Matrix-Multiplikatin

Mark Geiger Rico Fritzsche

21. April 2015

Zusammenfassung

Im Kurs Parallele Programmierung soll der Geschwindigkeitsunterschied verschiedener Umsetzungen der Matrixmultiplikation demonstriert werden. Dazu wurde ein Testprogramm erstellt, welches unterschiedliche Konzepte umsetzt und zeitlich bewertet. Zur Auswertung wurde ein Python-Skript erstellt, welches die Zeit als Funktion der Matrixgröße darstellt.

Umsetzung

In der Übung wurden die Multiplikation für quatratische Matrizen umgesetzt. Dazu sind wir in vier Schritten vorgegangen, um eine effizientere Geschwindigkeit der Operationen erzielen zu können. Jeder Schritt setzt eine eigene Implementierung der Multiplikation um. Begonnen haben wir mit der naiven Multiplikation. Diese durchläuft die Spalten der ersten Matrix und die Zeilen der zweiten Matrix, multipliziert diese und summiert die Ergebnisse. Aufgrund des zu kleinen Stacks, müssen die Matrizen mit dem C-Befehl *malloc()* angelegt werden, welcher Speicher auf dem Heap reserviert. Im zweiten Schritt haben wir den Blocke-Ansatz umgesetzt. Mit Recherche im Internet haben wir herausgefunden, dass wir zunächst mittels Schleifen festlegen müssen, in welchen Schritten das Programm die Blöcke der Matrizen durchlaufen soll. Innerhalb dieser Schleifen kann analog zu der naiven Implementierung vorgegangen werden. Dieser Ansatz zeigt im Vergleich zur naiven Implementierung einen deutlichen Geschwindigkeitsvorteil. Im dritten Schritt wurde die Bibliothek BLAS zur Berechnung der Matrixmultiplikation genutzt. Diese stellt nach der Einbindung mittels *#include cblas.h* die Funktion *cblas_dgemm()* bereit. Ihr werden unter anderem die Eingangsmatrizen übergeben, das Ergebnis wird als Pointer zurückgegeben. Auch hier konnten wir im Vergleich zu den vorherigen Ergebnissen eine Geschwindkeitszunahme der Matrixmultiplikation erkennen. Für

die letzte Anforderung haben wir den Blocked-Ansatz mittels SIMD (Single iInstruction, Multiple Data) optimiert. Dabei werden zur Berechnung die SSE (Streaming SIMD Extensions)-Einheiten des Prozessors verwendet. Diese wurden für Gleitkommazahl entwickelt und haben das Ziel auf der Instruktionsebene Programme mittels Parallelisierung zu beschleunigen. Für die Umsetzung mussten wir auf Quellen[1] im Internet zurückgreifen.

Ergebnisse

Damit die Ergebnisse verglichen werden können, wird die Funktion *get_time()* und *timespec_diff()* verwendet, welches über *#include „timing.h“* zur Verfügung stehen. Durch den Aufruf vor und nach der implementierten Matrixmultiplikation wird mittels Differenz die benötigte Zeit ermittelt. Da wir die Implementierung der BLAS-Funktionen nicht kennen, kann keine korrekte Aussage über die verwendeten Gleitkommaoperationen getroffen werden. Aus diesem Grund ist ein direkter Vergleich von MFLOPS der Ansätze nicht möglich.
Zur Auswertung werden die vier Testimplementationen mit verschiedenen Matrixgrößen wiederholt. Die Größe der Matrizen orientiert sich an der Zweier-Potenzreihe im Bereich von 16 bis 2048. Anschließend werden die Größen sowie die zur Berechnung benötigte Zeit für jeden Ansatz der Matrixmultiplikation in eine Textdatei gespeichert. Ein Python-Skript liest diese Datei ein und plottet die Ergbnisse in einem Liniendiagramm 1. Es ist zu erkennen, dass die Umsetzung der BLAS-Bibliothek eine sehr effiziente Lösung bereitstellt. Auch die SIMD-Implementierung des Blocked-Ansatz stellt eine effiziente Implementierung dar, ist jedoch marginal langsamer als die BLAS-Implementierung. Damit kann gezeigt werden, dass durch Parallelisierung eine deutliche Leistungssteigerung durch eine erhöhte Geschwindigkeit zu erreichen ist.

Ausführung

make
make run
make plot

[1] http://download.intel.com/design/PentiumIII/sml/24504501.pdf

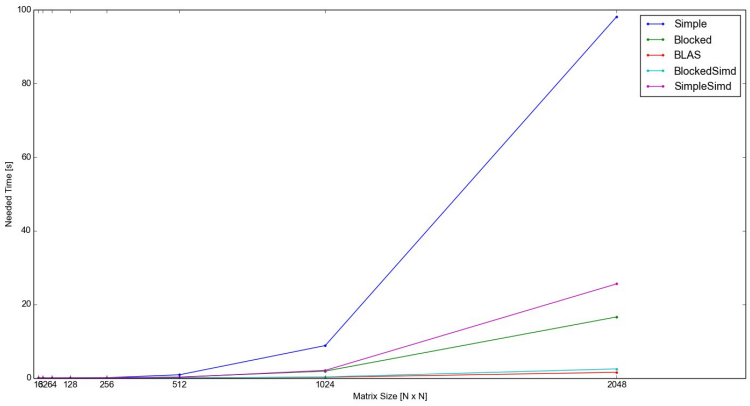

Abbildung 1: Ergebnis der Testimplementation, dargestellt mittels Python-Scripts