```
!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-52_yj781
  Running command git clone --filter=blob:none --quiet https://github.com/andreinechaev/nvcc4jupyter.git /tmp/pip-req-build-52_yj781
  Resolved https://github.com/andreinechaev/nvcc4jupyter.git to commit aac710a35f52bb78ab34d2e52517237941399eff
  Preparing metadata (setup.py) ... done
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=4305 sha256=cb139076a076c203ff73e44e4a1e24b60ca92eb7551
  Stored in directory: /tmp/pip-ephem-wheel-cache-o4q78u9x/wheels/db/c1/1f/a2bb07bbb4a1ce3c43921252aeafaa6205f08637e292496f04
Successfully built NVCCPlugin
Installing collected packages: NVCCPlugin
Successfully installed NVCCPlugin-0.0.2
```

```
%load_ext nvcc_plugin
```

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

```
%%cu
/*
 *  file name: matrix.cu
 *
 *  matrix.cu contains the code that realize some common used matrix operations in CUDA
 *
 *  this is a toy program for learning CUDA, some functions are reusable in other project
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#define BLOCK_SIZE 16

/*
*********************************************************************
function name: gpu_matrix_mult
description: dot product of two matrix (not only square)
parameters:
            &a GPU device pointer to a m X n matrix (A)
            &b GPU device pointer to a n X k matrix (B)
            &c GPU device output purpose pointer to a m X k matrix (C)
            to store the result
Note:
    grid and block should be configured as:
        dim3 dimGrid((k + BLOCK_SIZE - 1) / BLOCK_SIZE, (m + BLOCK_SIZE - 1) / BLOCK_SIZE);
        dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    further sppedup can be obtained by using shared memory to decrease global memory access times
return: none
*********************************************************************
*/
__global__ void gpu_matrix_mult(int *a,int *b, int *c, int m, int n, int k)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int sum = 0;
    if( col < k && row < m)
    {
        for(int i = 0; i < n; i++)
        {
            sum += a[row * n + i] * b[i * k + col];
        }
        c[row * k + col] = sum;
    }
}

/*
*********************************************************************
function name: gpu_square_matrix_mult
description: dot product of two matrix (not only square) in GPU
parameters:
            &a GPU device pointer to a n X n matrix (A)
            &b GPU device pointer to a n X n matrix (B)
```

```
            &c GPU device output purpose pointer to a n X n matrix (C)
            to store the result
Note:
    grid and block should be configured as:
        dim3 dim_grid((n - 1) / BLOCK_SIZE + 1, (n - 1) / BLOCK_SIZE + 1, 1);
        dim3 dim_block(BLOCK_SIZE, BLOCK_SIZE, 1);
return: none
*********************************************************************
*/
__global__ void gpu_square_matrix_mult(int *d_a, int *d_b, int *d_result, int n)
{
    __shared__ int tile_a[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ int tile_b[BLOCK_SIZE][BLOCK_SIZE];

    int row = blockIdx.y * BLOCK_SIZE + threadIdx.y;
    int col = blockIdx.x * BLOCK_SIZE + threadIdx.x;
    int tmp = 0;
    int idx;

    for (int sub = 0; sub < gridDim.x; ++sub)
    {
        idx = row * n + sub * BLOCK_SIZE + threadIdx.x;
        if(idx >= n*n)
        {
            // n may not divisible by BLOCK_SIZE
            tile_a[threadIdx.y][threadIdx.x] = 0;
        }
        else
        {
            tile_a[threadIdx.y][threadIdx.x] = d_a[idx];
        }

        idx = (sub * BLOCK_SIZE + threadIdx.y) * n + col;
        if(idx >= n*n)
        {
            tile_b[threadIdx.y][threadIdx.x] = 0;
        }
        else
        {
            tile_b[threadIdx.y][threadIdx.x] = d_b[idx];
        }
        __syncthreads();

        for (int k = 0; k < BLOCK_SIZE; ++k)
        {
            tmp += tile_a[threadIdx.y][k] * tile_b[k][threadIdx.x];
        }
        __syncthreads();
    }
    if(row < n && col < n)
    {
        d_result[row * n + col] = tmp;
    }
}

/*
*********************************************************************
function name: gpu_matrix_transpose
description: matrix transpose
parameters:
            &mat_in GPU device pointer to a rows X cols matrix
            &mat_out GPU device output purpose pointer to a cols X rows matrix
            to store the result
Note:
    grid and block should be configured as:
        dim3 dim_grid((n - 1) / BLOCK_SIZE + 1, (n - 1) / BLOCK_SIZE + 1, 1);
        dim3 dim_block(BLOCK_SIZE, BLOCK_SIZE, 1);
return: none
*********************************************************************
*/
__global__ void gpu_matrix_transpose(int* mat_in, int* mat_out, unsigned int rows, unsigned int cols)
{
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int idy = blockIdx.y * blockDim.y + threadIdx.y;

    if (idx < cols && idy < rows)
    {
        unsigned int pos = idy * cols + idx;
```

```
        unsigned int pos = idy * cols + idx;
        unsigned int trans_pos = idx * rows + idy;
        mat_out[trans_pos] = mat_in[pos];
    }
}
/*
*********************************************************************
function name: cpu_matrix_mult
description: dot product of two matrix (not only square) in CPU,
             for validating GPU results
parameters:
            &a CPU host pointer to a m X n matrix (A)
            &b CPU host pointer to a n X k matrix (B)
            &c CPU host output purpose pointer to a m X k matrix (C)
            to store the result
return: none
*********************************************************************
*/
void cpu_matrix_mult(int *h_a, int *h_b, int *h_result, int m, int n, int k) {
    for (int i = 0; i < m; ++i)
    {
        for (int j = 0; j < k; ++j)
        {
            int tmp = 0.0;
            for (int h = 0; h < n; ++h)
            {
                tmp += h_a[i * n + h] * h_b[h * k + j];
            }
            h_result[i * k + j] = tmp;
        }
    }
}

/*
*********************************************************************
function name: main
description: test and compare
parameters:
            none
return: none
*********************************************************************
*/
int main(int argc, char const *argv[])
{
    int m, n, k;
    /* Fixed seed for illustration */
    srand(3333);
    printf("please type in m n and k\n");
    scanf("%d %d %d", &m, &n, &k);

    // allocate memory in host RAM, h_cc is used to store CPU result
    int *h_a, *h_b, *h_c, *h_cc;
    cudaMallocHost((void **) &h_a, sizeof(int)*m*n);
    cudaMallocHost((void **) &h_b, sizeof(int)*n*k);
    cudaMallocHost((void **) &h_c, sizeof(int)*m*k);
    cudaMallocHost((void **) &h_cc, sizeof(int)*m*k);

    // random initialize matrix A
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            h_a[i * n + j] = rand() % 1024;
        }
    }

    // random initialize matrix B
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < k; ++j) {
            h_b[i * k + j] = rand() % 1024;
        }
    }

    float gpu_elapsed_time_ms, cpu_elapsed_time_ms;

    // some events to count the execution time
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
```

```
    // start to count execution time of GPU version
    cudaEventRecord(start, 0);
    // Allocate memory space on the device
    int *d_a, *d_b, *d_c;
    cudaMalloc((void **) &d_a, sizeof(int)*m*n);
    cudaMalloc((void **) &d_b, sizeof(int)*n*k);
    cudaMalloc((void **) &d_c, sizeof(int)*m*k);

    // copy matrix A and B from host to device memory
    cudaMemcpy(d_a, h_a, sizeof(int)*m*n, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, sizeof(int)*n*k, cudaMemcpyHostToDevice);

    unsigned int grid_rows = (m + BLOCK_SIZE - 1) / BLOCK_SIZE;
    unsigned int grid_cols = (k + BLOCK_SIZE - 1) / BLOCK_SIZE;
    dim3 dimGrid(grid_cols, grid_rows);
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);

    // Launch kernel
    if(m == n && n == k)
    {
        gpu_square_matrix_mult<<<dimGrid, dimBlock>>>(d_a, d_b, d_c, n);
    }
    else
    {
        gpu_matrix_mult<<<dimGrid, dimBlock>>>(d_a, d_b, d_c, m, n, k);
    }
    // Transefr results from device to host
    cudaMemcpy(h_c, d_c, sizeof(int)*m*k, cudaMemcpyDeviceToHost);
    cudaThreadSynchronize();
    // time counting terminate
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);

    // compute time elapse on GPU computing
    cudaEventElapsedTime(&gpu_elapsed_time_ms, start, stop);
    printf("Time elapsed on matrix multiplication of %dx%d . %dx%d on GPU: %f ms.\n\n", m, n, n, k, gpu_elapsed_time_ms);

    // start the CPU version
    cudaEventRecord(start, 0);

    cpu_matrix_mult(h_a, h_b, h_cc, m, n, k);

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&cpu_elapsed_time_ms, start, stop);
    printf("Time elapsed on matrix multiplication of %dx%d . %dx%d on CPU: %f ms.\n\n", m, n, n, k, cpu_elapsed_time_ms);

    // validate results computed by GPU
    int all_ok = 1;
    for (int i = 0; i < m; ++i)
    {
        for (int j = 0; j < k; ++j)
        {
            //printf("[%d][%d]:%d == [%d][%d]:%d, ", i, j, h_cc[i*k + j], i, j, h_c[i*k + j]);
            if(h_cc[i*k + j] != h_c[i*k + j])
            {
                all_ok = 0;
            }
        }
        //printf("\n");
    }

    // roughly compute speedup
    if(all_ok)
    {
        printf("all results are correct!!!, speedup = %f\n", cpu_elapsed_time_ms / gpu_elapsed_time_ms);
    }
    else
    {
        printf("incorrect results\n");
    }

    // free memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
    cudaFreeHost(h_a);
```

```
    cudaFreeHost(h_b);
    cudaFreeHost(h_c);
    cudaFreeHost(h_cc);
    return 0;
}
```

```
please type in m n and k
Time elapsed on matrix multiplication of -1638093008x21908 . 21908x-1666910242 on GPU: 1.264448 ms.

Time elapsed on matrix multiplication of -1638093008x21908 . 21908x-1666910242 on CPU: 0.002208 ms.

all results are correct!!!, speedup = 0.001746
```


✓ 1s completed at 5:47 PM