ネットワーク系演習II 高能率計算 1回目レポート

学籍番号：33114135
吉田樹

使用環境情報

- CPU: AMD Ryzen7 7700X 4.5GHz （ターボブースト5.4GHz） 8コア16スレッド
- メモリ: 64GB
- OS: Ubuntu 23.04
- コンパイラ: g++

課題 1

ソースコードの計測部分前後にタイマー関数を追記して実行速度を計測した.

なんどか実行した結果の実行時間の例を以下に示す.

exercise 1: loop = 10, size = 30

time	ms
time 0	0.00094
time 1	0.00016
time 2	0.00016
time 3	0.00017
time 4	0.00054
time 5	0.00015
time 6	0.00013
time 7	0.0001
time 8	0.00013
time 9	0.00015
time avg	0.000187778

考察：計算時間にはばらつきが観測できたが，おおむね一回目の計測時間が遅かった.

課題 2

コンパイラオプションをO0 Ofastに変えてプログラムをコンパイルした.

結果は以下のようになった.

exercise 2: loop = 100, size = 512

option	time[ms]	-mtune=native
-O0	17.8489	$18.082s$
-O1	3.60307	3.69401
-O2	3.72008	3.75577
-O3	0.772318	$0.66508s$
-Os	4.16024	4.41665
-Ofast	0.722098	0.730828
-Og	5.29588	5.07572

またコンパイル時間を下記に示す.

	-O0	-O1	-O2	-O3	-Os	-Ofast	-Og
real	$1.157s$	$1.748s$	$2.168s$	$2.604s$	$1.980s$	$2.607s$	$1.415s$
user	$1.031s$	$1.634s$	$1.984s$	$2.484s$	$1.868s$	$2.466s$	$1.309s$
sys	$0.110s$	$0.107s$	$0.177s$	$0.121s$	$0.113s$	$0.142s$	$0.108s$

考察：Ofastが最も高速に動作した．またコンパイル時間も最も長かった．O0とO1の実行速度の差は顕著だが，O1, O2の差はごくわずかであり，優位な差を計測するにはより大きなサイズの行列で計測する必要があると考えられる．一方，O3では再び大きく高速化ができており，O3で適用される最適化は大きな効果をもつことがわかる．コンパイル時間はO0 〜 O3にかけて徐々に長くなっている．実行速度の改善度によらずコンパイルにかかる時間は一定の時間ずつ増えている． そのほか，

```
-march=native -mtune=native
```

のオプションについても検証した結果，

```
-march=native
```

は指定しないと次のようなコンパイルエラーが出た．

```
/usr/lib/gcc/x86_64-linux-gnu/12/include/avxintrin.h:885:1: error: inlining failed
in call to ‘always_inline’ ‘void _mm256_store_ps(float*, __m256)’: target specific
option mismatch
  885 | _mm256_store_ps (float *__P, __m256 __A)
      | ^~~~~~~~~~~~~~
/usr/lib/gcc/x86_64-linux-gnu/12/include/avxintrin.h:1244:1: error: inlining failed
in call to ‘always_inline’ ‘__m256 _mm256_setzero_ps()’: target specific option
mismatch
 1244 | _mm256_setzero_ps (void)
      | ^~~~~~~~~~~~~~~~
```

コンパイラはデフォルトでAVX命令を行うバイナリファイルの生成を避けるため，AVX命令セットを含む関数でありを用いる場合は適切にコンパイルオプションを設定して，AVX命令セットを含むバイナリファイルを生成するようにしなければならない．これを設定すると，AVX命令を明示的に用いる場所以外についても高速化される可能性がある．(e.g., 浮動小数点演算)また，AVX命令セットを用いるため，AVX命令セットに対応していないアーキテクチャでの実行は不可能になる．また，今回のエラーではコンパイルオプションとして

```
-mavx2 -mfma
```

を指定することでもエラーを回避することができた．

```
-march=native
```

での指定はそのアーキテクチャへの最適化になるため，そのままのバイナリだと他のアーキテクチャでは使用できない場合がある．

```
-mtune=native
```

については，GCC公式ドキュメントを参照すると次のような記述がある．

```
Tune to cpu-type everything applicable about the generated code, except for the ABI
and the set of available instructions. While picking a specific cpu-type schedules
things appropriately for that particular chip, the compiler does not generate any
code that cannot run on the default machine type unless you use a -march=cpu-type
option.
```

CPUの特性に合わせた最適化を行うがCPUの特定の命令セットは用いないようにバイナリを生成する．つまり，バイナリをそのまま別のアーキテクチャでも用いることができる．しかし，最大限最適化が適用されるのはそのバイナリが作成されたCPUで動作するときである．実行時間はO3とOgのときしか改善が見られなかった．その他の場合はむしろ遅くなってしまっている．

課題３

以下のようにコードを書き換えた．

書き換え前

```
const float v = x.data[i];
ret.data[i] = 3.f * v * v * v * v * v * v * v
    + 3.f * v * v * v * v * v * v
    + 3.f * v * v * v * v * v
    + 3.f;
```

書き換え後

```
const float v = x.data[i];
ret.data[i] = 3.f * (v * v * v * (v * (v *(v + 1) + 1) + 1) + 1) + 1);
```

method	time [ms]
before	0.0156979
after	0.0113328

```
info:
default parameter: default_loop = 100000, default_size = 64
diff from ans: 1.63106e-13
```

考察：実行結果は0.004sほどの高速化ができている．しかし計算誤差が1.63106e-13発生しており，floatの有効桁数は10進数で7桁程度であり，それ以上に桁数が増える場合は誤差が大きくなる．

課題４

method	time [ms]
inline	0.00918588
precomp	0.00477012

```
info:
default parameter: default_loop = 10000, default_size = 64
diff from ans: 0
```

課題４の結果を比較すると，事前計算がある場合の実行時間は事前計算なしの場合と比較して，半分以下になっていることがわかる．このことから何度も利用する可能性のある定数は事前に計算しておいて，演算回数を減らすことは高速化に大きく寄与するということがわかる．

課題５

method	time[ms]
div 2lp	0.0223908
mul 2lp	0.0225196
div 1lp	0.0164302
mul 2lp	0.0157203

```
info:
default parameter: default_loop = 10000, default_size = 128
diff from ans: 3.31412e-13
```

これは最適化を切って行った場合の結果である．これを比較するとループをつぶした方は少しだけ高速化されているが，ループを潰していない方はかえって遅くなっている．これらのことから，最適化がきかない場合ではそこまで大きく計算時間の改善には至らないと考えられる． 一方次に示すのは最適化を-O2で行ったものである．

method	time[ms]
div 2lp	0.00909448
mul 2lp	0.0035838
div 1lp	0.00895762
mul 2lp	0.00317331

```
info:
default parameter: default_loop = 10000, default_size = 128
diff from ans: 3.33241e-13
```

こちらの結果を比較すると除算を乗算に変換したものでは3倍程度の高速化に成功している．このことからわかることは, 乗算のほうが最適化が効きやすく, そういった側面も吟味して考えると乗算のほうが除算よりも高速になりやすいとわかる． なお，最適化レベルが高い方が計算誤差もごく僅かであるが大きくなっている．

課題６

method	time [ms]
div x2	0.00462292
div x1	0.00234152

```
info:
default parameter: default_loop = 10000, default_size = 64
diff from ans: 3.62596e-12
```

除算を減らしたほうが計算時間がおよそ半分になっていた．このことから計算コストは乗算 < 除算であることがわかる．計算誤差は実際の数字計算と比較した場合除算が少ないほうがより正確である可能性が高い．

課題 7

exercise 7: loop = 10000, size = 64

method	time [ms]
pow 2	0.0303039
mul 2	0.00500175
pow 3	0.0303129
mul 3	0.00570386
pow 4	0.0303656
mul 4	0.00717103
pow 32	0.030364
mul 32	0.0376619

```
info:
default parameter: default_loop = 10000, default_size = 64
diff from ans: 1.52389e-31
```

小さいnではpowを用いらない方が6倍程度高速に計算することができている．一方nが大きくなると, 例えば$n = 32$の場合ではmulのほうが計算は遅くなっている．しかし, powの方はnが小さい場合と比較しても変化していない．ここでpowのみ$n = 1024$で実行した時間を計測すると，0.0306866msであった．これはpowがnに比例しない計算時間であることを示している．また，当然だが，単純な乗算を行う場合は計算時間は$O(n)$である．そのため，nが大きくなるとpowのほうが高速になると考えられる． 最適化オプションを02やO3に変更した場合は，$n = 2$のケースではmulの計算時間とpowの計算時間に大きな差は認められなかった．また，$n = 32$程度ではmulの方が高速であった．このことから，コンパイラは$n = 2$のpow計算を乗算に変換していると考えられる．それ以外のnでは大きな計算時間の改善には至っていなかった．

課題8

exercise 8: loop = 1000, size = 2048

method	time [ms]
8U	0.0930131
8S	0.093949
16S	1.05321
32S	1.45627
32F	1.26489
64F	2.66038

```
info:
default parameter: default_loop = 1000, default_size = 1024
diff  8S from 8U: 7448.93
diff 16S from 8U: 0
diff 32S from 8U: 0
diff 32F from 8U: 0
diff 64F from 8U: 0
```

課題８は行列のサイズを大きくしたほうがわかりやすく結果がでたため，サイズを2048とした． 基本的には数字が少ないbit数で表現されている方が計算がはやく，また，浮動小数点型のほうが計算がはやい．1bitと2bitの差はそこまで大きくはないが，2bitと4bitの差は大きく，計算時間が1.4倍になっている．また，size = 4096とすると，更に計算時間の差は開いて，次の表のようになった．

method	time [ms]
8U	0.937347
8S	0.970301
16S	6.17348
32S	7.88354
32F	7.83867
64F	12.2795

課題２２

画像の張り付けサンプル

図ｘ：シングルスレッドとマルチスレッドのルーフライン.

参考までに，CSEは1.3TFLOPSくらい出ます（試すのは絶対に夜中で．）．

．．．

課題２９

画像処理課題１

画像処理課題２

1回目の課題はここまで． 画像処理の共通の課題である上記１，２を忘れずにやること． これ以降の課題は2回目のレポートです.