1. Die in der Folie 10 gezeigten Leistungsunterschiede zwischen Compilern haben mehrere mögliche Gründe:

SIMD-Optimierungen : Der Code nutzt SIMD durch OpenMP #pragma omp simd. Diese ermöglichen die Vektorisierung, die erweiterte SIMD-Anweisungen wie AVX2 nutzt, um mehrere Elemente parallel zu verarbeiten. Clang generiert besser optimierten SIMD-Code als GCC und Intel Compiler Clang nutzt die Hardware effektiver.

Schleifenentrollung: Das Entrollungsmakro (\_\_unroll(8)) hilft, den Schleifen-Overhead zu reduzieren und bietet dem Compiler Möglichkeiten zur weiteren Optimierung. Clang verfügt über eine bessere Schleifenoptimierung und Entrollungsheuristik eine effizientere Ausführung.

Speicherausrichtung: Die Aligned-Klausel stellt sicher, dass der Speicher an 64-Byte-Grenzen ausgerichtet ist Anforderungen für AVX2-Register. Falsch ausgerichteter Speicher kann zu Nachteilen führen. Einige Compiler (z. B. Clang) handhaben die Ausrichtung effektiver als andere.

Clangs überlegener Vektorisierer: Clangs Implementierung der Schleifenvektorisierung und Befehlsplanung ist als besonders leistungsstark bekannt. Er kann engere Schleifen erzeugen, Speicherabhängigkeiten besser handhaben und intrinsische SIMD-Befehle effizienter nutzen als GCC oder der Intel-Compiler.

Compilerspezifische Optimierungen: Verschiedene Compiler priorisieren Optimierungen unterschiedlich. Clang verfügt möglicherweise über erweiterte Optimierungen für Gleitkommaoperationen und Speicherauslagerungen (z. B. std::swap) Übereinstimmung besser mit dem AVX2-Befehlssatz.

1. Latency (Latenz) ist die Anzahl an Taktzyklen, die eine Instruktion braucht, bis ihr Ergebnis zur Verfügung steht. Je niedriger die Latenz, desto schneller steht das Resultat für abhängige Berechnungen bereit. Throughput (Durchsatz) beschreibt, wie viele Instruktionen pro Taktzyklus abgearbeitet werden können. Hoher Durchsatz ist vorteilhaft, wenn viele unabhängige Operationen parallel ausgeführt werden können (z.B. bei SIMD).

Man hat vier mögliche Szenarien:

**Low Latency, High Throughput**: Ideal für schnelle Ausführung und Parallelisierung.

**Low Latency, Low Throughput**: Gut für serielle Aufgaben mit vielen Datenabhängigkeiten.

**High Latency, High Throughput**: Kann Latenz verstecken, wenn genug unabhängige Instruktionen vorliegen.

**High Latency, Low Throughput**: Schlechtestes Szenario bei sequenziellen und parallelen Workloads.

Für SIMD-Intrinsics wie \_mm256\_add\_ps (Addition) sind Latenz und Durchsatz in der Regel günstig. Komplexere Befehle wie \_mm256\_div\_ps erfordern mehr Taktzyklen, was höhere Latenz und oft geringeren Durchsatz verursacht. In der Praxis wählt man je nach Workload Instruktionen mit niedriger Latenz für kaskadierende Berechnungen und Instruktionen mit hohem Durchsatz für parallele Schleifen.

1. Das Paper befasst sich mit der Leistung von vektorisierten Hashtabellen in verschiedenen CPU-Architekturen, welche Auswirkungen auf die Effizienz der Datenverarbeitung. Folgendes war für mich interessant:

Auswirkungen der CPU-Heterogenität auf der Leistung:  
Das Paper zeigt Leistungsunterschiede zwischen x86-, ARM- und Power-Architekturen bei Verwendung vektorisierter Hashschemata auf. Sie zeigt die wachsende Bedeutung von ARM-CPUs, die voraussichtlich einen erheblichen Marktanteil erreichen werden, und ihre Auswirkungen auf moderne Datenbanksysteme. Dieser Trend weist auf die Notwendigkeit anpassbarer Hashimplementierungen hin, die die Hardwarevielfalt optimiert.

Innovationen bei Hashing-Techniken:  
Die Charakterisierung und Bewertung von vektorisiertem linearem Probing (VLP), vektorisiertem Fingerprinting (VFP) und Bucket-basiertem Vergleich (BBV) zeigt, wie vektorisierte Ansätze skalare Ansätze deutlich übertreffen, insbesondere in Szenarien mit hoher Belastung. BBV beispielsweise erreicht unter Voraussetzungen die zweifache bis dreifache Leistung herkömmlicher skalarer Methoden wie Robin Hood-Hashing. Das zeigt, wie ein fein abgestimmtes Datenstrukturdesign die SIMD-Funktionen für eine optimierte Datenverarbeitung nutzen kann.