20BCE1025_Abhishek_N_N_Lab_11_CUDA_Parallel and Distributed Computing(CSE4001)

November 5, 2022

[1]:
```
!nvcc --version
```

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2021 NVIDIA Corporation
Built on Sun_Feb_14_21:12:58_PST_2021
Cuda compilation tools, release 11.2, V11.2.152
Build cuda_11.2.r11.2/compiler.29618528_0
```

[2]:
```
!nvidia-smi
```

```
Sat Nov  5 15:17:09 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 460.32.03    Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   65C    P0    29W /  70W |      0MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

[3]:
```
!pip install git+https://github.com/andreinechaev/nvcc4jupyter.git
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Collecting git+https://github.com/andreinechaev/nvcc4jupyter.git
```

```
  Cloning https://github.com/andreinechaev/nvcc4jupyter.git to /tmp/pip-req-
build-uulyfjs_
  Running command git clone -q https://github.com/andreinechaev/nvcc4jupyter.git
/tmp/pip-req-build-uulyfjs_
Building wheels for collected packages: NVCCPlugin
  Building wheel for NVCCPlugin (setup.py) … done
  Created wheel for NVCCPlugin: filename=NVCCPlugin-0.0.2-py3-none-any.whl
size=4306
sha256=121337c267896d72b8ae53a2667b8287a5e374ceaad47723d37496a3223ee975
  Stored in directory: /tmp/pip-ephem-wheel-cache-5jgnm9vx/wheels/ca/33/8d/3c86e
b85e97d2b6169d95c6e8f2c297fdec60db6e84cb56f5e
Successfully built NVCCPlugin
Installing collected packages: NVCCPlugin
Successfully installed NVCCPlugin-0.0.2
```

[4]:
```
%load_ext nvcc_plugin
```

```
created output directory at /content/src
Out bin /content/result.out
```

[5]:
```
%%cu
#include<stdio.h>
#include<cuda.h>

int main()
{
    cudaDeviceProp p;
    int device_id;
    int major;
    int minor;


    cudaGetDevice(&device_id);
    cudaGetDeviceProperties(&p,device_id);

    major=p.major;
    minor=p.minor;

    printf("Name of GPU on your system is %s\n",p.name);

    printf("\n Compute Capability of a current GPU on your system is %d.
 ↪%d",major,minor);

    return 0;
}
```

```
Name of GPU on your system is Tesla T4
```

```
Compute Capability of a current GPU on your system is 7.5
```

1 example 1 matrix mul

[6]:
```
%%cu
#include <cassert>
#include <cstddef>
#include <cstdint>
#include <iomanip>
#include <iostream>
#include <random>
#include <stdexcept>
#include <vector>

#define BLOCK_DIM 32

#define checkCuda(val) check((val), #val, __FILE__, __LINE__)
template <typename T>
void check(T err, const char* const func, const char* const file,
           const int line)
{
    if (err != cudaSuccess)
    {
        std::cerr << "CUDA Runtime Error at: " << file << ":" << line
                  << std::endl;
        std::cerr << cudaGetErrorString(err) << " " << func << std::endl;
        std::exit(EXIT_FAILURE);
    }
}

template <typename T>
std::vector<T> create_rand_vector(size_t n)
{
    std::random_device r;
    std::default_random_engine e(r());
    std::uniform_int_distribution<int> uniform_dist(-256, 256);

    std::vector<T> vec(n);
    for (size_t i{0}; i < n; ++i)
    {
        vec.at(i) = static_cast<T>(uniform_dist(e));
    }

    return vec;
}
```

```
// mat_1: m x n
// mat_2: n x p
// mat_3: m x p
template <typename T>
void mm(T const* mat_1, T const* mat_2, T* mat_3, size_t m, size_t n, size_t p)
{
    // Compute the cells in mat_3 sequentially.
    for (size_t i{0}; i < m; ++i)
    {
        for (size_t j{0}; j < p; ++j)
        {
            T acc_sum{0};
            for (size_t k{0}; k < n; ++k)
            {
                acc_sum += mat_1[i * n + k] * mat_2[k * p + j];
            }
            mat_3[i * p + j] = acc_sum;
        }
    }
}

// mat_1: b x m x n
// mat_2: b x n x p
// mat_3: b x m x p
template <typename T>
void bmm(T const* mat_1, T const* mat_2, T* mat_3, size_t b, size_t m, size_t n,
         size_t p)
{
    // Iterate through the batch dimension.
    for (size_t i{0}; i < b; ++i)
    {
        mm(mat_1 + i * (m * n), mat_2 + i * (n * p), mat_3 + i * (m * p), m, n,
           p);
    }
}

template <typename T>
__global__ void mm_kernel(T const* mat_1, T const* mat_2, T* mat_3, size_t m,
                          size_t n, size_t p)
{
    // 2D block and 2D thread
    // Each thread computes one cell in mat_3.
    size_t i{blockIdx.y * blockDim.y + threadIdx.y};
    size_t j{blockIdx.x * blockDim.x + threadIdx.x};

    // Do not process outside the matrix.
    // Do not forget the equal sign!
```

```
    if ((i >= m) || (j >= p))
    {
        return;
    }

    T acc_sum{0};
    for (size_t k{0}; k < n; ++k)
    {
        acc_sum += mat_1[i * n + k] * mat_2[k * p + j];
    }
    mat_3[i * p + j] = acc_sum;
}

// It should be straightforward to extend a kernel to support batching.
template <typename T>
__global__ void bmm_kernel(T const* mat_1, T const* mat_2, T* mat_3, size_t b,
                           size_t m, size_t n, size_t p)
{
    // 2D block and 2D thread
    // Each thread computes one cell in mat_3.
    size_t i{blockIdx.y * blockDim.y + threadIdx.y};
    size_t j{blockIdx.x * blockDim.x + threadIdx.x};

    // Do not process outside the matrix.
    // Do not forget the equal sign!
    if ((i >= m) || (j >= p))
    {
        return;
    }

    // Process the cell of the same index along the batch dimension.
    for (size_t l{0}; l < b; ++l)
    {
        T acc_sum{0};
        for (size_t k{0}; k < n; ++k)
        {
            acc_sum +=
                mat_1[l * m * n + i * n + k] * mat_2[l * n * p + k * p + j];
        }
        mat_3[l * m * p + i * p + j] = acc_sum;
    }
}

template <typename T>
void mm_cuda(T const* mat_1, T const* mat_2, T* mat_3, size_t m, size_t n,
             size_t p)
{
```

```
    dim3 threads_per_block(BLOCK_DIM, BLOCK_DIM);
    dim3 blocks_per_grid(1, 1);
    blocks_per_grid.x = std::ceil(static_cast<double>(p) /
                                  static_cast<double>(threads_per_block.x));
    blocks_per_grid.y = std::ceil(static_cast<double>(m) /
                                  static_cast<double>(threads_per_block.y));
    mm_kernel<<<blocks_per_grid, threads_per_block>>>(mat_1, mat_2, mat_3, m, n,
                                                      p);
}

template <typename T>
void bmm_cuda(T const* mat_1, T const* mat_2, T* mat_3, size_t b, size_t m,
              size_t n, size_t p)
{
    dim3 threads_per_block(BLOCK_DIM, BLOCK_DIM);
    dim3 blocks_per_grid(1, 1);
    blocks_per_grid.x = std::ceil(static_cast<double>(p) /
                                  static_cast<double>(threads_per_block.x));
    blocks_per_grid.y = std::ceil(static_cast<double>(m) /
                                  static_cast<double>(threads_per_block.y));
    bmm_kernel<<<blocks_per_grid, threads_per_block>>>(mat_1, mat_2, mat_3, b,
                                                       m, n, p);
}

template <typename T>
bool allclose(std::vector<T> const& vec_1, std::vector<T> const& vec_2,
              T const& abs_tol)
{
    if (vec_1.size() != vec_2.size())
    {
        return false;
    }
    for (size_t i{0}; i < vec_1.size(); ++i)
    {
        if (std::abs(vec_1.at(i) - vec_2.at(i)) > abs_tol)
        {
            std::cout << vec_1.at(i) << " " << vec_2.at(i) << std::endl;
            return false;
        }
    }
    return true;
}

template <typename T>
bool random_test_mm_cuda(size_t m, size_t n, size_t p)
{
    std::vector<T> const mat_1_vec{create_rand_vector<T>(m * n)};
```

```
    std::vector<T> const mat_2_vec{create_rand_vector<T>(n * p)};
    std::vector<T> mat_3_vec(m * p);
    std::vector<T> mat_4_vec(m * p);
    T const* mat_1{mat_1_vec.data()};
    T const* mat_2{mat_2_vec.data()};
    T* mat_3{mat_3_vec.data()};
    T* mat_4{mat_4_vec.data()};

    mm(mat_1, mat_2, mat_3, m, n, p);

    T *d_mat_1, *d_mat_2, *d_mat_4;

    // Allocate device buffer.
    checkCuda(cudaMalloc(&d_mat_1, sizeof(T) * mat_1_vec.size()));
    checkCuda(cudaMalloc(&d_mat_2, sizeof(T) * mat_2_vec.size()));
    checkCuda(cudaMalloc(&d_mat_4, sizeof(T) * mat_4_vec.size()));

    // Copy data from host to device.
    checkCuda(cudaMemcpy(d_mat_1, mat_1, sizeof(T) * mat_1_vec.size(),
                         cudaMemcpyHostToDevice));
    checkCuda(cudaMemcpy(d_mat_2, mat_2, sizeof(T) * mat_2_vec.size(),
                         cudaMemcpyHostToDevice));

    // Run matrix multiplication on GPU.
    mm_cuda(d_mat_1, d_mat_2, d_mat_4, m, n, p);
    cudaDeviceSynchronize();
    cudaError_t err{cudaGetLastError()};
    if (err != cudaSuccess)
    {
        std::cerr << "CUDA Matrix Multiplication kernel failed to execute."
                  << std::endl;
        std::cerr << cudaGetErrorString(err) << std::endl;
        std::exit(EXIT_FAILURE);
    }

    // Copy data from device to host.
    checkCuda(cudaMemcpy(mat_4, d_mat_4, sizeof(T) * mat_4_vec.size(),
                         cudaMemcpyDeviceToHost));

    // Free device buffer.
    checkCuda(cudaFree(d_mat_1));
    checkCuda(cudaFree(d_mat_2));
    checkCuda(cudaFree(d_mat_4));

    return allclose<T>(mat_3_vec, mat_4_vec, 1e-4);
}
```

```
template <typename T>
bool random_test_bmm_cuda(size_t b, size_t m, size_t n, size_t p)
{
    std::vector<T> const mat_1_vec{create_rand_vector<T>(b * m * n)};
    std::vector<T> const mat_2_vec{create_rand_vector<T>(b * n * p)};
    std::vector<T> mat_3_vec(b * m * p);
    std::vector<T> mat_4_vec(b * m * p);
    T const* mat_1{mat_1_vec.data()};
    T const* mat_2{mat_2_vec.data()};
    T* mat_3{mat_3_vec.data()};
    T* mat_4{mat_4_vec.data()};

    bmm(mat_1, mat_2, mat_3, b, m, n, p);

    T *d_mat_1, *d_mat_2, *d_mat_4;

    // Allocate device buffer.
    checkCuda(cudaMalloc(&d_mat_1, sizeof(T) * mat_1_vec.size()));
    checkCuda(cudaMalloc(&d_mat_2, sizeof(T) * mat_2_vec.size()));
    checkCuda(cudaMalloc(&d_mat_4, sizeof(T) * mat_4_vec.size()));

    // Copy data from host to device.
    checkCuda(cudaMemcpy(d_mat_1, mat_1, sizeof(T) * mat_1_vec.size(),
                         cudaMemcpyHostToDevice));
    checkCuda(cudaMemcpy(d_mat_2, mat_2, sizeof(T) * mat_2_vec.size(),
                         cudaMemcpyHostToDevice));

    // Run matrix multiplication on GPU.
    bmm_cuda(d_mat_1, d_mat_2, d_mat_4, b, m, n, p);
    cudaDeviceSynchronize();
    cudaError_t err{cudaGetLastError()};
    if (err != cudaSuccess)
    {
        std::cerr << "CUDA Matrix Multiplication kernel failed to execute."
                  << std::endl;
        std::cerr << cudaGetErrorString(err) << std::endl;
        std::exit(EXIT_FAILURE);
    }

    // Copy data from device to host.
    checkCuda(cudaMemcpy(mat_4, d_mat_4, sizeof(T) * mat_4_vec.size(),
                         cudaMemcpyDeviceToHost));

    // Free device buffer.
    checkCuda(cudaFree(d_mat_1));
    checkCuda(cudaFree(d_mat_2));
    checkCuda(cudaFree(d_mat_4));
```

```
    return allclose<T>(mat_3_vec, mat_4_vec, 1e-4);
}

template <typename T>
bool random_multiple_test_mm_cuda(size_t num_tests)
{
    std::random_device r;
    std::default_random_engine e(r());
    std::uniform_int_distribution<int> uniform_dist(1, 256);

    size_t m{0}, n{0}, p{0};
    bool success{false};

    for (size_t i{0}; i < num_tests; ++i)
    {
        m = static_cast<size_t>(uniform_dist(e));
        n = static_cast<size_t>(uniform_dist(e));
        p = static_cast<size_t>(uniform_dist(e));
        success = random_test_mm_cuda<T>(m, n, p);
        if (!success)
        {
            return false;
        }
    }

    return true;
}

template <typename T>
bool random_multiple_test_bmm_cuda(size_t num_tests)
{
    std::random_device r;
    std::default_random_engine e(r());
    std::uniform_int_distribution<int> uniform_dist(1, 256);

    size_t b{0}, m{0}, n{0}, p{0};
    bool success{false};

    for (size_t i{0}; i < num_tests; ++i)
    {
        b = static_cast<size_t>(uniform_dist(e));
        m = static_cast<size_t>(uniform_dist(e));
        n = static_cast<size_t>(uniform_dist(e));
        p = static_cast<size_t>(uniform_dist(e));
        success = random_test_bmm_cuda<T>(b, m, n, p);
        if (!success)
```

```
        {
            return false;
        }
    }

    return true;
}

template <typename T>
float measure_latency_mm_cuda(size_t m, size_t n, size_t p, size_t num_tests,
                              size_t num_warmups)
{
    cudaEvent_t startEvent, stopEvent;
    float time{0.0f};

    checkCuda(cudaEventCreate(&startEvent));
    checkCuda(cudaEventCreate(&stopEvent));

    T *d_mat_1, *d_mat_2, *d_mat_4;

    // Allocate device buffer.
    checkCuda(cudaMalloc(&d_mat_1, sizeof(T) * m * n));
    checkCuda(cudaMalloc(&d_mat_2, sizeof(T) * n * p));
    checkCuda(cudaMalloc(&d_mat_4, sizeof(T) * m * p));

    for (size_t i{0}; i < num_warmups; ++i)
    {
        mm_cuda(d_mat_1, d_mat_2, d_mat_4, m, n, p);
    }

    checkCuda(cudaEventRecord(startEvent, 0));
    for (size_t i{0}; i < num_tests; ++i)
    {
        mm_cuda(d_mat_1, d_mat_2, d_mat_4, m, n, p);
    }
    checkCuda(cudaEventRecord(stopEvent, 0));
    checkCuda(cudaEventSynchronize(stopEvent));
    cudaError_t err{cudaGetLastError()};
    if (err != cudaSuccess)
    {
        std::cerr << "CUDA Matrix Multiplication kernel failed to execute."
                  << std::endl;
        std::cerr << cudaGetErrorString(err) << std::endl;
        std::exit(EXIT_FAILURE);
    }
    checkCuda(cudaEventElapsedTime(&time, startEvent, stopEvent));
```

```
    // Free device buffer.
    checkCuda(cudaFree(d_mat_1));
    checkCuda(cudaFree(d_mat_2));
    checkCuda(cudaFree(d_mat_4));

    float latency{time / num_tests};

    return latency;
}

template <typename T>
float measure_latency_bmm_cuda(size_t b, size_t m, size_t n, size_t p,
                               size_t num_tests, size_t num_warmups)
{
    cudaEvent_t startEvent, stopEvent;
    float time{0.0f};

    checkCuda(cudaEventCreate(&startEvent));
    checkCuda(cudaEventCreate(&stopEvent));

    T *d_mat_1, *d_mat_2, *d_mat_4;

    // Allocate device buffer.
    checkCuda(cudaMalloc(&d_mat_1, sizeof(T) * b * m * n));
    checkCuda(cudaMalloc(&d_mat_2, sizeof(T) * b * n * p));
    checkCuda(cudaMalloc(&d_mat_4, sizeof(T) * b * m * p));

    for (size_t i{0}; i < num_warmups; ++i)
    {
        bmm_cuda(d_mat_1, d_mat_2, d_mat_4, b, m, n, p);
    }

    checkCuda(cudaEventRecord(startEvent, 0));
    for (size_t i{0}; i < num_tests; ++i)
    {
        bmm_cuda(d_mat_1, d_mat_2, d_mat_4, b, m, n, p);
    }
    checkCuda(cudaEventRecord(stopEvent, 0));
    checkCuda(cudaEventSynchronize(stopEvent));
    cudaError_t err{cudaGetLastError()};
    if (err != cudaSuccess)
    {
        std::cerr << "CUDA Matrix Multiplication kernel failed to execute."
                  << std::endl;
        std::cerr << cudaGetErrorString(err) << std::endl;
        std::exit(EXIT_FAILURE);
    }
```

```
    checkCuda(cudaEventElapsedTime(&time, startEvent, stopEvent));

    // Free device buffer.
    checkCuda(cudaFree(d_mat_1));
    checkCuda(cudaFree(d_mat_2));
    checkCuda(cudaFree(d_mat_4));

    float latency{time / num_tests};

    return latency;
}

int main()
{
    constexpr size_t num_tests{10};

    assert(random_multiple_test_mm_cuda<int32_t>(num_tests));
    assert(random_multiple_test_mm_cuda<float>(num_tests));
    assert(random_multiple_test_mm_cuda<double>(num_tests));
    assert(random_multiple_test_bmm_cuda<int32_t>(num_tests));
    assert(random_multiple_test_bmm_cuda<float>(num_tests));
    assert(random_multiple_test_bmm_cuda<double>(num_tests));

    constexpr size_t num_measurement_tests{100};
    constexpr size_t num_measurement_warmups{10};
    size_t b{128}, m{1024}, n{1024}, p{1024};

    float mm_cuda_int32_latency{measure_latency_mm_cuda<int32_t>(
        m, n, p, num_measurement_tests, num_measurement_warmups)};
    float mm_cuda_float_latency{measure_latency_mm_cuda<float>(
        m, n, p, num_measurement_tests, num_measurement_warmups)};
    float mm_cuda_double_latency{measure_latency_mm_cuda<double>(
        m, n, p, num_measurement_tests, num_measurement_warmups)};

    float bmm_cuda_int32_latency{measure_latency_bmm_cuda<int32_t>(
        b, m, n, p, num_measurement_tests, num_measurement_warmups)};
    float bmm_cuda_float_latency{measure_latency_bmm_cuda<float>(
        b, m, n, p, num_measurement_tests, num_measurement_warmups)};
    float bmm_cuda_double_latency{measure_latency_bmm_cuda<double>(
        b, m, n, p, num_measurement_tests, num_measurement_warmups)};

    std::cout << "Matrix Multiplication CUDA Latency" << std::endl;
    std::cout << "m: " << m << " "
              << "n: " << n << " "
              << "p: " << p << std::endl;
    std::cout << "INT32: " << std::fixed << std::setprecision(5)
              << mm_cuda_int32_latency << " ms" << std::endl;
```

```
    std::cout << "FLOAT: " << std::fixed << std::setprecision(5)
              << mm_cuda_float_latency << " ms" << std::endl;
    std::cout << "DOUBLE: " << std::fixed << std::setprecision(5)
              << mm_cuda_double_latency << " ms" << std::endl;

    std::cout << "Batched Matrix Multiplication CUDA Latency" << std::endl;
    std::cout << "b: " << b << " "
              << "m: " << m << " "
              << "n: " << n << " "
              << "p: " << p << std::endl;
    std::cout << "INT32: " << std::fixed << std::setprecision(5)
              << bmm_cuda_int32_latency << " ms" << std::endl;
    std::cout << "FLOAT: " << std::fixed << std::setprecision(5)
              << bmm_cuda_float_latency << " ms" << std::endl;
    std::cout << "DOUBLE: " << std::fixed << std::setprecision(5)
              << bmm_cuda_double_latency << " ms" << std::endl;
}
```

```
Matrix Multiplication CUDA Latency
m: 1024 n: 1024 p: 1024
INT32: 4.12928 ms
FLOAT: 3.46946 ms
DOUBLE: 9.34288 ms
Batched Matrix Multiplication CUDA Latency
b: 128 m: 1024 n: 1024 p: 1024
INT32: 512.25275 ms
FLOAT: 508.87042 ms
DOUBLE: 1337.29053 ms
```

2 Example 2

[7]:
```
%%cu
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// CUDA kernel. Each thread takes care of one element of c
__global__ void vecAdd(double *a, double *b, double *c, int n)
{
    // Get our global thread ID
    int id = blockIdx.x*blockDim.x+threadIdx.x;

    // Make sure we do not go out of bounds
    if (id < n)
```

```
        c[id] = a[id] + b[id];
}

int main( int argc, char* argv[] )
{
    // Size of vectors
    int n = 100000;

    // Host input vectors
    double *h_a;
    double *h_b;
    //Host output vector
    double *h_c;

    // Device input vectors
    double *d_a;
    double *d_b;
    //Device output vector
    double *d_c;

    // Size, in bytes, of each vector
    size_t bytes = n*sizeof(double);

    // Allocate memory for each vector on host
    h_a = (double*)malloc(bytes);
    h_b = (double*)malloc(bytes);
    h_c = (double*)malloc(bytes);

    // Allocate memory for each vector on GPU
    cudaMalloc(&d_a, bytes);
    cudaMalloc(&d_b, bytes);
    cudaMalloc(&d_c, bytes);

    int i;
    // Initialize vectors on host
    for( i = 0; i < n; i++ ) {
        h_a[i] = sin(i)*sin(i);
        h_b[i] = cos(i)*cos(i);
    }

    // Copy host vectors to device
    cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy( d_b, h_b, bytes, cudaMemcpyHostToDevice);

    int blockSize, gridSize;

    // Number of threads in each thread block
```

```
    blockSize = 1024;

    // Number of thread blocks in grid
    gridSize = (int)ceil((float)n/blockSize);

    // Execute the kernel
    vecAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);

    // Copy array back to host
    cudaMemcpy( h_c, d_c, bytes, cudaMemcpyDeviceToHost );

    // Sum up vector c and print result divided by n, this should equal 1␣
↪within error
    double sum = 0;
    for(i=0; i<n; i++)
        sum += h_c[i];
    printf("final result: %f\n", sum/n);

    // Release device memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    // Release host memory
    free(h_a);
    free(h_b);
    free(h_c);

    return 0;
}
```

```
final result: 1.000000
```

[ ]:

```
!apt-get install texlive texlive-xetex texlive-latex-extra pandoc
!pip install pypandoc
from google.colab import drive
drive.mount('/content/drive')
!cp "/content/drive/MyDrive/5th_fall_sem_22_23/cse4001_pdc_l/
↪20BCE1025_Abhishek_N_N_Lab_11_CUDA_Parallel and Distributed␣
↪Computing(CSE4001).ipynb" ./
!jupyter nbconvert --to PDF "20BCE1025_Abhishek_N_N_Lab_11_CUDA_Parallel and␣
↪Distributed Computing(CSE4001).ipynb"
```

```
Reading package lists… Done
Building dependency tree
Reading state information… Done
```

```
pandoc is already the newest version (1.19.2.4~dfsg-1build4).
texlive is already the newest version (2017.20180305-1).
texlive-latex-extra is already the newest version (2017.20180305-2).
texlive-xetex is already the newest version (2017.20180305-1).
The following package was automatically installed and is no longer required:
  libnvidia-common-460
Use 'apt autoremove' to remove it.
0 upgraded, 0 newly installed, 0 to remove and 5 not upgraded.
```