Guide: Data Alignment for 128-bit Data in Intel Architectures

Overview

When working with 128-bit data types (e.g., __m128) in Intel architectures, it's crucial to ensure that data is 16-byte aligned. Misalignment can lead to performance penalties and, in some cases, execution faults. This guide outlines how to properly align data and leverage compiler features to achieve optimal performance.

1. Importance of Data Alignment

Data must be 16-byte aligned when loading to and storing from the 128-bit XMM registers. The benefits of proper alignment include:

- **Avoiding Performance Penalties**: Accessing unaligned data is significantly slower.
- **Preventing Execution Faults**: Using aligned instructions on misaligned data can cause faults.

2. Compiler-Supported Alignment Techniques

2.1 Alignment by Using Data Types

F32vec4 or __m128 Data Types

The Intel C++ Compiler automatically aligns F32vec4 or __m128 types to a 16-byte boundary:

- Both global and local data are aligned.
- Stack frames for functions are aligned as well.

2.2 Using __declspec(align(16))

You can explicitly specify 16-byte alignment using the __declspec(align(16)) directive. This is useful for ensuring that both local and global variables are properly aligned.

Example:

```
__declspec(align(16)) float buffer[400];
// This allows the use of buffer as if it contains 100 objects of type __m128
```

2.3 Alignment by Using a Union Structure

Using a union can automatically provide the correct alignment for your data structure. This approach makes it clear to the compiler that you are working with 128-bit data.

Example:

```
union {
   float f[400];
   __m128 m[100];
} buffer;
// This ensures 16-byte alignment due to the __m128 type in the union.
```

2.4 Alignment in Classes/Structs

In C++, you can enforce alignment for classes or structs using __declspec(align(16)). However, using a union is often preferred for clarity.

Example:

```
class my_m128 {
   union {
      __m128 m;
      float f[4];
   };
};
// This allows for both float and __m128 access while ensuring alignment.
```

2.5 Alignment with __m64 or Double Data

The compiler may align routines with __m64 or double data types to 16 bytes by default. You can control this behavior with the command-line switch -QSFALIGN16.

- **Default Behavior**: -QSFALIGN8 aligns to 8 or 16 bytes.
- **Changed Behavior**: -QSFALIGN16 aligns routines specifically for 128-bit data.

3. Improving Memory Utilization

Improving memory performance for SSE, SSE2, and MMX technology can involve the following techniques:

3.1 Data Structure Layout

Optimize the layout of data structures to enhance cache usage and access patterns.

3.2 Strip-Mining for Vectorization

Reorganize loops and data accesses to take advantage of SIMD vectorization.

3.3 Loop-Blocking

Break down loops into smaller blocks to better utilize cache memory.

3.4 Prefetch and Streaming Store Instructions

Utilize cacheability instructions to improve memory utilization. Prefetching data can reduce cache misses.

Conclusion

Properly aligning 128-bit data is critical for maximizing performance when using Intel's SSE instructions. By leveraging the alignment features of the Intel C++ Compiler and optimizing memory usage, you can significantly improve the performance of your applications. For more detailed information, refer to the Intel compiler documentation.