key point

Allocate variables in their suitable memory position. Tiling can divide large matrices into smaller submatrices With data reusing and improving cache efficiency.

For naive approach of matrices multiplies, A(M*K), B(K*N) and C(M*N). Each element of A is accessed N time and B is M times (for each column of B and each row if A).

The tiling method is like the block matrix multiplication in Linear Algebra. Reducing the need of exchanging data from device to host since a smaller matrix needs less data and faster speed in computing. For example: Multilevel segmentation.

For a **rectangle matrix**, we can add 0.0f to it and make it becomes a square matrix. Thus the x and y dimension would be a multiple of submatrices. This won't have bad influence on previous matrix.

1 solutions

1.1

No, it can't. For matrix addition, every element would be used for only one times. There's no need for reuse of the repeated elements.

Single use per element and **no data reuse**.

1.2

For a 2×2 tiling, it takes 4×4 blocks to compute the matrix, while a 4×4 tiling takes 2×2 blocks to compute the matrix. The amount of data exchange for the previous one is 4 times larger than the last one. ~~So it is proportional to the dimension size of tiles.~~

The relationship is inversely proportional to the square of the tile dimension.

1.3

If one forget the first function it may leading a mistake on the multiplies in Pvalues, since Mds and Nds haven't fully prepared for the next new calculation.

~~While for the second function, it won't leading many errors because it just order all thread waiting for all other threads for the next iteration.~~

The function *__syncthreads*() serves as a barrier synchronization point, ensuring that all threads in the block reach the barrier before any thread proceeds beyond it. This guarantees that all shared memory operations (reads and writes) preceding the barrier are completed and visible to all threads.

The first one ensures that all data is loaded into Mds and Nds before any thread begins computation. Lacking it leads to threads using incomplete data, resulting in correct multiplication results.

The second one ensures that all threads have finished computing with the current tiles before they starts loading the next tiles. Lacking ut allows threads to start loading new data into shared memory before all threads have finished computing with the current data, causing data races and incorrect results.

In conclusion, *__syncthreads*() can ensures data integrity for preventing race conditions and ensuring correct results in case that threads operate on incomplete or corrupted data, resulting in incorrect computations and unpredictable behavior.

1.4

When we have lots of registers and shared memory, the top selection is shared memory since it can pass data quicker than registers. Registers need exchange data from ~~warps to warps~~, while shared memory can avoid this step.

Shared memory accessibility, synchronization and avoiding redundant loads.

Registers are private to each thread and cannot be directly accessed by other threads or warps. This require each thread to have its own copy, leading to redundancy. While shared memory allows all threads within a block to access the same data without the need for data exchange between warps. It reduces the need for redundant data storage and minimizes the overhead associated with data synchronization between threads.

However, for a single thread, register can reaches highest speed than any other ways of expressing data.

1.5

~~For about $\lfloor M\%32 \rfloor \times \lfloor N\%32 \rfloor$~~

By a factor of 32.

Though for matrices where M or N are not multiples of 32, zero-padding is not required in considering it efficiency since the presence of some partial tile slightly decreases the overall reduction efficiency but does not change much of the asymptotic bandwidth reduction, which remains O(1/32).

1.6

A kernel launched with 1000 thread blocks of 512 threads shall makes every single thread's register a copy. So the answer is $512 \cdot 1000$.

1.7

For a variable placed in shared memory, every thread in the same block would use the shared memory together. So the answer is 1000.

1.8

1.8.1

For 2*N times.

1.8.2

~~For $\lceil 2 * N/T \rceil$ times.~~

For $2 * \lceil N/T \rceil$ times.

1.9

Since there are 36 floating-point operations and 7 * 32-bit global memory accesses/s. 36 FLOPs per thread, 7 accesses per thread and 7*32 bits = 224bits = 28bytes.

So, the operational intensity is $\frac{FLOPs}{Bytes} = \frac{36}{28} \approx 1.29 FLOPs/Byte$.

For device configurations:

1.9.1

~~For 200 GFLOPS and 100 GB/second. it doesn't reach the limit of the GFLOP line, it is compute-bound.~~

Compute capacity = 200 GFLOPS. Memory capacity in FLOPs = Memory Bandwidth * OI = 100 * 1.29 = 129 GFLOPS,

So 129 ¡ 200, it is memory-bound.

1.9.2

~~It reaches almost the line, so it is both compute-bound and memory-bound.~~ Memory capacity in FLOPs = Memory Bandwidth * OI = 250 * 1.29 = 322.5 GFLOPS.

So 322.5 ¿ 300, it is compute-bound.

Roofline model, defined by compute capacity and memory capacity.

1.10

1.10.1

~~For those number that is smaller than block's width and height.~~

In thread block dimensions, CUDA allows 1024 threads per block and 32 block at least. So the limit of 20 is safe.

Shared memory takes 20*20*sizeof(float) = 1600 bytes, lower than the common 48KB.

So, all BLOCK_WIDTH values from 1 to 20 will allow the kernel to execute correctly.

1.10.2

~~It didn't execute boundary check for the matrix. Since some BLOCK_WIDTH may not be the multiple of the matrix. Padding zero to the matrix is necessary.~~

i: inconsistent variable naming: The BLOCK_WIDTH and BLOCK_SIZE are used interchangeable, it should be declared before compiling.

ii: Missing function __syncthreads(), without a synchronization, it may cause unpredictable problems.

I totally misunderstand this question.

1.11

1.11.1

8*128, since every thread in every block need a variable i.

1.11.2

8*128, same as a.

1.11.3

8, it is stored in shared memory.

1.11.4

8, same as c.

1.11.5

129*bytes(float).**=(1+128)×4=512 bytes.**

1.11.6

I don't know how to solve it.

For the floating-point operations per byte (OP/B), we need to calculate: i: FLOPs:
Multiplications:
2.5f * x[0];
3.7f * x[1];
6.3f * x[2];
8.5f * x[3];
y_s * b_s[threadIdx.x].
Counts 5
Additions: adding the results together, it takes 5.(For 6 numbers added together, it takes 5 steps.)
For the step: y_s = 7.4f, it is a data movement operation and does not constitute a floating-point operaton.

So, total FLOPs is 10.

ii: Global Memory Accesses:
Reads:
x[j] = a[j*blockDim.x*gridDim.x + i]; → 4 reads from a[]
b_s[threadIdx.x] = b[i]; → 1 read from b[]
Writes:
b[i] = ...; → 1 write to b[]
Total number per threads: 6.

Total Bytes accessed per thread: 6*size(float) = 6*4 = 24 bytes.

iii: OP/B Calculation:
The ratio = 10/24 ≈ 0.4167 OP/B.

1.12

1.12.1

The maximum number of threads is 32*64=2048, number of registers is 55296, and shared memory takes 128KB which is larger than 96KB. So it can't achieve full occupancy, the shared memory is the limiting factor.

1.12.2

The maximum number of threads is 32*256=8192, much larger than 2048 threads/SM. So it can't achieve full occupancy.**Besides, both registers and shared memory also surpass their respective capacities.**