PCA-EXP-6-MATRIX-TRANSPOSITION-USING-SHARED-MEMORY-AY-23-24

Dharani Elango

212221230021

EX. NO:6

DATE:06/05/2024

MATRIX TRANSPOSITION USING SHARED MEMORY

Implement Matrix transposition using GPU Shared memory.

AIM:

To perform Matrix Multiplication using Transposition using shared memory.

EQUIPMENTS REQUIRED:

Hardware – PCs with NVIDIA GPU & CUDA NVCC Google Colab with NVCC Compiler

PROCEDURE:

CUDA_SharedMemory_AccessPatterns:

1. Begin Device Setup 1.1 Select the device to be used for computation 1.2 Retrieve the properties of the selected device
2. End Device Setup

3. Begin Array Size Setup 3.1 Set the size of the array to be used in the computation 3.2 The array size is determined by the block dimensions (BDIMX and BDIMY)
4. End Array Size Setup
5. Begin Execution Configuration 5.1 Set up the execution configuration with a grid and block dimensions 5.2 In this case, a single block grid is used
6. End Execution Configuration
7. Begin Memory Allocation 7.1 Allocate device memory for the output array d_C 7.2 Allocate a corresponding array gpuRef in the host memory
8. End Memory Allocation
9. Begin Kernel Execution 9.1 Launch several kernel functions with different shared memory access patterns (Use any two patterns) 9.1.1 setRowReadRow: Each thread writes to and reads from its row in shared memory 9.1.2 setColReadCol: Each thread writes to and reads from its column in shared memory 9.1.3 setColReadCol2: Similar to setColReadCol, but with transposed coordinates 9.1.4 setRowReadCol: Each thread writes to its row and reads from its column in shared memory 9.1.5 setRowReadColDyn: Similar to setRowReadCol, but with dynamic shared memory allocation 9.1.6 setRowReadColPad: Similar to setRowReadCol, but with padding to avoid bank conflicts 9.1.7 setRowReadColDynPad: Similar to setRowReadColPad, but with dynamic shared memory allocation
10. End Kernel Execution
11. Begin Memory Copy 11.1 After each kernel execution, copy the output array from device memory to host memory
12. End Memory Copy
13. Begin Memory Free 13.1 Free the device memory and host memory
14. End Memory Free
15. Reset the device
16. End of Algorithm

PROGRAM:

```
!pip install git+https://github.com/andreinechaev/nvcc4jupyter.git
%load_ext nvcc4jupyter

%%cuda
#include <stdio.h>
#include <cuda_runtime.h>
#include <cuda.h>
#include <sys/time.h>

#ifndef _COMMON_H
#define _COMMON_H

#define CHECK(call)
{
    const cudaError_t error = call;
    if (error != cudaSuccess)
    {
        fprintf(stderr, "Error: %s:%d, ", __FILE__, __LINE__);
        fprintf(stderr, "code: %d, reason: %s\n", error,
                cudaGetErrorString(error));
        exit(1);
    }
}

#define CHECK_CUBLAS(call)
{
    cublasStatus_t err;
    if ((err = (call)) != CUBLAS_STATUS_SUCCESS)
    {
        fprintf(stderr, "Got CUBLAS error %d at %s:%d\n", err, __FILE__,
                __LINE__);
        exit(1);
    }
}

#define CHECK_CURAND(call)
{
    curandStatus_t err;
    if ((err = (call)) != CURAND_STATUS_SUCCESS)
    {
        fprintf(stderr, "Got CURAND error %d at %s:%d\n", err, __FILE__,
                __LINE__);
        exit(1);
    }
}

#define CHECK_CUFFT(call)
{
```

```
    cufftResult err;
    if ( (err = (call)) != CUFFT_SUCCESS)
    {
        fprintf(stderr, "Got CUFFT error %d at %s:%d\n", err, __FILE__,
                __LINE__);
        exit(1);
    }
}

#define CHECK_CUSPARSE(call)
{
    cusparseStatus_t err;
    if ((err = (call)) != CUSPARSE_STATUS_SUCCESS)
    {
        fprintf(stderr, "Got error %d at %s:%d\n", err, __FILE__, __LINE__);
        cudaError_t cuda_err = cudaGetLastError();
        if (cuda_err != cudaSuccess)
        {
            fprintf(stderr, "  CUDA error \"%s\" also detected\n",
                    cudaGetErrorString(cuda_err));
        }
        exit(1);
    }
}

inline double seconds()
{
    struct timeval tp;
    struct timezone tzp;
    int i = gettimeofday(&tp, &tzp);
    return ((double)tp.tv_sec + (double)tp.tv_usec * 1.e-6);
}

#endif // _COMMON_H

#define BDIMX 16
#define BDIMY 16
#define IPAD  2
void printData(char *msg, int *in,  const int size)
{
    printf("%s: ", msg);

    for (int i = 0; i < size; i++)
    {
        printf("%4d", in[i]);
        fflush(stdout);
    }

    printf("\n\n");
```

```
}

__global__ void setRowReadRow(int *out)
{
    // static shared memory
    __shared__ int tile[BDIMY][BDIMX];

    // mapping from thread index to global memory index
    unsigned int idx = threadIdx.y * blockDim.x + threadIdx.x;

    // shared memory store operation
    tile[threadIdx.y][threadIdx.x] = idx;

    // wait for all threads to complete
    __syncthreads();

    // shared memory load operation
    out[idx] = tile[threadIdx.y][threadIdx.x] ;
}

__global__ void setColReadCol(int *out)
{
    // static shared memory
    __shared__ int tile[BDIMX][BDIMY];

    // mapping from thread index to global memory index
    unsigned int idx = threadIdx.y * blockDim.x + threadIdx.x;

    // shared memory store operation
    tile[threadIdx.x][threadIdx.y] = idx;

    // wait for all threads to complete
    __syncthreads();

    // shared memory load operation
    out[idx] = tile[threadIdx.x][threadIdx.y];
}

__global__ void setColReadCol2(int *out)
{
    // static shared memory
    __shared__ int tile[BDIMY][BDIMX];

    // mapping from 2D thread index to linear memory
    unsigned int idx = threadIdx.y * blockDim.x + threadIdx.x;

    // convert idx to transposed coordinate (row, col)
    unsigned int irow = idx / blockDim.y;
    unsigned int icol = idx % blockDim.y;
```

```
    // shared memory store operation
    tile[icol][irow] = idx;

    // wait for all threads to complete
    __syncthreads();

    // shared memory load operation
    out[idx] = tile[icol][irow] ;
}

__global__ void setRowReadCol(int *out)
{
    // static shared memory
    __shared__ int tile[BDIMY][BDIMX];
// mapping from 2D thread index to linear memory
    unsigned int idx = threadIdx.y * blockDim.x + threadIdx.x;

    // convert idx to transposed coordinate (row, col)
    unsigned int irow = idx / blockDim.y;
    unsigned int icol = idx % blockDim.y;

    // shared memory store operation
    tile[threadIdx.y][threadIdx.x] = idx;

    // wait for all threads to complete
    __syncthreads();

    // shared memory load operation
    out[idx] = tile[icol][irow];
}

__global__ void setRowReadColPad(int *out)
{
    // static shared memory
    __shared__ int tile[BDIMY][BDIMX + IPAD];

    // mapping from 2D thread index to linear memory
    unsigned int idx = threadIdx.y * blockDim.x + threadIdx.x;

    // convert idx to transposed (row, col)
    unsigned int irow = idx / blockDim.y;
    unsigned int icol = idx % blockDim.y;

    // shared memory store operation
    tile[threadIdx.y][threadIdx.x] = idx;

    // wait for all threads to complete
    __syncthreads();
```

```
    // shared memory load operation
    out[idx] = tile[icol][irow] ;
}

__global__ void setRowReadColDyn(int *out)
{
    // dynamic shared memory
    extern  __shared__ int tile[];
    // mapping from thread index to global memory index
    unsigned int idx = threadIdx.y * blockDim.x + threadIdx.x;

    // convert idx to transposed (row, col)
    unsigned int irow = idx / blockDim.y;
    unsigned int icol = idx % blockDim.y;

    // convert back to smem idx to access the transposed element
    unsigned int col_idx = icol * blockDim.x + irow;

    // shared memory store operation
    tile[idx] = idx;

    // wait for all threads to complete
    __syncthreads();

    // shared memory load operation
    out[idx] = tile[col_idx];
}

__global__ void setRowReadColDynPad(int *out)
{
    // dynamic shared memory
    extern  __shared__ int tile[];

    // mapping from thread index to global memory index
    unsigned int g_idx = threadIdx.y * blockDim.x + threadIdx.x;

    // convert idx to transposed (row, col)
    unsigned int irow = g_idx / blockDim.y;
    unsigned int icol = g_idx % blockDim.y;

    unsigned int row_idx = threadIdx.y * (blockDim.x + IPAD) + threadIdx.x;

    // convert back to smem idx to access the transposed element
    unsigned int col_idx = icol * (blockDim.x + IPAD) + irow;

    // shared memory store operation
    tile[row_idx] = g_idx;
```

```
    // wait for all threads to complete
    __syncthreads();

    // shared memory load operation
    out[g_idx] = tile[col_idx];
}

int main(int argc, char **argv)
{
    // set up device
    int dev = 0;
    cudaDeviceProp deviceProp;
    CHECK(cudaGetDeviceProperties(&deviceProp, dev));
    printf("%s at ", argv[0]);
    printf("device %d: %s ", dev, deviceProp.name);
    CHECK(cudaSetDevice(dev));

    cudaSharedMemConfig pConfig;
    CHECK(cudaDeviceGetSharedMemConfig ( &pConfig ));
    printf("with Bank Mode:%s ", pConfig == 1 ? "4-Byte" : "8-Byte");

    // set up array size
    int nx = BDIMX;
    int ny = BDIMY;

    bool iprintf = 0;

    if (argc > 1) iprintf = atoi(argv[1]);

    size_t nBytes = nx * ny * sizeof(int);

    // execution configuration
    dim3 block (BDIMX, BDIMY);
    dim3 grid  (1, 1);
    printf("<<< grid (%d,%d) block (%d,%d)>>>\n", grid.x, grid.y, block.x,
            block.y);

    // allocate device memory
    int *d_C;
    CHECK(cudaMalloc((int**)&d_C, nBytes));
    int *gpuRef  = (int *)malloc(nBytes);

    CHECK(cudaMemset(d_C, 0, nBytes));
    setRowReadRow<<<grid, block>>>(d_C);
    CHECK(cudaMemcpy(gpuRef, d_C, nBytes, cudaMemcpyDeviceToHost));

    if(iprintf)  printData("setRowReadRow       ", gpuRef, nx * ny);
CHECK(cudaMemset(d_C, 0, nBytes));
    setColReadCol<<<grid, block>>>(d_C);
```

```
    CHECK(cudaMemcpy(gpuRef, d_C, nBytes, cudaMemcpyDeviceToHost));

    if(iprintf)  printData("setColReadCol       ", gpuRef, nx * ny);

    CHECK(cudaMemset(d_C, 0, nBytes));
    setColReadCol2<<<grid, block>>>(d_C);
    CHECK(cudaMemcpy(gpuRef, d_C, nBytes, cudaMemcpyDeviceToHost));

    if(iprintf)  printData("setColReadCol2      ", gpuRef, nx * ny);

    CHECK(cudaMemset(d_C, 0, nBytes));
    setRowReadCol<<<grid, block>>>(d_C);
    CHECK(cudaMemcpy(gpuRef, d_C, nBytes, cudaMemcpyDeviceToHost));

    if(iprintf)  printData("setRowReadCol       ", gpuRef, nx * ny);

    CHECK(cudaMemset(d_C, 0, nBytes));
    setRowReadColDyn<<<grid, block, BDIMX*BDIMY*sizeof(int)>>>(d_C);
    CHECK(cudaMemcpy(gpuRef, d_C, nBytes, cudaMemcpyDeviceToHost));

    if(iprintf)  printData("setRowReadColDyn    ", gpuRef, nx * ny);

    CHECK(cudaMemset(d_C, 0, nBytes));
    setRowReadColPad<<<grid, block>>>(d_C);
    CHECK(cudaMemcpy(gpuRef, d_C, nBytes, cudaMemcpyDeviceToHost));

    if(iprintf)  printData("setRowReadColPad    ", gpuRef, nx * ny);

    CHECK(cudaMemset(d_C, 0, nBytes));
    setRowReadColDynPad<<<grid, block, (BDIMX + IPAD)*BDIMY*sizeof(int)>>>(d
    CHECK(cudaMemcpy(gpuRef, d_C, nBytes, cudaMemcpyDeviceToHost));

    if(iprintf)  printData("setRowReadColDynPad ", gpuRef, nx * ny);

    // free host and device memory
    CHECK(cudaFree(d_C));
    free(gpuRef);

    // reset device
    CHECK(cudaDeviceReset());
    return EXIT_SUCCESS;
}
```

OUTPUT:

```
/tmp/tmpzvzm5d7c/a3ff9560-3908-4377-9585-c4595dec3c96/cuda_exec.out at device 0: Tesla T4 with Bank Mode:4-Byte <<< grid (1,1) block (16,16)>>>
```

RESULT:

The Matrix transposition on shared memory with grid (1,1) block (16,16) is demonstrated successfully.