3. Praktikum

Gruppe 174: - Ulf Gebhardt (rbg: hu56nifa)
- Michael Scholz (rbg: mi48azih)

Aufgabe 1: Matrixmultiplikation in Assembler:

a)

$$c_{ij} = \sum_{k=1}^{m} a_{ik} * b_{kj}$$

$$C = A * B$$

$$\begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} * \begin{pmatrix} -1 & 9 & -2 \\ 3 & -11 & 5 \\ 7 & 4 & 2 \end{pmatrix}$$

$$\begin{pmatrix} 26 & -1 & 2 \\ 53 & 5 & 5 \\ 80 & 11 & 8 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} * \begin{pmatrix} -1 & 9 & -2 \\ 3 & -11 & 5 \\ 7 & 4 & 2 \end{pmatrix}$$

Wobei sich die Elemente der Matrix C wie folgt zusammensetzen:

$$c_{11} = 1 * -1 + 2 * 3 + 3 * 7 = 26$$
$$c_{12} = 1 * 9 + 2 * -11 + 3 * 4 = -1$$
$$c_{13} = 1 * (-2) + 2 * 5 + 3 * (-2) = 2$$
$$c_{21} = 4 * (-1) + 5 * 3 + 6 * 7 = 53$$
$$c_{22} = 4 * 9 + 5 * (-11) + 6 * 5 = 5$$
$$c_{23} = 4 * (-2) + 5 * 5 + 6 * (-2) = 5$$
$$c_{31} = 7 * (-1) + 8 * 3 + 9 * 7 = 80$$
$$c_{32} = 7 * 9 + 8 * (-11) + 9 * 4 = 11$$
$$c_{33} = 7 * (-2) + 8 * 5 + 9 * (-2) = 8$$

b) ⇨ Extra Programm (Datei: matrixmultiplikation.asm)

c) ⇨ Programm ist mit vorgegebenen Werten getestet.

Aufgabe 2: Matrixmultiplikation in C:

➡ Extra Programm (Datei: prak3.c)

d) / e)

1000 * 1000 Matrix auf ClientSSH3

	multiplyCbyColumn()	**multiplyCbyRow()**
Ohne Parallelisierung		
Sys time	0.004s	0.004s
User time	8.537s	10.169s
Real time	8.567s	10.206s
Mit 2 Threads		
Sys time	0.008s	0.008s
User time	8.529s	8.785s
Real time	5.412s	5.700s
Mit 4 Threads		
Sys time	0.012s	0.000s
User time	13.569s	12.433s
Real time	4.383s	4.239s

Aufgabe 3: Vergleich der Implementierung:

Funktionen	Ohne Optimierung	O1	O2	i368 (ohne Optimierung)	i368 (mit O1)
MultiplyCby Column() ohne OpenMP	r 1m9.231s u 1m9.012s s 0m0.024s	r 1m9.859s u 1m9.640s s 0m0.024s	r 1m10.049s u 1m9.828s s 0m0.028s	r 1m47.374s u 1m47.055s s 0m0.020s	r 1m10.684s u 1m10.456s s 0m0.028s
MultiplyCby Row() ohne OpenMP	r 1m20.591s u1m20.337s s 0m0.028s	r 1m17.647s u 1m17.409s s 0m0.024s	r 1m17.791s u 1m17.565s s 0m0.012s	r 2m35.310s u 2m32.094s s 0m0.028s	r 2m21.315s u 2m20.881s s 0m040s
MultiplyCby Column() mit OpenMP(1)	r 1m8.532s u 1m8.312s s 0m0.028s	r 1m27.526s u 1m27.261s s 0m0.020s	r 1m25.361s u 1m25.085s s 0m0.040s	r 0m30.353s u 1m41.942s s 0m0.024s	r 0m35.092s u 1m58.615s s 0m0.036s
MultiplyCby Row() mit OpenMP(1)	r 1m16.577s u 1m16.309s s 0m0.056s	r 1m32.447s u 1m27.781s s 0m0.052s	r 1m13.742s u 1m13.257s s 0m0.024s	r 0m38.186s u 2m17.025s s 0m0.060s	r 0m47.755s u 2m2.612s s 0m0.068s
MultiplyCby Column() mit OpenMP(2)	r 0m54.710s u 1m42.514s s 0m0.012s	r 0m59.906s u 1m36.298s s 0m0.072s	r 0m44.066s u 1m24.757s s 0m0.032s	r 0m32.666s u 1m34.090s s 0m0.080s	r 0m47.600s u 1m58.963s s 0m0.028s
MultiplyCby Row() mit OpenMP(2)	r 1m7.262s u 1m51.583s s 0m0.048s	r 0m45.932s u 1m9.568s s 0m0.040s	r 0m54.131s u 1m40.538s s 0m0.032s	r 0m38.384s u 1m44.091s s 0m0.036s	r 0m36.382s u 2m16.321s s 0m0.056s
MultiplyCby Column() mit OpenMP(4)	r 0m23.193s u 1m24.113s s 0m0.024s	r 0m39.282s u 1m45.435s s 0m0.016s	r 0m32.596s u1m33.118s s 0m0.028s	r 0m29.815s u 1m33.022s s 0m0.024s	r 0m35.896s u 1m59.547s s 0m0.024s
MultiplyCby Row() mit OpenMP(4)	r 0m54.879s u 3m10.032s s 0m0.032s	r 0m30.721s u 1m37.610 s 0m0.036s	r 0m31.525s u 1m36.070s s 0m0.080s	r 0m29.917s u 1m31.830s s 0m0.044s	r 0m45.183s u 1m56.267s s 0m0.032s
MultiplyCby Double() ohne OpenMP()	r 1m33.511s u 1m33.210s s 0m0.032s	r 1m32.995s u 1m32.702s s 0m0.028s	r 1m32.848s u 1m32.354s s 0m0.048s	r 1m32.434s u 1m32.146s s 0m0.032s	r 1m55.576s u 1m48.319s s 0m0.036s

Alle Zeitmessungen auf ClientSSH3 durchgeführt.
*OpenMP(*) → * = Anzahl der verwendeten Threads; r = real time, u = user time, s = system time*

Auswertung:
Zu Beginn ist zu sagen, dass die Zeiten teilweise starken Schwankungen unterliegen, was mit Auslastung des ClientSSH3 zusammenhängt. Jedoch sind trotzdem klare Zeitunterschiede erkennbar. So arbeitet *multiplyCbyColumn()* gegenüber *multiplyCbyRow()* immer schneller. Dies liegt daran, dass durch die spaltenweise Adressierung der Cache besser genutzt werden kann und nicht so häufig Daten nachgeladen werden müssen.
Bei den Optimierungen mittels OpenMP ist zu erkennen, dass die Anzahl der Threads, wie zu erwarten war, deutliche Auswirkung auf die Laufzeit der einzelnen Implementierungen hat. So wird die Laufzeit bei vier Threads gegenüber der Laufzeit bei zwei Threads halbiert. Dies sollte eigentlich auch vom Sprung von einem auf zwei Threads der Fall sein. Dies konnten wir jedoch nicht ganz festellen. Ein Grund hierfür könnte wie schon die zu Beginn erwähnte Ausnutzung der

Resourcen des ClientSSH3 sein oder die mangelnde Effizienz von zwei Threads bei dieser Problemstellung(Overhead durch Threadnutzung). Die Laufzeitverkürzung entsteht durch die parallele Berechnung der einzelnen Elemente der Ergebnismatrix.
Vergleicht man die beiden Implementierungen mit *multiplyCbyDouble(),* so ist zu erkennen, dass die Implementierung der Gleitkommaberechnung mit doppelter Genauigkeit bei jeder Optimierungsstufe mehr Zeit in Anspruch ninmmt. Dies liegt an den komplexen Rechenoperationen, die bei der Gleitkommaarithmetik verwendet werden müssen (vgl. Hausübung 5). Weiter ist zu erkennen, dass die Compileroptimierungsstufen O1, O2, i386 ohne O1 und i386 mit O1 wenig bis keine Verbesserungen der verschiedenen Implementierungen bringen. Dieser Effekt wurde jedoch auch schon von Herrn Hennes erkannt. Warum dies der Fall ist, weiss bisher scheinbar noch niemand. Wir vermuten allerdings, dass die Gleitkommarechnungen auf der FPU durchgeführt werden und keine weitere Optimierung möglich ist. *(vgl Forum: https://moodle.informatik.tu-darmstadt.de/mod/forum/discuss.php?d=6237).*

Speedup der OpenMP Implementierung:

Die Folgende Tabelle listet die Speedup-Werte auf.
Es wird immer mit **MultiplyCbyColumn() ohne OpenMP** bzw
MultiplyCbyRow() ohne OpenMP verglichen.
Außerdem beziehen sich alle Werte auf die Real-Time.

Funktionen	**Ohne Optimierung**	**O1**	**O2**	**i368 (ohne Optimierung)**	**i368 (mit O1)**
MultiplyCbyColumn() mit OpenMP(1)	**1,01**	**0,8**	**0,82**	**3,54**	**2,01**
MultiplyCbyRow() mit OpenMP(1)	**1,05**	**0,84**	**1,05**	**4,07**	**2,96**
MultiplyCbyColumn() mit OpenMP(2)	**1,27**	**1,17**	**1,59**	**3,29**	**1,48**
MultiplyCbyRow() mit OpenMP(2)	**1,2**	**1,69**	**1,44**	**4,05**	**3,88**
MultiplyCbyColumn() mit OpenMP(4)	**2,98**	**1,78**	**2,15**	**3,6**	**1,97**
MultiplyCbyRow() mit OpenMP(4)	**1,29**	**2,53**	**2,47**	**5,19**	**3,13**

Anmerkung: Der Speedup ist die verhältnismäßige Beschleunigung, in unserem Fall, zu den nicht optimierten Zeitmessungen.
Die Tabelle ist folgendermaßen zu lesen: MultipyCbyColumn mit OMP(4) ist 2,98 fach so schnell, wie die nicht optimierte Variante.

Die erreichten Flops:

Der ClientSSH3 hat laut Datenblatt (siehe Forum)
46.88 GFLOPS <=> 46 880 000 000 FLOPS

Unsere **multiplyCbyColumn()** bzw. **MultiplyCDouble()** hat:

2000*2000x +	=	4000000	+ Operationen
2000*2000*2000*3x *	=	24000000000	* Operationen
2000*2000*2000*4x -	=	32000000000	- Operationen
2000*2000*2000*3x +	=	24000000000	+ Operationen
		80004000000	Gleitkommaoperationen

MultiplyCbyColumn() ohne OpenMP Ohne Optimierung benötigte: 1m9.231s <=> **69.231s**

80004000000 FLOP / 69.231s = 1155609481,30173 FLOPS

1155609481,30173 FLOPS / 46880000000 FLOPS = 0,0246503728946615 =~ 2,5%

Wir haben ca. **2,5%** der maximalen Rechenleistung erreicht.

MultiplyCbyColumn() mit OpenMP(4) Ohne Optimierung benötigte: 0m23.193s <=> **23.193s** (Bestzeit)

80004000000 FLOP / 23.193s = 3449489069,97801 FLOPS

3449489069,97801 FLOPS / 46880000000 FLOPS = 0,073581251492705 =~ 7,3%

Wir haben ca. **7,3%** der maximalen Rechenleistung erreicht.

MultiplyCbyDouble() ohne OpenMP() Mit O2 benötigte: 1m32.848s <=> **92.848s**

80004000000 FLOP / 92.848s = 861666379,458901 FLOPS

861666379,458901 FLOPS / 46880000000 FLOPS = 0,0183802555345329 =~ 1,8%

Wir haben ca. **1,8%** der maximalen Rechenleistung erreicht.