## key point

A block contains many wraps with a few threads. For ensuring its synchronization, using $\_\_syncwarp()$.

Those in different brunch order in the same warp will be under control divergence for keeping synchronous.

Uncorrected allocating the registers and blocks may lead of great waste of efficiency.

The following function can help get number of CUDA devices:

```
int devCount;
cudaGetDeviceCount(&devCount);
```

This one returns its properties with its number:

```
cudaDeviceProp devProp;
for(unsigned int i = 0; i < devCount; i++) {
    cudaGetDeviceProperties(&devProp, i);
    // Decide if device has sufficient  resources/capabilities
}
```

Pay attention to the number of blocks,threads and registers for maximum the number of threads.

## 1 Solutions

### 1.1

#### 1.1.1

4

There's 128 threads in a block, the number of thread in a warp is 32, so the answer is 128/32 = 4.

#### 1.1.2

32

In a grid, there is 8 block totally, so the number of warp is $4 \cdot 8 = 32$

#### 1.1.3

i:~~3~~ 24

In every block, there is 3 warps are active which is warp 0,1 and 3. While there is 8 blocks in the grid, so there is $3 \cdot 8 = 24$ active warps.

ii:~~2~~ 16

A divergent warp is that partly is active while the other is inactive. As the 3.iexpressed, the number of divergent is $2 \cdot 8 = 16$ instead single 16.

iii:100%

In 0-31 all thread ¡ 40, it is fully active.

iv:25%

In 32-63, only 32-40 is active, so the SIMD efficiency is $(40 - 32)/32$.

v:75%

In 97-128, only 104-128 is active, so the SIMD efficiency is $(128 - 104)/32$.

#### 1.1.4

i:32 ii:32 iii:50%

#### 1.1.5

i:~~42~~ 3

In iteration, i can be form 0-127, i%3 can be 0-2. The iterating time is depend on i%3 of every thread. For j = 0,1,2 all iteration would compute it. While for j = 3,4 it may stop when i=1/2 that leads j ¿= 5 - i%3.

ii:~~129~~ 2

As a result the number of divergent iteration is 2 for j = 3 and 4.

## 1.2

2048

Each block can hold 512 threads, and each thread handles one element, it need enough blocks to cover all 2000 elements.

Number of Blocks = $\lceil 2000/512 \rceil = 4$ blocks.

Total threads = $4 blocks threads per block = 2048 threads$.

## 1.3

1

Only warp 62(1984-2015) is divergence for 1984-1999 is valid and 2000-2015 is invalid.

Other over 62 is invalid and lower 62 is valid.

## 1.4

17%

The maximum execution time = 3.0 microseconds. Waiting time is the minimum execution time = 1.9 microseconds.

Total execution time = $\sum execution time per thread/(8 \cdot minimum execution time)\% \approx 17.08\%$

## 1.5

~~No, the number of warp is limited by the device, setting only 32 thread cause the use of warp increase. A limited number of warp can use fewer threads to compute resulting in a loss of efficiency.~~

**I copied GPT's answer:**

No, omitting __syncthreads() is not recommended even when each block has only 32 threads (one warp). While it's true that a single warp executes in lockstep, __syncthreads() ensures memory operations are completed and visible to all threads within the block. Without it, you risk memory inconsistencies and race conditions. Additionally, using only 32 threads per block can limit the GPU's ability to hide latency and fully utilize its computational resources, potentially leading to reduced efficiency.

**Here's detailed reasons:** It is not suggested to leave this function out for the following reasons:

1.__syncthreads() keeps all memory operations work before threads within the block are completed for keeping memory safe.

2. It can prevent reordering synchronization to maintaining the intended execution order.

3. Even if current block size is 32 threads, further changes might involve increasing block size for more robust and easier to maintain or scale.

4. Warp-level synchronization isn't sufficient, it also need block-level synchronization.

**Correcting my answer:**

A restrict number of block do exists, while the actual number is very high. FOr grid dimension.x:$2^{31} - 1$, for gird dimension.y:65535, for grid dimension.z:65535. And the maximum number per block is constrained: 1024 thread per block. Accessible via the cudaDevideProp structure.

What infect most is resource constraints. Each block consumes shared memory and registers causing lower occupance and performance degradation.

## 1.6

C

For option A, number of block:4. Total threads is $128 \cdot 4 = 512$ threads.

For option B, number of block:4. Total threads is $256 \cdot 4 = 1024$ threads.

For option C, number of block:3. Total threads is $512 \cdot 3 = 1536$ threads.

For option D, number of block:1. Total threads is $1024 \cdot 1 = 1024$ threads.

## 1.7

a,50%,b,50%,c,50%,d,100%,e,100%

All numbers of block are $\leq 64$ and all numbers of threads are $\leq 2048$.

## 1.8

a. No. The number of total thread is $32 \cdot 128 = 4096$. Much higher than 2048 threads.

b. Yes. All parameter is lower than setting(1024 threads and 29696 registers).

c. No. The number of block and register is far higher than setting $256 \cdot 32 = 8192$.

## 1.9

It may be not enough to multiply two matrices. Since multiplying two matrices takes not only the number of its elements. With $512 \cdot 8$ threads is far form necessary threads. It would be impossible.

**Not exactly:**

The student's configuration exceeds the maximum allowed threads per block by a factor of 2.

Total threads per SM: $512 \cdot 8 = 4096$ threads/SM.

Result matrix elements: $1024 \cdot 1024 = 1048576$ elements. Insufficient to handle all elements efficiently.

Revised approach:

Adjust thread block size. Make it lower than 512 as the limitation. For example: setting $16 \times 16 = 256$ threads per block. Total block needed is $(1024/16) \times (1024/16) = 4096$ blocks. Up to 8 blocks per SM means no more than 8 active blocks, allowing the GPU to manage workload distribution effectively, not the maximnum number of block can be set.