Answers of the questions

### Bandbreitenbeschränkt (Bandwidth-Bound)

Bandbreitenbeschränkte Berechnungen beziehen sich auf Situationen, in denen der Flaschenhals der Leistung hauptsächlich durch die Datenübertragungsrate begrenzt ist. In solchen Fällen wird die Geschwindigkeit der Berechnung durch die Geschwindigkeit begrenzt, mit der Daten von einem Ort (z.B. Speicher, Festplatte oder Netzwerk) zu einem anderen (z.B. Verarbeitungseinheit) bewegt werden können. Selbst wenn die Verarbeitungseinheit (z.B. CPU oder GPU) über eine starke Rechenkapazität verfügt, wird die Ausführungsgeschwindigkeit des Programms beeinträchtigt, wenn sie die Daten nicht rechtzeitig für die Verarbeitung erhalten kann.

\*\*Beispielszenarien\*\*:

- Das Lesen großer Datenmengen von einer Festplatte für die Verarbeitung, wenn die Lesegeschwindigkeit der Festplatte langsam ist.

- In verteilten Systemen, wo die Übertragungsgeschwindigkeit von Daten im Netzwerk zum Flaschenhals wird.

- In Aufgaben der Hochleistungsrechnung, wo die Speicherbandbreite nicht ausreicht, um schnellen Datenzugriff zu unterstützen.

### Rechenbeschränkt (Compute-Bound)

Rechenbeschränkte Situationen sind solche, in denen der Flaschenhals der Leistung hauptsächlich durch die Rechenkapazität der Verarbeitungseinheit begrenzt ist. In diesen Fällen wird die Ausführungsgeschwindigkeit des Programms durch die Geschwindigkeit begrenzt, mit der der Prozessor (CPU oder GPU) Anweisungen ausführen kann. Selbst wenn die Datenübertragungsgeschwindigkeit hoch ist, wird die Effizienz der Programmausführung beeinträchtigt, wenn der Prozessor nicht schnell genug mit diesen Daten umgehen kann.

\*\*Beispielszenarien\*\*:

- Komplexe mathematische Berechnungen, wie umfangreiche wissenschaftliche Berechnungen oder Bildverarbeitung, die viel CPU- oder GPU-Zeit für die Verarbeitung benötigen.

- Intensive Schleifen oder rekursive Aufrufe, bei denen der Prozessor viel Zeit für die Fertigstellung der Berechnungen benötigt.

### Zusammenfassung der Unterschiede

- \*\*Ursprung des Flaschenhalses\*\*: Bandbreitenbeschränkte Probleme stammen aus der Datenübertragungsrate, während rechenbeschränkte Probleme aus der Verarbeitungskapazität der Verarbeitungseinheit stammen.

- \*\*Strategien zur Leistungssteigerung\*\*: Bei bandbreitenbeschränkten Berechnungen ist die Erhöhung der Datenübertragungsrate (z.B. Optimierung des Speicherzugriffsmusters, Verwendung schnellerer Speichergeräte oder Erhöhung der Netzwerkbandbreite) der Schlüssel zur Leistungsverbesserung. Bei rechenbeschränkten Berechnungen ist die Steigerung der Recheneffizienz der Verarbeitungseinheit (z.B. durch Verwendung schnellerer Prozessoren, Optimierung des Algorithmus oder Parallelverarbeitung) entscheidend.

- \*\*Ressourcennutzung\*\*: In bandbreitenbeschränkten Szenarien kann die Verarbeitungseinheit eine hohe Leerlaufzeit aufweisen, während sie auf Daten wartet; in rechenbeschränkten Szenarien ist die Nutzung der Verarbeitungseinheit oft hoch, kann aber aufgrund von Algorithmen- oder Verarbeitungskapazitätsbeschränkungen nicht weiter gesteigert werden.

Das Verständnis der Unterschiede zwischen diesen beiden Beschränkungen hilft bei der Diagnose von Leistungsengpässen und der Anwendung geeigneter Optimierungsstrategien, um die Ausführungseffizienz von Programmen wirksam zu verbessern.

About the Data locality:

```c++

void matmul\_mnk(double\* i\_A,

double\* i\_B,

double\* io\_C,

std::size\_t i\_m,

std::size\_t i\_n,

std::size\_t i\_k ){

for( std::size\_t l\_m = 0; l\_m < i\_m; l\_m++ )

for( std::size\_t l\_n = 0; l\_n < i\_n; l\_n++ )

for( std::size\_t l\_k = 0; l\_k < i\_k; l\_k++ )

io\_C[l\_m\*i\_n + l\_n] += i\_A[l\_m\*i\_k + l\_k] \* i\_B[l\_k\*i\_n + l\_n];

}

```

```

//mkn

Processing row 1 / 3

C[0][0] += A[0][0] \* B[0][0] -> 3

C[0][1] += A[0][0] \* B[0][1] -> 4

C[0][2] += A[0][0] \* B[0][2] -> 0

C[0][0] += A[0][1] \* B[1][0] -> 6

C[0][1] += A[0][1] \* B[1][1] -> 8

C[0][2] += A[0][1] \* B[1][2] -> 5

C[0][0] += A[0][2] \* B[2][0] -> 18

C[0][1] += A[0][2] \* B[2][1] -> 22

C[0][2] += A[0][2] \* B[2][2] -> 21

Processing row 2 / 3

C[1][0] += A[1][0] \* B[0][0] -> 12

C[1][1] += A[1][0] \* B[0][1] -> 22

C[1][2] += A[1][0] \* B[0][2] -> 6

C[1][0] += A[1][1] \* B[1][0] -> 24

C[1][1] += A[1][1] \* B[1][1] -> 38

C[1][2] += A[1][1] \* B[1][2] -> 26

C[1][0] += A[1][2] \* B[2][0] -> 54

C[1][1] += A[1][2] \* B[2][1] -> 73

C[1][2] += A[1][2] \* B[2][2] -> 66

Processing row 3 / 3

C[2][0] += A[2][0] \* B[0][0] -> 0

C[2][1] += A[2][0] \* B[0][1] -> 6

C[2][2] += A[2][0] \* B[0][2] -> 12

C[2][0] += A[2][1] \* B[1][0] -> 21

C[2][1] += A[2][1] \* B[1][1] -> 34

C[2][2] += A[2][1] \* B[1][2] -> 47

C[2][0] += A[2][2] \* B[2][0] -> 69

C[2][1] += A[2][2] \* B[2][1] -> 90

C[2][2] += A[2][2] \* B[2][2] -> 111

```

```

// nkm

Processing column 1 / 3

C[0][0] += A[0][0] \* B[0][0] -> 0

C[1][0] += A[1][0] \* B[0][0] -> 0

C[2][0] += A[2][0] \* B[0][0] -> 0

C[0][0] += A[0][1] \* B[1][0] -> 3

C[1][0] += A[1][1] \* B[1][0] -> 12

C[2][0] += A[2][1] \* B[1][0] -> 21

C[0][0] += A[0][2] \* B[2][0] -> 15

C[1][0] += A[1][2] \* B[2][0] -> 42

C[2][0] += A[2][2] \* B[2][0] -> 69

Processing column 2 / 3

C[0][1] += A[0][0] \* B[0][1] -> 0

C[1][1] += A[1][0] \* B[0][1] -> 3

C[2][1] += A[2][0] \* B[0][1] -> 6

C[0][1] += A[0][1] \* B[1][1] -> 4

C[1][1] += A[1][1] \* B[1][1] -> 19

C[2][1] += A[2][1] \* B[1][1] -> 34

C[0][1] += A[0][2] \* B[2][1] -> 18

C[1][1] += A[1][2] \* B[2][1] -> 54

C[2][1] += A[2][2] \* B[2][1] -> 90

Processing column 3 / 3

C[0][2] += A[0][0] \* B[0][2] -> 0

C[1][2] += A[1][0] \* B[0][2] -> 6

C[2][2] += A[2][0] \* B[0][2] -> 12

C[0][2] += A[0][1] \* B[1][2] -> 5

C[1][2] += A[1][1] \* B[1][2] -> 26

C[2][2] += A[2][1] \* B[1][2] -> 47

C[0][2] += A[0][2] \* B[2][2] -> 21

C[1][2] += A[1][2] \* B[2][2] -> 66

C[2][2] += A[2][2] \* B[2][2] -> 111

```

\* The first loop order (faster): In the mkn sequence, the inner loop fixes n and varies m. Since matrices A and C are stored in row-major order (the default array layout in C/C++), this means that access to A and C is sequential. This access pattern exhibits good spatial locality, thus allowing for efficient utilization of the cache.

\* The second loop order (slower): In the nkm sequence, the inner loop fixes m and varies n. When computing each element of C, it skips entire rows to access the next element (for example, from C[0][0] to C[1][0]), which does not comply with the contiguous storage of the C matrix, resulting in poor spatial locality. Each iteration may cause a cache miss and necessitate reloading data from the main memory into the cache.

### Strong Temporal Locality Example

In this example, the variable `sum` is accessed multiple times in a tight loop. This demonstrates strong temporal locality because `sum` is repeatedly used within a short period of time.

```c

#include <stdio.h>

int main() {

int sum = 0;

for (int i = 0; i < 100; i++) {

sum += i; // `sum` is accessed repeatedly

}

printf("Sum: %d\n", sum);

return 0;

}

### Example Lacking Temporal Locality

```markdown

### Lack of Temporal Locality Example

In this example, a large array is accessed in a non-sequential manner, which lacks temporal locality. The accesses to the `data` array are spread out over time and do not benefit from the cache as effectively.

```c

#include <stdio.h>

#include <stdlib.h>

int main() {

int size = 10000;

int \*data = (int\*)malloc(size \* sizeof(int));

// Initialize the array

for (int i = 0; i < size; i++) {

data[i] = i;

}

// Accessing elements in a non-sequential manner

for (int i = 0; i < size; i += 500) {

printf("%d ", data[i]); // Sparse access pattern

}

free(data);

return 0;

}

These examples illustrate how temporal locality can be effectively utilized or missed in program design, affecting the overall performance due to the caching mechanisms of modern CPUs.