CUDA Data Alignment

Introduction

In order to get the best performance, similar to the data alignment requirement in C++, CUDA also requires data alignment.

In this blog post, I would like to quickly discuss the data alignment requirement in CUDA.

Coalesced Access to Global Memory

Global memory resides in device memory and device memory is accessed via 32-, 64-, or 128-byte memory transactions. These memory transactions must be naturally aligned: Only the 32-, 64-, or 128-byte segments of device memory that are aligned to their size (i.e., whose first address is a multiple of their size) can be read or written by memory transactions.

When a warp executes an instruction that accesses global memory, it coalesces the memory accesses of the threads within the warp into one or more of these memory transactions depending on the size of the word accessed by each thread and the distribution of the memory addresses across the threads. In general, the more transactions are necessary, the more unused words are transferred in addition to the words accessed by the threads, reducing the instruction throughput accordingly.

For devices of compute capability 6.0 or higher, the requirements can be summarized quite easily: the concurrent accesses of the threads of a warp will coalesce into a number of transactions equal to the number of 32-byte transactions necessary to service all of the threads of the warp.

Any address of a variable residing in global memory or returned by one of the memory allocation routines from the driver or runtime API, such as `cudaMalloc` or `cudaMallocPitch`, is always aligned to at least 256 bytes.

Example

For example, if the each of the thread in a warp that consists of 32 threads wants to read a 4-byte data, and if the 4-byte data from all the threads in the warp (128-byte data) are adjacent to each other and 32-byte aligned, i.e., the the address of the first 4-byte data is a multiple of 32, the memory access is coalesced and GPU will make 4×3232=44×3232=4 32-byte memory transactions. Maximum memory transaction throughput is achieved because GPU made the fewest transactions possible.

If the 128-byte data is not 32-byte aligned on the memory, say it is 4-byte aligned instead, one additional 32-byte memory transactions will have to be made, and therefore the memory access throughput becomes 45=80%45=80% of the maximum theoretical throughput (speed of light).

Furthermore, if the 4-byte data from all the threads are not adjacent to each other and are scattered sparsely on the memory, it is possible that at most 32 32-byte memory transactions will have to be made, and the throughput becomes only 432=12.5%432=12.5% of the maximum theoretical throughput.

Size and Alignment Requirement

Global memory instructions support reading or writing words of size equal to 1, 2, 4, 8, or 16 bytes. Any access (via a variable or a pointer) to data residing in global memory compiles to a single global memory instruction if and only if the size of the data type is 1, 2, 4, 8, or 16 bytes and the data is naturally aligned (i.e., its address is a multiple of that size).

If this size and alignment requirement is not fulfilled, the access compiles to multiple instructions with interleaved access patterns that prevent these instructions from fully coalescing. It is therefore recommended to use types that meet this requirement for data that resides in global memory.

Reading non-naturally aligned 8-byte or 16-byte words produces incorrect results (off by a few words), so special care must be taken to maintain alignment of the starting address of any value or array of values of these types.

Therefore, working with word of size equal to 1, 2, 4, 8, or 16 bytes is sometimes straightforward because as mentioned above the starting memory address returned by the memory allocation CUDA APIs is always aligned to at least 256 bytes, which is already 1, 2, 4, 8, or 16 byte-aligned. So we could safely save the word sequence, such as a numerical array, matrix, or tensor, into the allocated memory without having to worry about the reading of the words of 8-byte or 16-byte size produces incorrect results. To achieve the best memory access throughput, special attention is paid to the kernel implementation so that the coalesced memory access is also naturally aligned.

But, what if the word size is not 1, 2, 4, 8, or 16 bytes? The starting memory address returned by the memory allocation CUDA APIs will not guarantee that it is naturally aligned, and therefore the memory access throughput would be compromised significantly. There are usually two ways to handle this.

1. Use the Built-in Vector Types whose alignment requirements is already specified and fulfilled.
2. Similar to the compiler specifier `alignas` used for enforcing the structure data alignment in GCC, the compiler specifier `__align__` is used for enforcing the structure data alignment in NVCC.

```
struct __align__(4) int8_3_4_t
{
    int8_t x;
    int8_t y;
    int8_t z;
};

struct __align__(16) float3_16_t
{
    float x;
    float y;
    float z;
};
```

Conclusions

Always making the word size equal to 1, 2, 4, 8, or 16 bytes and the data naturally aligned.

Reading words and producing incorrect results rarely happen if the memory allocated is only used for a sequence of words of the same type, because the starting memory address returned by the memory allocation CUDA APIs is always aligned to at least 256 bytes. However, if one single piece of large memory is allocated for multiple sequences of words of different types with or without paddings, special care must be taken to maintain alignment of the starting address of any word or word sequence as it may produce incorrect result (for non-naturally aligned 8-byte or 16-byte words).

References