key points

A summary for the beginner tutorial.

The introduction of memory coalescing, helping memory program better.

I have to admit that this chapter is pretty difficult, it should be pay great attention on the conclusion part.

1

1.1

```
unsigned int row = blockIdx.y*blockDim.y + threadIdx.y;
unsigned int col = blockIdx.x*blockDim.x + threadIdx.x;
if (row < Width && col < Width){
    float Pvalue = 0.0f;
    for(unsigned int k = 0;k < Width;++k){
        Pvalue += N[row*Width + k]*M[col*Width + k];
    }
    P[row*Width + col] = Pvalue;
}
```

Basically correct, while using shared memory, tilting and branching can make it perform better. An optimized kernel with corner turning and shared memory:

```
#define TILE_WIDTH 16
__global__ void matrixMulCornerTurning(float* N, float* M, float* P, int
    Width) {
    // Calculate the row and column index of the element
    int row = blockIdx.y * TILE_WIDTH + threadIdx.y;
    int col = blockIdx.x * TILE_WIDTH + threadIdx.x;

    // Allocate shared memory for tiles of N and M
    __shared__ float tileN[TILE_WIDTH][TILE_WIDTH];
    __shared__ float tileM[TILE_WIDTH][TILE_WIDTH];

    float Pvalue = 0.0f;

    // Loop over tiles
    for (int t = 0; t < (Width + TILE_WIDTH - 1) / TILE_WIDTH; ++t) {
        // Load elements into shared memory with corner turning
        if (row < Width && (t * TILE_WIDTH + threadIdx.x) < Width)
            tileN[threadIdx.y][threadIdx.x] = N[row * Width + t *
                TILE_WIDTH + threadIdx.x];
        else
            tileN[threadIdx.y][threadIdx.x] = 0.0f;

        if (col < Width && (t * TILE_WIDTH + threadIdx.y) < Width)
            // Transpose M while loading to ensure coalesced access
            tileM[threadIdx.x][threadIdx.y] = M[col * Width + t *
                TILE_WIDTH + threadIdx.y];
        else
            tileM[threadIdx.x][threadIdx.y] = 0.0f;

        __syncthreads();

        // Multiply the two tiles
        for (int k = 0; k < TILE_WIDTH; ++k) {
            Pvalue += tileN[threadIdx.y][k] * tileM[threadIdx.x][k];
        }

        __syncthreads();
    }
```

```
        // Write the result to global memory
        if (row < Width && col < Width) {
            P[row * Width + col] = Pvalue;
        }
    }
```

1.2

The fit value of BLOCK_SIZE could be the ~~multiple~~ **divisor** number of the ~~width~~ **warp width**. This shall makes it fully consider all elements without extra operation.

Common values like 8,16 and 32 align with warp size considerations and hardware constraints.For ensuring coalesced memory accesses effectively.

Using multiples of wap size can lead to inefficient resource utilization and exceed the maximum threads per block.

1.3

1.3.1

~~uncoaleseed~~ coalesced, a[i] where i = BlockIdx.x * BlockDim.x + threadIdx.x.

1.3.2

~~uncoaleseed~~ NA, each thread accesses a unique index threadIdx.x, instead, bank conflicts are the primary concern.

1.3.3

coalesced.

1.3.4

~~NA~~ coalesced, for each j, threads access c[i*4+j]. Assuming i is consecutive across threads, the accessed address are also consecutive for each j.

1.3.5

~~uncoaleseed~~ NA, each thread accesses a unique index based on threadIdx.x and loop variable j. It should concern the bank conflicts rather than coalescing.

1.3.6

~~coaleseed~~ NA, each thread access a unique index based on threadIdx.x.

1.3.7

~~NA~~ coalesced, for each i+8, if i is consecutive across threads, they are consecutive memory addresses.

1.3.8

~~coaleseed~~ NA, only one accesses for an index, based on threadIdx.x.

1.3.9

NA

coalesced access maximize menory throughput and minimize latency, leading to significant performance improvements.(For contiguous acces, aligned access and same memory space)

uncoalesced access orrors when threads in a warp access memory loactions that are scattered or noncontiguous, leading to multiple memory transactions, increasing latency and reducing overall performance.(For strided access, random access and misaligned access) e.g.:B[idx] = A[4 * idx], index like 0,4,8,12$\cdots$ are not contiguous.

coalescing not available pertains to certain types of memory accesses where coalescing is irrelevant or handled differently.(For shared memory access, registers and loacl memory, constant and texture memory)

e.g.:__shared__ float sharedA[256];sharedA[idx] = A[idx];B[idx] = sharedA[idx] * 2.0f; where threads access sharedA[idx], which is shared memory.

1.4

1.4.1

per element C[i][j], Multiplications: N, additions:N-1, total FLOPs $\approx$ 2N. Total FLOPs for entire matrix: $N^2 \times 2N = 2N^3$.

Memory access: matrix A: each element accessed once per C[i][j], total is N$\hat{3}$(number of elements(N^2) * access per element(N) = N^3), matrix B is also N^3. Global memory writes for matrix C = N^2.

So total memory accesses = N^3 + N^3 + N^2 = $2N^3$ + N^2 $\approx$ $2N^3$.

Bytes = $2N^3$ * 4 = $8N^3$ bytes.

So OP/B = $\frac{2N^3}{8N^3} = \frac{1}{4} = 0.25 OP/B$.

1.4.2

FLOPs is also $2N^3$. Memory access is $2 * \frac{N^3}{32} = \frac{N^3}{16}$. Bytes = $N^3/4$ bytes.

So, OP/B = $\frac{2N^3}{N^3/4}$.

1.4.3

For a further step, coarsening is: $\frac{\frac{N^3}{4}}{4(the thread coarsening factor)} = \frac{N^3}{16}$ bytes. So it should be $\frac{2N^3}{\frac{N^3}{16}} = 32$ OP/B.,

Conclusion: memory access ratio = $\frac{Total Floating Point Operations}{Bytes accessed}$

A tile can decrease N times for a N*N tile, and coalesce can decrease N time of the total Operations Per Byte (OP/B) (under ideal circumstance, depend on the memory access pattern and how ell the data accesses align with the GPU's memory architecture.).