Übungen zur Vorlesung "Architektur und Programmierung von Grafik- und Koprozessoren"
Übungsblatt 1

Sommersemester 2018

1 Programmieren für Cache Architekturen

Aufgabe 1.1

Gegeben sei die Matrix A, für die gilt:

$$A_{i,j} = (i, j), i, j = 0, 1, .., N - 1. \tag{1}$$

Visualisieren Sie A für $N = 8$ mit Hilfe einer Tabelle. Gehen Sie dabei wie folgt vor. Wählen Sie für die Matrixelemente deren Binärrepräsentation. Führen Sie für die Paare von Binärzahlen (i, j) die *in shuffle* Operation durch:

$$k(i, j) = (k_5, .., k_2, k_1, k_0) := (i_2, j_2, .., i_0, j_0). \tag{2}$$

Die Binärrepräsentation der sich ergebenden natürlichen Zahl k setzt sich immer abwechselnd aus den Binärziffern der Zahlpaare (i, j) zusammen. Tragen Sie das Ergebnis in die Tabelle der Größe 8×8 als binäre Festkommazahl, sowie als Dezimalzahl, ein. Verbinden Sie die Elemente gemäß ihres Betrags aufsteigend sortiert mit einem Linienzug. (5 Punkte)

Aufgabe 1.2

a.)
Übersetzen Sie die den Übungsunterlagen beigefügte Datei `morton.cpp` mit einem C++11 kompatiblen Compiler. Aktivieren Sie dabei die höchstmögliche Optimierungsstufe mit Bezug auf die zu erwartende Programmausführungsgeschwindigkeit. Eine entsprechende Kommandozeile für `gcc` kann beispielsweise wie folgt aussehen.

```
g++ morton.cpp -std=c++11 -O3 -o morton
```

Die Template Klasse `Grid` implementiert ein zweidimensionales Array dynamischer Größe, auf dessen Elemente Sie lesend und schreibend mit `operator()` zugreifen können:

```
Grid<float> grid(W, H); // 2-D Array der Größe W x H
grid(1, 2) = 3.14f; // Schreiboperation an Speicherstelle [1,2]
```

In der Funktion `main()` wird zunächst ein 2-D floating point Array erzeugt und dann mit Zufallszahlen gefüllt. Im Anschluss wird ein Bildverarbeitungsfilter auf das Array angewendet, der zunächst zeilenweise, und danach spaltenweise über das Array iteriert.

Führen Sie das vorhin kompilierte Programm aus. Was fällt Ihnen bzgl. der Ausführungszeiten für das Anwenden des Bildverarbeitungsfilters auf? Wie erklären Sie sich das Verhalten? (2 Punkte)

b.)
Fügen Sie zur Klasse `Grid` nun eine Alternativimplementierung für `operator()` hinzu, die Sie mittels des bereits definierten Flags `MORTON` zur Kompilierzeit aktivieren und deaktivieren können. Dazu implementieren Sie die *raumfüllenden Kurven* (engl.: space-filling curves), die Sie sich in Aufgabe 1.1 hergeleitet haben. Implementieren Sie eine Hilfsroutine `expand_bits()`. Mit Hilfe dieser lässt sich die *in shuffle* Operation ausführen:

```
int z = expand_bits(x) | (expand_bits(y) << 1)
```

Mit dem so erzeugten *"Morton Code"* können Sie in das Daten-Array der `Grid` Klasse hinein indexieren. `expand_bits()` zieht die x/y Koordinaten mit Hilfe von Bit-Operationen und Maskenkonstanten sukzessive auseinander:

```
1. x = --------.--------.FEDCBA98.76543210
2. x = --------.FEDCBA98.--------.76543210
3. x = ----FEDC.----BA98.----7654.----3210
4. x = --FE--DC.--BA--98.--76--54.--32--10
5. x = -F-E-D-C.-B-A-9-8.-7-6-5-4.-3-2-1-0
```

(Die mit "−" annotierten Bits ersetzen Sie durch 0-bits, damit die *in shuffle* Operation mit Hilfe des bitweisen Oders durchgeführt werden kann.)

Führen Sie das so modifizierte Programm aus. Vergleichen Sie die Ausführungszeiten der neuen Implementierung mit denen für die alte Implementierung. Was fällt Ihnen auf und welche Erklärung haben Sie? (10 Punkte)

c.)
Im vorangegangenen Aufgabenteil haben Sie eine Cache Optimierung für 2-D Datenstrukturen implementiert. Fällt Ihnen im konkreten Fall eine pragmatischere Lösung ein, um die Programmausführungszeit zu beschleunigen? (Hinweis: Bei der Lösung sei es erlaubt, die Anordnung aller Daten im Speicher zu verändern.) Können Sie sich trotzdem Umstände vorstellen, unter denen die Cache Optimierung aus dem vorangegangenen Aufgabenteil vorteilhaft ist? (Hinweis: Überlegen Sie sich, ob es Speicherzugriffsmuster mit bestimmten Charakteristika gibt, für die eine Cache Optimierung mit Morton Codes sinnvoll ist.) (3 Punkte)

Abgabe bitte bis zum 25.04.2018, 22:00h in Ilias.