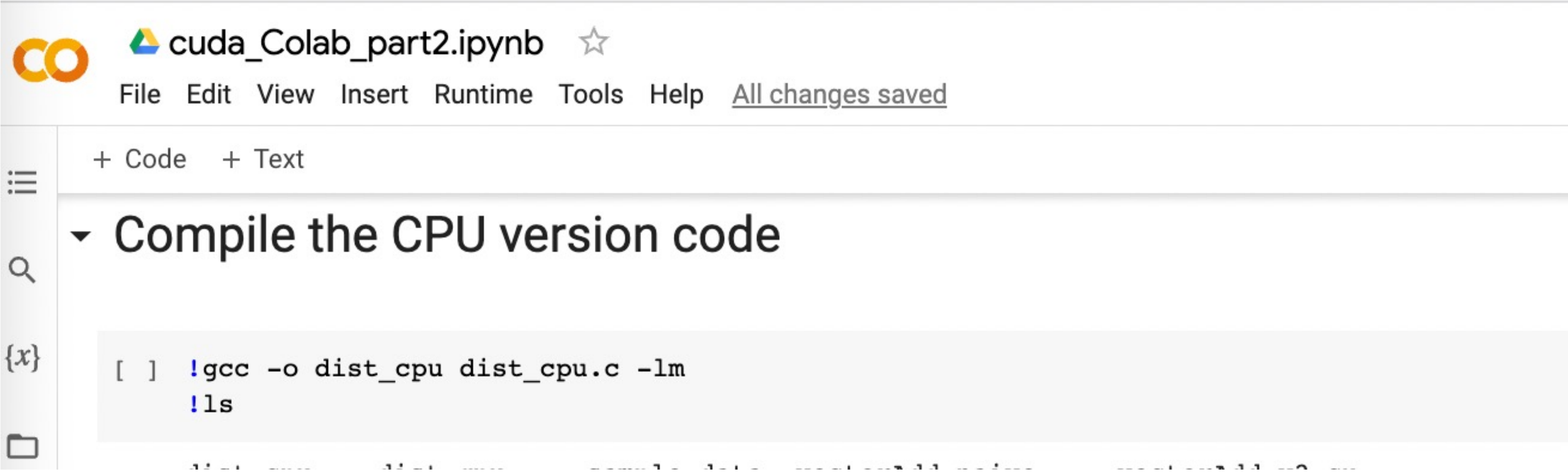
Write the CPU version of the 1D *dist* program in Lecture 2

```
%%writefile dist_cpu.c
#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h> //Include standard math library containing sqrt.

// Timer
double get_cpu_millisecond() {
   struct timeval tp;
   gettimeofday(&tp,NULL);
   return ((double)tp.tv_sec*1.e6 + (double)tp.tv_usec);
}

// A scaling function to convert integers 0,1,...,N-1 to evenly spaced floats
```

1 compile the CPU version of the 1D dist program



2 compile the GPU version of the 1D dist program

cuda_Colab_part2.ipynb

File Edit View Insert Runtime Tools Help

+ Code + Text

Write the GPU version of the 1D *dist* program in Lecture 2

```
%%writefile dist_gpu.cu
#include <sys/time.h>
#include <stdio.h>
#include <math.h> //Include standard math library containing sqrt.

#define TPB 128     // Specify threads per block

// Timer
double get_cpu_millisecond() {
    struct timeval tp;
```

2 compile the GPU version of the 1D dist program

3 run a batch of tests using different input

File Edit View Insert Runtime Tools Help Saving...

+ Code + Text

Run the GPU version code 10 times with increased input length

```
i=0
n=131072
while i<10:
  !./dist_gpu $n
  i=i+1
  n=n*2
```

```
68.00, 104.00, 173.00, 316.00, 600.00, 1154.00, 2265.00, 4494.00, 8959.00, 17919.00,
```

Plot and Compare GPU and CPU performance

cuda_Colab_part2.ipynb

File Edit View Insert Runtime Tools Help All changes saved

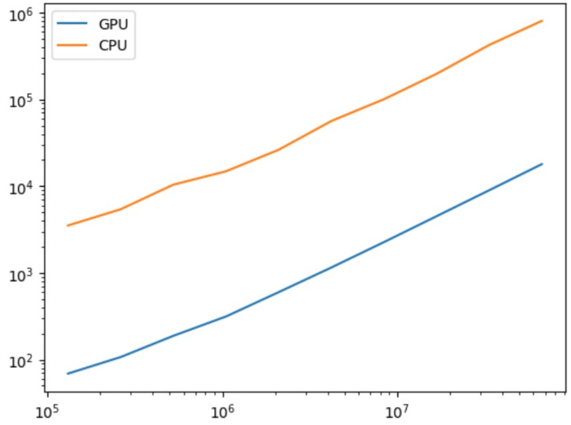
+ Code + Text

Next, plot the timing results from your CPU and GPU versions

```
import matplotlib.pyplot as plt

x = [131072, 262144, 524288, 1048576, 2097152, 4194304, 8388608, 16777216, 33554432, 6
y_gpu = [69.00, 107.00, 188.00, 314.00, 599.00, 1152.00, 2262.00, 4500.00, 8967.00, 17
y_cpu = [3518.00, 5412.00, 10398.00, 14829.00, 26232.00, 56267.00, 100461.00, 197027.0

plt.plot(x, y_gpu, label='GPU');
plt.plot(x, y_cpu, label='CPU');
plt.xscale("log")
plt.yscale("log")
plt.legend(loc="upper left")
plt.show()
```



Optimize a naïve 2D matrix-multiplication

- This exercise still uses the Google Colab environment introduced in the 1st hands-on session
  - Using *cuda_Colab_part2.ipynb*
  - *Separate codes can be found in* `sgemm_naive.cu`
- First compile and run the naïve version on GPU
- Run two batches of tests using CPU and GPU versions
- Plot their performance difference

1 Compile and run the naïve version on GPU

cuda_Colab_part2.ipynb

File Edit View Insert Runtime Tools Help

+ Code + Text

Write the GPU version of a naive 2D matrix multiplication

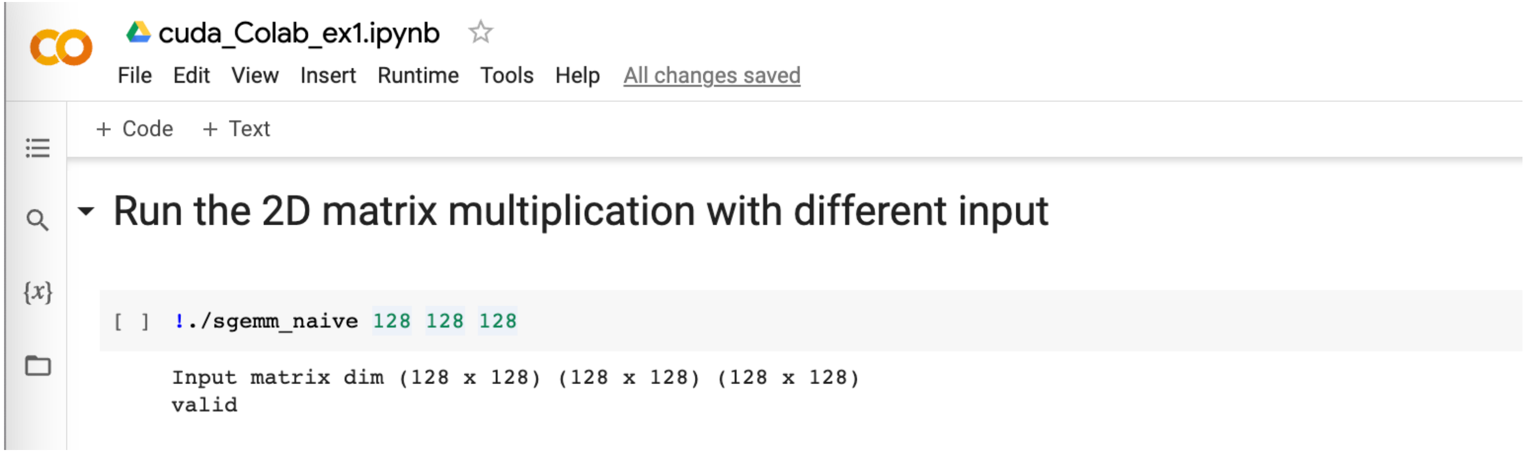
```
%%writefile sgemm_naive.cu

#include <stdio.h>
#include <sys/time.h>

#define DataType double

// Compute C = A * B
// Sgemm stands for single precision general matrix-matrix multiply
__global__ void gemm(DataType *A, DataType *B, DataType *C, int numARows,
```

2 Run the code with different input



3 Profile the code with Nvprof

next, profiling with nvprof

```
!nvprof ./sgemm_naive 128 512 512
```

```
Input matrix dim (128 x 512) (512 x 512) (128 x 512)
==12051== NVPROF is profiling process 12051, command: ./sgemm_naive 128 512 512
valid
==12051== Profiling application: ./sgemm_naive 128 512 512
==12051== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   65.22%  823.02us         1  823.02us  823.02us  823.02us  gemm(double*, double*, do
                   30.86%  389.43us         2  194.72us  45.695us  343.74us  [CUDA memcpy HtoD]
                    3.92%  49.407us         1  49.407us  49.407us  49.407us  [CUDA memcpy DtoH]
      API calls:   98.82%  205.10ms         3  68.366ms  3.5130us  205.02ms  cudaMalloc
                    0.98%  2.0247ms         3  674.91us  132.41us  1.2980ms  cudaMemcpy
                    0.12%  239.84us         3  79.947us  15.321us  118.01us  cudaFree
                    0.05%  112.57us       101  1.1140us     143ns  47.178us  cuDeviceGetAttribute
                    0.01%  28.986us         1  28.986us  28.986us  28.986us  cudaLaunchKernel
                    0.01%  25.681us         1  25.681us  25.681us  25.681us  cuDeviceGetName
                    0.01%  11.400us         1  11.400us  11.400us  11.400us  cuDeviceGetPCIBusId
                    0.00%  2.9870us         2  1.4930us  1.1030us  1.8840us  cuDeviceGet
                    0.00%  2.7060us         3     902ns     228ns  2.1710us  cuDeviceGetCount
                    0.00%     589ns         1     589ns     589ns     589ns  cuDeviceTotalMem
                    0.00%     493ns         1     493ns     493ns     493ns  cuModuleGetLoadingMode
                    0.00%     236ns         1     236ns     236ns     236ns  cuDeviceGetUuid
```

2D Matrix Multiplication – Exercise

For a matrix A of (128x128) and B of (128x128):

- Explain how many CUDA threads and thread blocks are used in your tests.

- Profile your program with nvprof. What are the top 3 activities in time?

2D Matrix Multiplication – Exercise

For a matrix A of (511x1023) and B of (1023x4094):

- Did your program still work? If not, what changes did you make?

- Explain how many CUDA threads and thread blocks you used.

2D Matrix Multiplication – Exercise

Optimize the naïve implementation of with shared memory and tiles

```
__global__ void gemm(DataType *A, DataType *B, DataType *C, int numARows,
                     int numAColumns, int numBRows, int numBColumns){
  //@@ Insert code to implement matrix multiplication here
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  if (row < numARows && col < numBColumns) {
    DataType sum = 0;
    for (int ii = 0; ii < numAColumns; ii++) {
      sum += A[row * numAColumns + ii] * B[ii * numBColumns + col];
    }
    C[row * numBColumns + col] = sum;
  }
}
```

Calculating histogram using atomic operation

- This exercise still uses the Google Colab environment introduced in the 1st hands-on session
  - Using *cuda_Colab_part2.ipynb*
  - *Separate codes can be found in* `Histogram_atomic.cu`
- First compile and run the naïve GPU version
- Run a batch of tests using increased array
- Plot the output histogram

1 Compile the naïve histogram code

cuda_Colab_part2.ipynb

File Edit View Insert Runtime Tools Help Saving...

+ Code + Text

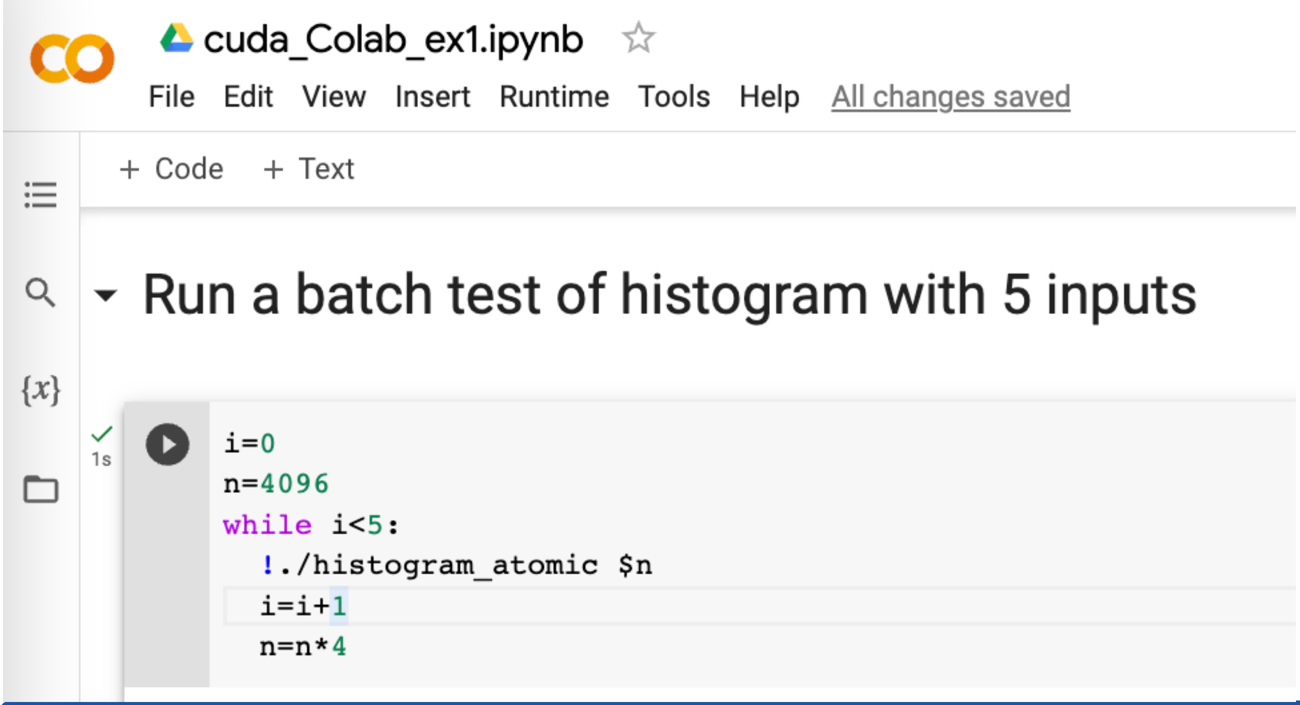
Write a simple CUDA code that calculates histogram of an input array of N elements

```
%%writefile histogram_atomic.cu

#include <stdio.h>
#include <sys/time.h>
#include <random>

#define NUM_BINS 4096
```

Run a batch tests of different N



Plot and Compare the output histogram

cuda_Colab_ex1.ipynb

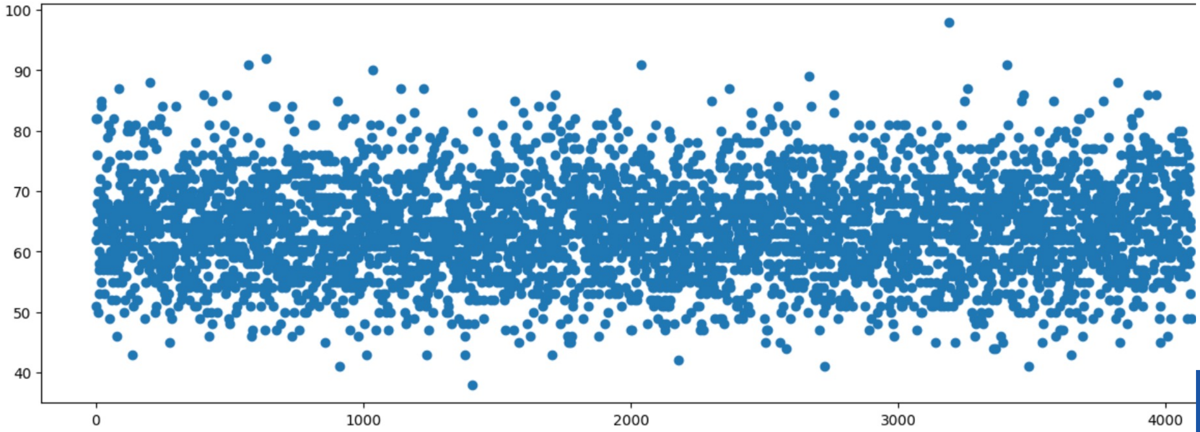
File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

Plot the output histogram

```
import matplotlib.pyplot as plt

NUM_BINS = 4096
y = [...]
plt.rcParams['figure.figsize'] = [15, 5]
plt.plot(y,'o');
plt.show()
```



Histogram – Exercise

For an array of length 1048576:

- How many global reads are performed?

- How many atomic operations are used?

Histogram – Exercise

Optimize the naïve implementation to reduce the number of atomic operations used

```
__global__ void histogram_kernel(unsigned int *input, unsigned int *bins,
                                 unsigned int num_elements,
                                 unsigned int num_bins) {

  unsigned int tid = blockIdx.x * blockDim.x + threadIdx.x;

  // Privatized bins
  extern __shared__ unsigned int bins_s[];
  for (unsigned int binIdx = threadIdx.x; binIdx < num_bins;
       binIdx += blockDim.x) {
    bins_s[binIdx] = 0;
  }
  __syncthreads();

  // Histogram
  for (unsigned int i = tid; i < num_elements; i += blockDim.x * gridDim.x) {
    atomicAdd(&(bins_s[input[i]]), 1);
  }
  __syncthreads();

  // Commit to global memory
  for (unsigned int binIdx = threadIdx.x; binIdx < num_bins;
       binIdx += blockDim.x) {
    atomicAdd(&(bins[binIdx]), bins_s[binIdx]);
  }
}
```

Q & A