# four-b

April 9, 2025

[ ]:
```
# This Python 3 environment comes with many helpful analytics libraries␣
↪installed
# It is defined by the kaggle/python Docker image: https://github.com/kaggle/
↪docker-python
# For example, here's several helpful packages to load

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

# Input data files are available in the read-only "../input/" directory
# For example, running this (by clicking run or pressing Shift+Enter) will list␣
↪all files under the input directory

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

# You can write up to 20GB to the current directory (/kaggle/working/) that␣
↪gets preserved as output when you create a version using "Save & Run All"
# You can also write temporary files to /kaggle/temp/, but they won't be saved␣
↪outside of the current session
```

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

```
Wed Apr  9 13:52:11 2025
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 560.35.03              Driver Version: 560.35.03      CUDA Version: 12.6     |
|-----------------------------------------+------------------------+----------------------+
| 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   47C    P8              11W /   70W |       1MiB /  15360MiB |      0%
Default |
|                                         |                        |
N/A |
+-----------------------------------------+------------------------+------------
----------+
|   1  Tesla T4                       Off |   00000000:00:05.0 Off |
0 |
| N/A   52C    P8              12W /   70W |       1MiB /  15360MiB |      0%
Default |
|                                         |                        |
N/A |
+-----------------------------------------+------------------------+------------
----------+

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

[3]:
```
!nvcc --version
```

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2023 NVIDIA Corporation
Built on Tue_Aug_15_22:02:13_PDT_2023
Cuda compilation tools, release 12.2, V12.2.140
Build cuda_12.2.r12.2/compiler.33191640_0
```

[4]:
```
%%writefile matrix_mul_refined.cu
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <math.h> // For fabs
```

```
// Simple CUDA Error Handling Macro
#define CHECK_CUDA_ERROR(err) \
    if (err != cudaSuccess) { \
        fprintf(stderr, "CUDA Error at %s:%d: %s\n", __FILE__, __LINE__, cudaGetErrorString(err)); \
        exit(EXIT_FAILURE); \
    }

// Tile dimension (adjust based on GPU architecture, e.g., 16 or 32)
#define TILE_DIM 16

// Tiled Matrix Multiplication Kernel (C = A * B)
__global__ void matrixMulTiledKernel(const float *A, const float *B, float *C, int width) {
    __shared__ float ds_A[TILE_DIM][TILE_DIM];
    __shared__ float ds_B[TILE_DIM][TILE_DIM];

    int tx = threadIdx.x;   // Thread index within block (col)
    int ty = threadIdx.y;   // Thread index within block (row)
    int row = blockIdx.y * TILE_DIM + ty; // Global row index for C
    int col = blockIdx.x * TILE_DIM + tx; // Global col index for C

    float Cvalue = 0.0f;
    int numTiles = (width + TILE_DIM - 1) / TILE_DIM;

    // Loop over tiles
    for (int t = 0; t < numTiles; ++t) {
        // Load tile for A from global memory to shared memory
        int A_row_idx = row;
        int A_col_idx = t * TILE_DIM + tx;
        if (A_row_idx < width && A_col_idx < width) {
            ds_A[ty][tx] = A[A_row_idx * width + A_col_idx];
        } else {
            ds_A[ty][tx] = 0.0f;
        }

        // Load tile for B from global memory to shared memory
        int B_row_idx = t * TILE_DIM + ty;
        int B_col_idx = col;
        if (B_row_idx < width && B_col_idx < width) {
            ds_B[ty][tx] = B[B_row_idx * width + B_col_idx];
        } else {
            ds_B[ty][tx] = 0.0f;
        }

        __syncthreads(); // Ensure tiles are loaded before computation
```

```
        // Multiply tiles loaded in shared memory
        for (int k = 0; k < TILE_DIM; ++k) {
            // Check if the k-th element corresponds to a valid element
            // in the original matrices (handles non-perfect divisibility)
            // Note: The padding above largely handles this, but this is␣
↪belt-and-suspenders.
            if (t * TILE_DIM + k < width) {
               Cvalue += ds_A[ty][k] * ds_B[k][tx];
            }
        }

        __syncthreads(); // Ensure computation is done before loading next tile
    }

    // Write result Cvalue to global memory
    if (row < width && col < width) {
        C[row * width + col] = Cvalue;
    }
}

// Basic CPU Matrix Multiplication for Verification
void matrixMulCPU(const float *A, const float *B, float *C, int width) {
    for (int row = 0; row < width; ++row) {
        for (int col = 0; col < width; ++col) {
            float sum = 0.0f;
            for (int k = 0; k < width; ++k) {
                sum += A[row * width + k] * B[k * width + col];
            }
            C[row * width + col] = sum;
        }
    }
}

int main() {
    // Use a multiple of TILE_DIM for simplicity, but kernel handles others
    int width = 1024;
    printf("Matrix Multiplication (Tiled CUDA)\nMatrix dimensions: %d x %d\n",␣
↪width, width);
    printf("Tile dimensions: %d x %d\n", TILE_DIM, TILE_DIM);

    size_t size = (size_t)width * width * sizeof(float);
    float mb_size = (float)size / (1024 * 1024);

    // Host memory
    float *h_A      = (float*)malloc(size);
    float *h_B      = (float*)malloc(size);
```

```
    float *h_C_gpu  = (float*)malloc(size);
    float *h_C_cpu  = NULL; // Allocate only if verifying
    bool verify = true;     // Set to false to skip CPU verification

    if (!h_A || !h_B || !h_C_gpu) {
        fprintf(stderr, "Failed to allocate host matrices!\n"); return EXIT_FAILURE;
    }
    if (verify) {
        h_C_cpu = (float*)malloc(size);
        if (!h_C_cpu) { fprintf(stderr, "Failed to allocate host CPU result matrix!\n"); verify = false; }
    }

    // Initialize host matrices (simple values)
    printf("Initializing host matrices (%.2f MB each)...\n", mb_size);
    for (int i = 0; i < width * width; ++i) {
        h_A[i] = (float)(i % width + 1) / width; // Some pattern
        h_B[i] = (float)(i / width + 1) / width; // Some pattern
    }

    // Device memory
    float *d_A = NULL, *d_B = NULL, *d_C = NULL;
    printf("Allocating %.2f MB on device...\n", 3.0f * mb_size);
    CHECK_CUDA_ERROR(cudaMalloc(&d_A, size));
    CHECK_CUDA_ERROR(cudaMalloc(&d_B, size));
    CHECK_CUDA_ERROR(cudaMalloc(&d_C, size));

    // Copy data Host -> Device
    printf("Copying data to device...\n");
    CHECK_CUDA_ERROR(cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice));
    CHECK_CUDA_ERROR(cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice));

    // Kernel launch configuration (2D grid, 2D block)
    dim3 blockSize(TILE_DIM, TILE_DIM);
    dim3 gridSize((width + TILE_DIM - 1) / TILE_DIM, (width + TILE_DIM - 1) / TILE_DIM);
    printf("Launching kernel (Grid: %dx%d, Block: %dx%d)...\n",
           gridSize.x, gridSize.y, blockSize.x, blockSize.y);

    // --- Use CUDA events for timing (more accurate than CPU timers) ---
    cudaEvent_t start, stop;
    CHECK_CUDA_ERROR(cudaEventCreate(&start));
    CHECK_CUDA_ERROR(cudaEventCreate(&stop));

    CHECK_CUDA_ERROR(cudaEventRecord(start)); // Record start time
```

```
    // Launch kernel
    matrixMulTiledKernel<<<gridSize, blockSize>>>(d_A, d_B, d_C, width);

    CHECK_CUDA_ERROR(cudaEventRecord(stop));    // Record stop time
    CHECK_CUDA_ERROR(cudaEventSynchronize(stop));// Wait for the event to␣
↪complete

    CHECK_CUDA_ERROR(cudaPeekAtLastError()); // Check for launch errors
    // cudaDeviceSynchronize() is implicitly done by cudaEventSynchronize(stop)␣
↪above

    float milliseconds = 0;
    CHECK_CUDA_ERROR(cudaEventElapsedTime(&milliseconds, start, stop));
    printf("Kernel execution time: %.3f ms\n", milliseconds);
    double gflops = 2.0 * width * width * width / (milliseconds / 1000.0) / 1e9;
    printf("Performance: %.2f GFLOPS\n", gflops);

    // Copy data Device -> Host
    printf("Copying result back to host...\n");
    CHECK_CUDA_ERROR(cudaMemcpy(h_C_gpu, d_C, size, cudaMemcpyDeviceToHost));

    // Verification
    if (verify) {
        printf("Performing CPU verification...\n");
        matrixMulCPU(h_A, h_B, h_C_cpu, width);
        printf("Comparing results...\n");

        double max_error = 0.0;
        double avg_error = 0.0;
        int errors = 0;
        float tolerance = 1e-3f; // Adjust tolerance as needed

        for (int i = 0; i < width * width; ++i) {
            double error = fabs(h_C_gpu[i] - h_C_cpu[i]);
            if (error > max_error) max_error = error;
            avg_error += error;
            if (error > tolerance) {
                errors++;
            }
        }
        avg_error /= (width * width);

        if (errors == 0) {
            printf("Verification Successful! Max Error: %e, Avg Error: %e\n",␣
↪max_error, avg_error);
        } else {
```

```
            printf("Verification FAILED! Errors: %d, Max Error: %e, Avg Error:␣
↪%e\n", errors, max_error, avg_error);
        }
    }

    // Cleanup
    printf("Freeing memory...\n");
    CHECK_CUDA_ERROR(cudaEventDestroy(start));
    CHECK_CUDA_ERROR(cudaEventDestroy(stop));
    CHECK_CUDA_ERROR(cudaFree(d_A));
    CHECK_CUDA_ERROR(cudaFree(d_B));
    CHECK_CUDA_ERROR(cudaFree(d_C));
    free(h_A);
    free(h_B);
    free(h_C_gpu);
    if (h_C_cpu) free(h_C_cpu);

    printf("Matrix multiplication complete.\n");
    return EXIT_SUCCESS;
}
```

```
Writing matrix_mul_refined.cu
```

[5]:
```
!nvcc matrix_mul_refined.cu -o matrix_mul_refined
```

[6]:
```
!./matrix_mul_refined
```

```
Matrix Multiplication (Tiled CUDA)
Matrix dimensions: 1024 x 1024
Tile dimensions: 16 x 16
Initializing host matrices (4.00 MB each)…
Allocating 12.00 MB on device…
Copying data to device…
Launching kernel (Grid: 64x64, Block: 16x16)…
Kernel execution time: 141.489 ms
Performance: 15.18 GFLOPS
Copying result back to host…
Performing CPU verification…
Comparing results…
Verification Successful! Max Error: 0.000000e+00, Avg Error: 0.000000e+00
Freeing memory…
Matrix multiplication complete.
```