SR16000 モデルM1、XM1

目次

SR16000 モデルM1

SR16000/M1システムの1ノードでの実行に関しての記述です。

構成の概略は以下の様になっています。

プロセッサ:power7
周波数:3.83GHz
CPUコア数 32(物理的),64(論理的)
理論最大性能　980.48 GFLOPs
メモリ容量 256GB
メモリアーキテクチャー　NUMA,(16論理コア単位でflat)
SIMD(Single Instruction Multiple Data)を
サポートするVSX機構付き
L3キャッシュ　On-Chip 32MB/8コア
演算器/物理コア　乗加算器4つ

また,メモリアクセス性能をみるのにSR16000/XM1と比較しています
SR16000/xm1は周波数が3.3GHzで他はSR16000/M1 と同じです。

演算性能だけみれば,SR16000/M1 1ノードはSR16000/xm1の１６％性能向上版ともいえます。

1. 行列積計算

C=AB の計算のプログラムには,外積型,内積型があります。

外積型

```
   DO 30 K=1,NN
   DO 20 J=1,NN
   DO 10 I=1,NN
   C(I,J)=C(I,J)+A(I,K)*B(K,J)
10  CONTINUE
20  CONTINUE
30  CONTINUE
```

内積型

```
   DO 30 J=1,NN
   DO 20 I=1,NN
    S=0.0D0
     DO 10 K=1,NN
     S=S+A(I,K)*B(K,J)
10 CONTINUE
     C(I,J)=S
20 CONTINUE
30 CONTINUE
```

行列積計算で用いられる用語にM段N列というのがあります。
これは,上の例では

DO 30　K=1,NN,N:DO 30 J=1,NN,N のアンローリング数をN列
DO 20　J=1,NN,M:DO 20 I=1,NN,M のアンローリング数をM段

の事を言います。

講習会では,キャツシュチューニングを意識した4段6列
内積型のプログラムが示されました。

```
      iu1=6
      iu2=4
      ib1=120
      ib2=120
      ib3=960
*poption indep(c)
      do kk=1,nn,ib1
      do jj=1,nn,ib2
      do ii=1,nn,ib3
      do k=kk,min(kk+ib1-1,nn),iu1
      do j=jj,min(jj+ib2-1,nn),iu2
      t11=c(j,k)
         :
      t64=c(j+3,k+5)
      do i=ii,min(ii+ib3-1,nn)
      t11=t11+a(i,j)*b(i,k)
      :
      t64=t64+a(I,j+3)*b(I,k+5)
      end do
      c(j,k)=t11
      :
      c(j+3,k+5)=t64
      end do
      end do
      end do
      end do
      end do
```

<= ここに注目してください。

1ノード8スレッド/8core;コンパイルオプション –Os –parallel
NN=2880 の場合の実行結果は以下の様になりました。
*poption indep(c) 行のあるなし(指定あり,指定なし)の実行結果に大きな差がでました。

	実行時間(秒)	性能(GFLOPs)
指定あり	0.263	181.6
指定なし	10.033	4.8

コンパイルリストでは*poption indep(c) の指定あり,なしともに並列化はされています。

指定あり
*end of compilation : MAIN
*end of compilation : _parallel_func_1_MAIN
*end of compilation : _parallel_func_2_MAIN
*end of compilation : _parallel_func_3_MAIN

例示された箇所が
DO kk=1,nn,ib1 で並列化されました。

指定なし

```
*program name = MAIN
*end of compilation : MAIN
*end of compilation : _parallel_func_1_MAIN
*end of compilation : _parallel_func_2_MAIN
*end of compilation : _parallel_func_3_MAIN
*end of compilation : _parallel_func_4_MAIN
*end of compilation : _parallel_func_5_MAIN
*end of compilation : _parallel_func_6_MAIN
```

(1)
(2)
(3)
(4)

例示された箇所が

(1) do kk=1,nn,ib1 で並列化されました。
(2) do jj=1,nn,ib2 で並列化されました。
(3) do ii=1,nn,ib3 で並列化されました。
(4) do k=kk,min(kk+ib1-1,nn),iu1 で並列化されました。

すなわち,*poption indep(c) の指定あり,なしでは一般に
言われる並列化率は同じです。

このため,実行時間の大きな差は,並列化回数の差による
ものと考えられます。

並列化回数の差は, $2880 \times 2880 \times 3 \times 4 = 99532800$回

となっています。

2. SMP同期のオーバーヘッド時間

1回あたりの並列化オーバーヘッドを以下のプログラムで測定しました。

テストプログラム

```
*poption noparallel
*soption unroll(1)
      do 20 j=1,loop
*soption nosimd
*poption parallel
     do 10 i=1,n
     c(i)=a(i)+b(i)
 10  continue
 20  continue
```

Element parallelizing rate : (TOTAL)/(Max * TDs)
CPU time : 98.26[%] = 6.453/(0.820984*8)
Flop : 99.70[%] = 1046501134/(131210596*8)

6.453*(1-0.9826)=0.122sec 。1,000,000 回並列化ですので,1回あたりのオーバーヘッドは122 nsec と推定されます。

行列積計算のプログラムでは,

(10.033-0.263) sec/99532800=98nsec です。

目安として,smp同期時間のオーバーヘッドは約100nsec ということになります。

この段階では,原因究明,対策が優先で,122,98nsecの差にこだわる必要はありません。

3. MPI同期のオーバーヘッド時間
以下の2つのプログラムの差分で測定しています。

```
      case1
sec1 = mpi_wtime()
call mpi_barrier(mpi_comm_world,ierr)
do i=1,1000000
call mpi_barrier(mpi_comm_world,ierr)
do i1=1,n/npe
xx=x30(i1+n*id/npe)
end do
call mpi_barrier(mpi_comm_world,ierr)
end do
call mpi_barrier(mpi_comm_world,ierr)
sec2 = mpi_wtime() - sec1
```

```
      case2
   sec1 = mpi_wtime()
   call mpi_barrier(mpi_comm_world,ierr)
*soption unroll(1)     <=アンローリングの最適化を抑止
   do i=1,1000000
   do i1=1,n/npe
   xx=x30(i1+n*id/npe)
   end do
   end do
   call mpi_barrier(mpi_comm_world,ierr)
   sec2 = mpi_wtime() - sec1
```

測定結果

MPI数	case1（秒）	case2（秒）	同期時間 nsec/回
1	0.877	0.231	323
2	1.846	0.118	864
4	2.036	0.062	987
8	2.664	0.034	1315
16	3.565	0.023	1771
32	5.344	0.009	2667.5
64	10.412	0.012	5200

同期のオーバーヘッド時間は,最大5マイクロ秒ですので,smpオーバーヘッドの数十倍となります。

以上から1ノードで実行する場合は,SMP並列で実行するのが良い事がわかります。

4.性能モニターのサブルーチンコールのオーバーヘッド時間

以下のプログラムで測定しています。

```
C
      s=0.0d0
      a=1.0d0
      b=2.0d0
        :
      do i=1,1000000
      call sub(s,a,b)
      end do
        :
      write(6,*) 's=',s
        :
      subroutine sub(s,a,b)
      implicit real*8 (a-h,o-z)
      s=s+a+b
      return
      end
```

性能モニター使用オプション指定の場合の結果

```
 s=   3000000.00000000000
 elapse=   3.998645067214965 82       sec
```

性能モニター使用オプション指定なしの場合の結果

```
 s=   3000000.00000000000
 elapse=  0.10959863662719726 6E-001  sec
```

実行時間に明らかに大きな差がでています。この結果から
サブルーチンコール1回あたりの性能モニターのオーバーヘッド
時間は,
(3.998645-0.0109598) sec/1,000,000=3.987685 micro sec
より、約4micro sec となります。

例えば,あるサブルーチンの演算量が100FLOP,呼ばれる回数が
100億回とします。するとこのサブルーチンの総演算量は
1000GFLOPとなり,1ノードで実行すれば,数十秒(1分以内)で終了
するでしょう。

ところが,性能モニターを使用すると,
40,000秒≒11時間強(約半日)かかる事になります。

このため,モニターリングのオーバーヘッド時間を測定するか,
運営サイトに質問するのが良いでしょう。

もし,サブルーチンのインライン化をして実行してくださいと
いう回答が来れば,マニュアル等で調べる手間を省かない
様にするのが良いでしょう。

5. メモリアクセス性能

以下の簡単な5つのプログラムで測定しています。

(MMAX=8193)

2D Copy

```
do j = 1,MMAX
do i = 1,MMAX
r(i,j)=p(i,j)
enddo
enddo
```

2D Scale

```
do j = 1,MMAX
do i = 1,MMAX
p(i,j)=scalar*q(i,j)
enddo
enddo
```

2D Add

```
do   j = 1,MMAX
  do i = 1,MMAX
  q(i,j)=p(i,j)+r(i,j)
  enddo
  enddo
```

2D Triad

```
do j=1,MMAX
do i=1,MMAX
r(i,j) = p(i,j) + scalar*q(i,j)
enddo
enddo
```

Transpose

```
do j=1,MMAX
do i=1,MMAX
r(i,j)=p(j,i)
enddo
enddo
```

SR16000/XM1とSR16000/M1の2機種で測定していますが,
約3倍の差があり,演算性能の差16%とは大きな差があります。
特に,Transpose　64スレッドでの差が顕著です。

メモリバンド幅性能測定結果(MB/sec)

SR16000/XM1 MB/sec

function	1スレッド	2スレッド	4スレッド	8スレッド	16スレッド	32スレッド	64スレッド
2D Copy	5070	8004	13786	27134	43057	68565	62532
2D Scale	4682	7982	13707	27011	41294	61917	58637
2D ADD	6563	11054	16604	32834	48544	78820	70376
2D Triad	7307	12097	16954	33525	49752	77751	70680
Transpose	426	725	1470	2649	4883	3801	2866

SR16000/M1 MB/sec

function	1スレッド	2スレッド	4スレッド	8スレッド	16スレッド	32スレッド	64スレッド
2D Copy	11661	23356	46292	92051	165536	138902	148588
2D Scale	12528	25012	49218	97630	170661	140657	163474
2D ADD	12918	25797	49839	97796	176648	151006	174514
2D Triad	12680	25216	48697	96034	174808	149709	171588
Transpose	1052	2055	4072	7971	15921	16231	30540

QCD(量子色力学)計算の主要な処理に,計算機から見た場合アドレス非連続な複素数変数の行列積計算があります。
この処理を性能テスト用に切り出したプログラムを SR16000/M1,SR16000/XM1 で実行した結果は以下の様になりました。

サイズ条件　NX=16,NY=16,NZ=64,NT=16
コンパイルオプション　-Oss
実行環境　64SMP/32CORE

チューニングの内容は以下の2つです。

(1) 複素変数を実数変数化し,コンパイラ最適化とVSX(SIMD命令の適用）機能の適用を容易にしました。
(2)mod関数をIAND関数に置き換え整数演算の高速化と使用する整数用レジスタの数を削減しました。

実行結果

GFLOPs 一覧表

	オリジナルソース	チューニングソース
SR16000/M1	117.690	216.408
Xm1	29.966	79.118

メモリアクセス性能測定結果とよく対応しており,
チューニング効果は,SR16000/XM1に比べてSR16000/M1 が小さくなっています。この事から,

(1)QCDというプログラムは演算性能よりもメモリアクセス性能に大きく依存する。
(2)SR16000/M1 でのチューニングは更なる検討が今後の課題となる。(今後適時記述予定)

という事がわかります。

6. パターンマッチングの利用

行列積計算の様にアンローリングの効果の大きいものがあります。ただ,プログラムのメンテナンスや,コーデイングの手間を考えた場合,基本通りにコーデイングしたものが高い性能を出してくれるコンパイラが望ましいものです。

例えば,サイズN=2377の行列積計算をコーデイングする場合,アンローリングによるコーデイング数の増加に加え,端数部分の処理も追加しなければなりません。また計算機機種によりアンローリング数の最適値が変わる事があり,その都度プログラム修正の手間がかかります。

アンローリングもしない基本どうりのコーデイングで高い性能が得られればその手間も省けますので,運営サイトに問合わせるのも良いでしょう。

SR16000／M1　の場合の例を以下に示します。

行列積計算(アンローリングとの比較)

外積型

```
    DO   10 K=1,NN
     DO 10 J=1,NN
     DO 10 I=1,NN
     A(I,J)=A(I,J)+B(I,K)*C(K,J)
10 CONTINUE
```

内積型

```
    DO 10 J=1,NN
     DO 20 I=1,NN
     S=0.0D0
     DO 30 K=1,NN
     S=S+B(I,K)*C(K,J)
30 CONTINUE
      A(I,J)=S
20 CONTINUE
10 CONTINUE
```

N=3072　–Oss でコンパイル

		外積型	内積型
アンローリングなし	smt on 64smp/32core	334.6Gflops	40.7Gflops
	smt off 32smp/32core	182.8Gflops	22.9Gflops
アンローリングあり (四段四列)	smt on 64smp/32core	253.7Gflops	165.8Gflops
	smt off 32smp/32core	106.5Gflops	107.6Gflops

尚,この例は,実数型倍精度変数ですが,複素数型倍精度変数でも,パターンマッチングが適用されています。

7. ストラッセンの行列積

演算量の削減を考えたアルゴリズムとして,ストラッセンの行列積,FFTなどがあります。これらの演算の特徴として,元の演算(行列積計算,フーリエ変換計算)は乗算と加算が同数とバランスが良く実行性能効率は高く,アルゴリズム変更後は,演算量は削減されますが,乗算と加算の演算比率のバランスが悪くなり,また中間結果の格納のためメモリアクセス負荷が高まるため,実行時間は短縮されますが,実行性能効率は低くなる事が挙げられます。

実行性能効率より,実行時間短縮を優先させるのが良いでしょう。以下例としてFFT,ストラッセンの行列積の演算量を示しました。

一次元ＦＦＴ $n = 2^k$

乗算 : $\frac{1}{2} n \log_2 n$

加減算 : $n \log_2 n$

ストラッセンの行列積

A(k) : 加減算回数

M(k): 乗算回数

$$N = 2^k \times N_0 \quad k : 段数 (k \geq 1)$$

$$M(0) = {N_0}^3, A(0) = {N_0}^3$$

$$J = 1,2,....,k$$

$$M(J) = 7 \times M(J-1)$$

$$A(J) = 7 \times A(J-1) + 15 \times (\frac{1}{2} 2^J N_0)^2$$

行列積の演算量 $2N^3$ をストラッセンの行列積の実行時間で割った見かけの性能が理論最大性能を上回るサイズ,段数により,その計算機のメモリアクセス性能を判断する事が出来ます。サイズが小さく,段数が少なくて済むものほどメモリアクセス性能が高いと言えます。

S16000/M1 での実測は以下の様になっています。
N=61440,段数４段,コンパイルオプションは-Oss

使用コア数	実測結果 (GFLOPs)	理論最大性能 (GFLOPs)
8	307.465	245.12
16	589.953	490.24
32	1027.31	980.48

8. 基本演算性能データ

アルゴリズムによっては,乗算と加減算のバランスが悪かったり,除算が多いものなどがあります。一般にアナウンスされる計算機の性能は,同時に動作する浮動小数点演算器(加算器と乗算器のみ)の数をもとに算出されています。このため,除算やべき乗計算,数学関数などの性能を測定するか,運営サイトに問合わせて確認する必要があります。

SR16000/M1 では、性能モニターで測定した演算数は以下の様になっています。

関数	R*8 演算数	R*16 演算数	C*16 演算数	C*32 演算数
SQRT	4	67	26	278
EXP	29	118	90	543
LOG	26	170	133	454
SIN	42	187	154	789
COS	42	196	155	793
TAN	77	179	―――――	―――――
べき乗	98	298	263	1081
絶対値	―――――	―――――	16	106

ここで、倍精度実数SQRTの演算数は４なので
倍精度複素数$z = a + b*i$では絶対値計算$\sqrt{a^2 + b^2}$
は演算数７で済むのではと考えられますが,値は
16になっています。これは、提供されている複素数
絶対値計算は精度を考慮し,a, bの値により場合わけ
を行う事によります。もしそのような心配がない
場合は、$16/7 \geq 2$倍以上の性能向上が出来る事になります。

また,a,b,c を倍精度実数配列,IN 整数配列とした場合
以下のような事がありますので注意してください。

```
Case1

DO I=1,N
C(i)=a(i)**b(i)
End do

Case2

DO i=1,N
C(i)=a(i)**IN(i)
End do
```

B(i)にはすべて5．0d0,IN(i)にはすべて5が入っていいる場合の1要素あたりの浮動小数点演算数は

```
case1    98
case2     4
```

となります。

選択した16個の基本演算DOループ

Cで始まる変数は複素数変数

TEST1

```
  DO 10 I=1,N
  RV(I)=RV2(I)+RV3(I)
10 CONTINUE
```

TEST2

```
  DO 10 I=1,N
  RV(I)=RV2(I)*RV3(I)
10 CONTINUE
```

TEST3

```
  DO 10 I=1,N
  RV(I)=RV2(I)/RV3(I)
10 CONTINUE
```

TEST4

```
  DO 10 I=1,N
  CV(I)=CV2(I)+CV3(I)
10 CONTINUE
```

TEST5

```
  DO 10 I=1,N
  CV(I)=CV2(I)*CV3(I)
10 CONTINUE
```

TEST6

```
  DO 10 I=1,N
  CV(I)=CV2(I)/CV3(I)
10 CONTINUE
```

TEST7

```
  T=0.0D0
  DO 10 I=1,N
  T=T+RV2(I)
10 CONTINUE
```

test8

```
  T=0.0D0
  DO 10 I=1,N
  T=T+RV2(I)*RV3(I)
10 CONTINUE
```

test9

```
      DO 10 I=1,N
      RV(I)=SQRT(RV2(I))
   10 CONTINUE
```

test10

```
      DO 10 I=1,N
      RV(I)=SIN(RV2(I))
   10 CONTINUE
```

test11

```
      DO 10 I=1,N
      RV(I)=COS(RV2(I))
   10 CONTINUE
```

test12

```
      DO 10 I=1,N
      RV(I)=TAN(RV2(I))
   10 CONTINUE
```

test13

```
      DO 10 I=1,N
      RV(I)=EXP(RV2(I))
   10 CONTINUE
```

test14

```
      DO 10 I=1,N
      RV(I)=LOG(RV2(I))
   10 CONTINUE
```

test15

```
      DO 10 I=1,N
      RV(I)=ABS(CV2(I))
   10 CONTINUE
```

test16

```
      S1=0.0D0
      S2=0.0D0
      DO 10 I=1,N
      S1=S1+(RV2(I)-RV3(I))**2
      S2=S2+(RV2(I)+RV3(I))**2
   10 CONTINUE
```

9. 連立一次方程式

直接法ではTOP500で性能評価される事で有名ですが,
3次元問題などになると,メモリ容量,演算量が
非常に大きくなり,実用上使用される事がなく,反復法
が良く使用されています。ただここ10年来,精度上の問題
で一部の次元の小さい方程式を直接法で解く事も行われる
様になってきています。(構造計算など)

また,連立一次方程式の直接法は20年前にはドンガラ
レポートとして実用ベースの次元数100,1000の各種計算
機の性能比較が行われていました。

100元　ソース修正不可 : ハードウエア及びコンパイラの
最適化能力こみの評価

1000元　ソース修正可:上記に加えチューニング能力も
含めた評価

1000元オーダーあたりが実用的に使用されるサイズ
で,この部分を含んだ箇所を並列化するということ
が多いため,SR16000/M1　では1000元,1コア実行での
性能を評価しています。

コンパイルオプション　-Oss –noparallel

S16000/M1 1コア(論理最大性能 30.64GFLOPs) 実行結果

次元数	N=1000	
アンローリング		
段数	列数	性能（GFLOPs）
1	1	1.967
2	2	3.447
3	3	6.203
4	4	8.055
5	5	6.925
6	6	7.117
7	7	5.881
8	8	6.537
10	10	5.466
16	16	3.712
20	20	1.384

実行結果からも推測されるとは思いますが,SR16000/M1以外の多くの計算機でも段数,列数4~8あたりが最適となっています。

10. 反復法

固有値計算など多岐に亘って反復法が使用されています。テストとしては対称行列にはCG法,PCG法，非対称行列にはCGS法,BCG法で最適化の影響などをチェックしています。

Ax=b　A:行列,x,bはベクトル

Aを定め,解がすべて1となる様にbを設定しています。

反復の初期値はx(i)=0.0d0 (i=1,2,....,n)としています。

また,SR16000/M1で実行したときの収束状況を一覧にして次ページに記載しています。

A=

$$\begin{bmatrix} -1 & 2+R & -1 & 0 & .... & 0 & 0 & 0 \\ 2+R & -1 & 0 & .... & 0 & 0 & 0 \\ -1 & 2+R & -1 & .... & 0 & 0 & 0 \\ 0 & -1 & 2+R & .... & 0 & 0 & 0 \\ : & : & : & : & : & : \\ 0 & 0 & 0 & .... & 2+R & -1 & 0 \\ 0 & 0 & 0 & .... & -1 & 2+R & -1 \\ 0 & 0 & 0 & .... & 0 & -1 & 2+R \end{bmatrix}$$

対称

A=

$$\begin{bmatrix} -1 & 2+R & -1 & 0 & .... & 0 & 0 & 0 \\ 2+R & -1 & 0 & .... & 0 & 0 & 0 \\ -1 & 2+R & -1 & .... & 0 & 0 & 0 \\ 0 & -1 & 2+R & .... & 0 & 0 & 0 \\ : & : & : & : & : & : \\ 0 & 0 & 0 & .... & 2+R & -1 & 0 \\ 0 & 0 & 0 & .... & -1 & 2+R & -1 \\ 0 & 0 & 0 & .... & 0 & -2 & 2+R \end{bmatrix}$$

非対称

収束状況一覧表

$n = 1{,}000{,}000$　　収束判定値 $= 10^{-12}$

(1)対称行列

反復回数と精度

R	CG反復回数	CG精度(最大誤差)
10^{-4}	2510	3.439×10^{-11}
10^{-3}	823	1.432×10^{-11}
10^{-2}	254	2.518×10^{-11}
10^{-1}	77	6.575×10^{-11}
10^{-0}	24	1.517×10^{-10}

R	PCG反復回数	PCG精度(最大誤差)
10^{-4}	収束せず(5000回)	――――
10^{-3}	3601	5.054×10^{-13}
10^{-2}	361	3.874×10^{-13}
10^{-1}	80	3.711×10^{-11}
10^{-0}	24	1.512×10^{-10}

(2)非対称行列

反復回数と精度

R	CGS反復回数	CGS精度(最大誤差)
10^{-5}	3805	2.101×10^{-10}
10^{-4}	1291	3.624×10^{-11}
10^{-3}	430	8.834×10^{-12}
10^{-2}	135	1.038×10^{-11}
10^{-1}	42	1.255×10^{-11}
10^{-0}	13	4.428×10^{-11}

R	BCG反復回数	BCG精度(最大誤差)
10^{-5}	収束せず(5000回)	――――
10^{-4}	収束せず(5000回)	――――
10^{-3}	収束せず(5000回)	――――
10^{-2}	315	1.384×10^{-11}
10^{-1}	80	8.396×10^{-11}
10^{-0}	24	1.553×10^{-11}

11. SIMD(Single Instruction Multiple Data)

高速化手法として、以前からSIMDという事が言われていましたが、最近とみに使われる様になりました。
SIMDの効果を考える場合、つぎの2つのケースがあります。

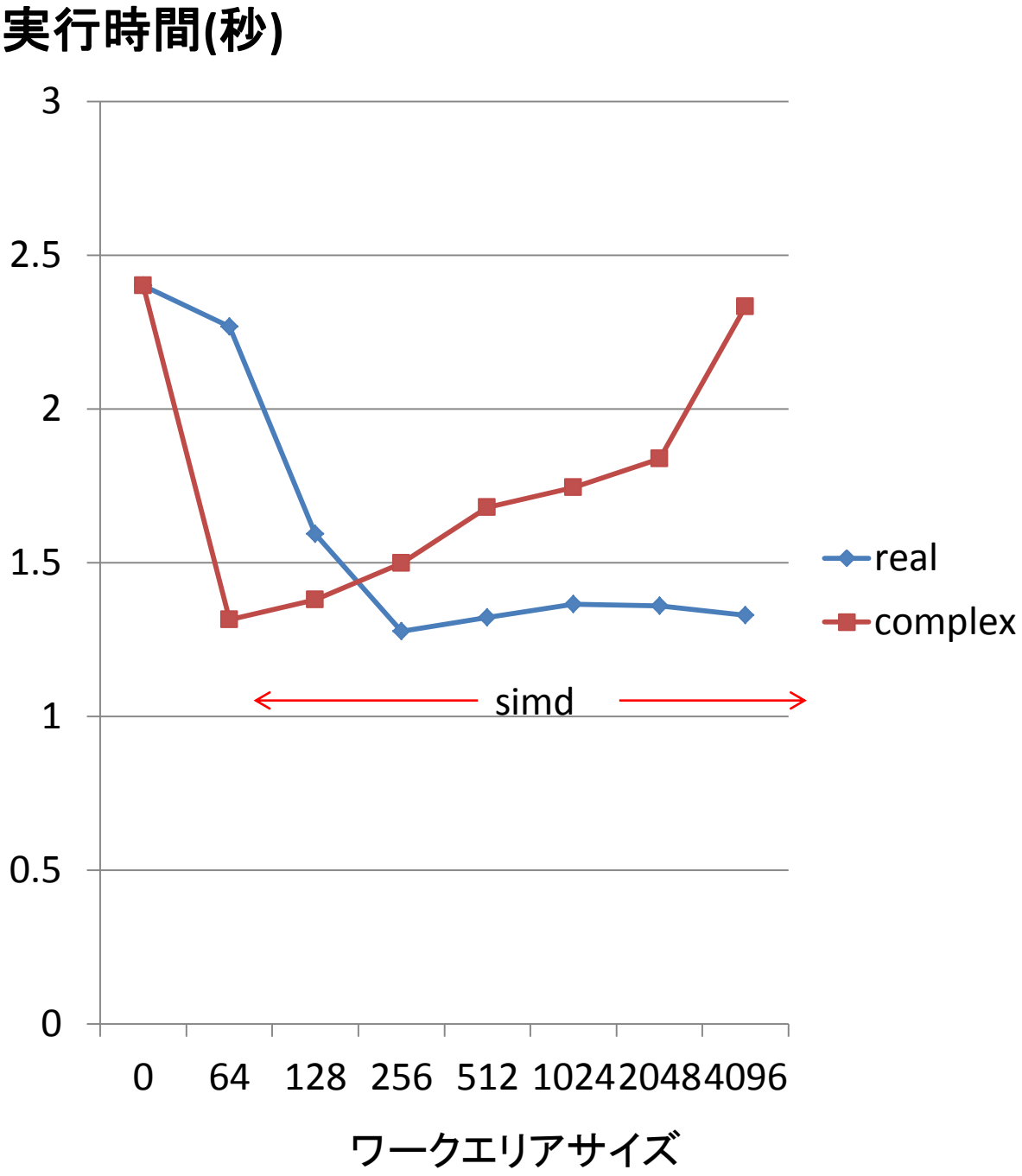
演算	GPU		SR16000/M1	
	SMP	SIMD	SMP	SIMD
倍精度実数乗算	OK	OK	OK	OK
倍精度複素数乗算	OK	OK	OK	NG
4倍精度実数加算	OK	OK	OK	NG
4倍精度実数乗算	OK	OK	OK	NG

OK:適用可　NG:適用不可

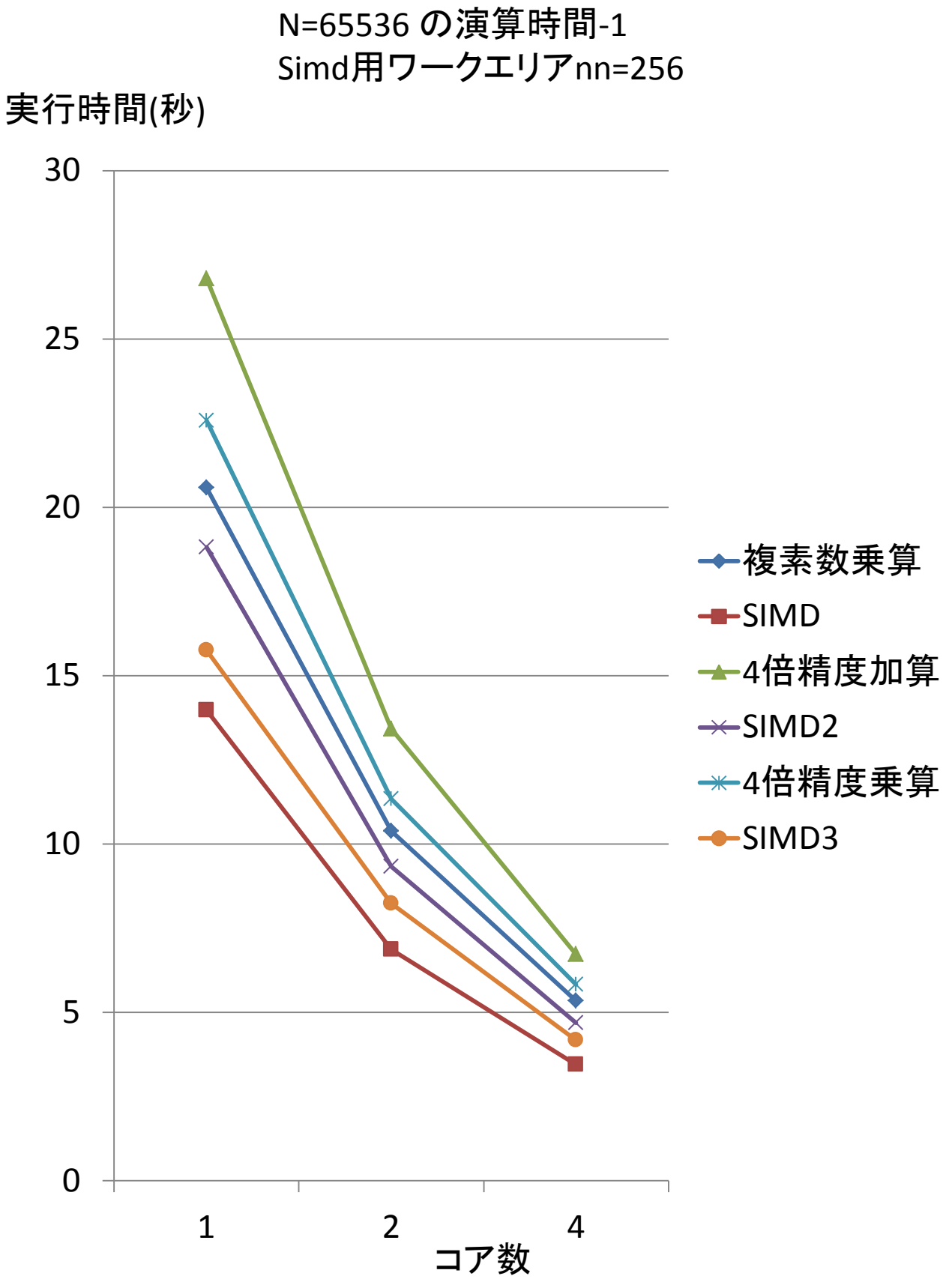
SR16000/M1では,キャッシュL1D 32KB/コア,L2 256KB/コア、L3　32MB/8コアを考慮して、実数型倍精度変数ワークエリアを使用して、複素数型倍精度変数の乗算、
実数型4倍精度変数の加減算、乗算におけるSIMD効果のテストを行いました。
(1)ワークエリアと取り方と最適なサイズの調査
(2)並列実行時も含めた効果の調査
の順で実施したテスト結果は以下の様になりました。

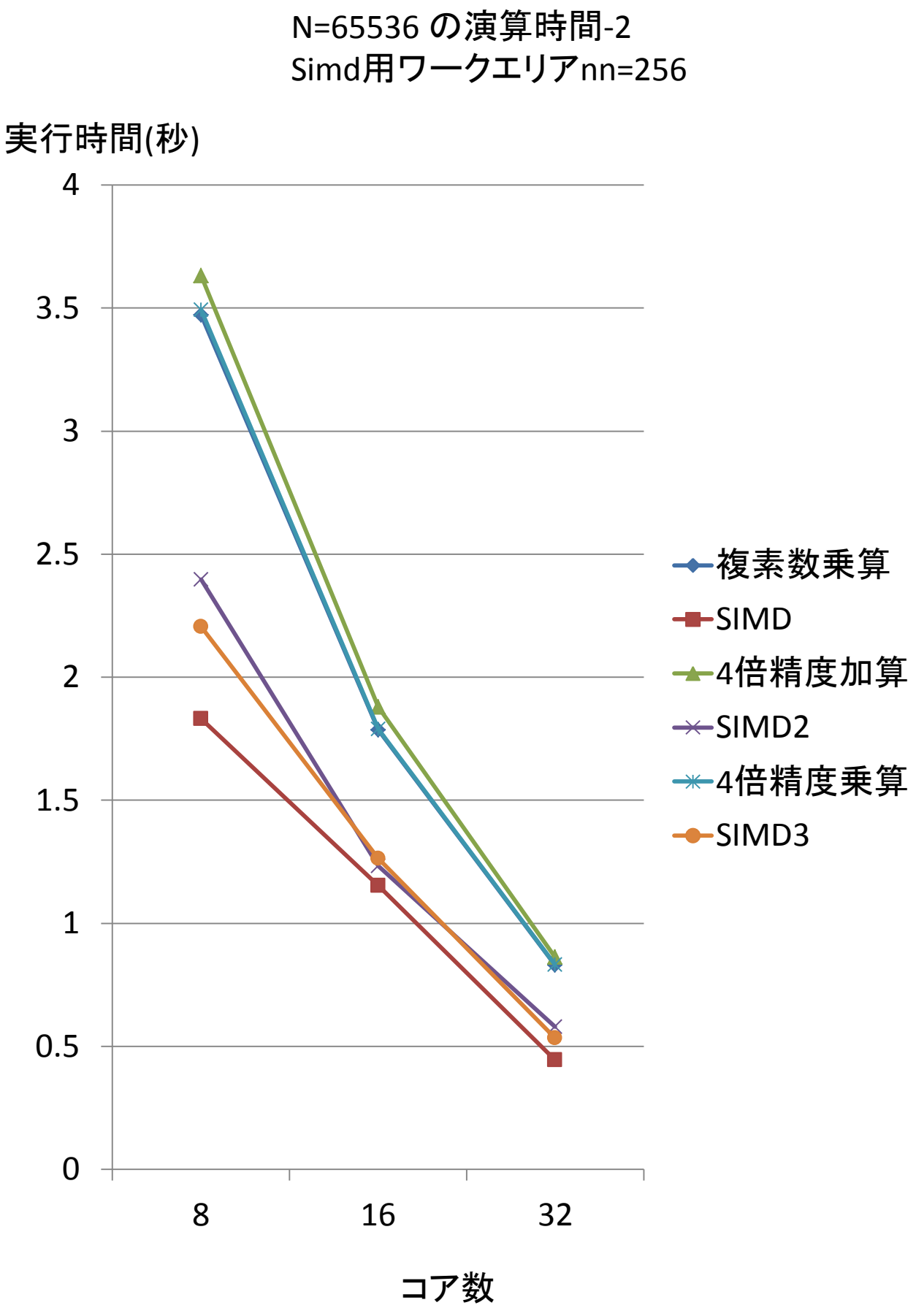
ワークエリアと取り方と最適なサイズ調査

N=16384 の複素数乗加算演算時間



並列化も含めた効果の調査




N=65536 の演算時間-2
Simd用ワークエリアnn=256
実行時間(秒)
4
3.5
3
2.5
2
1.5
1
0.5
0
8
16
32
コア数
複素数乗算
SIMD
4倍精度加算
SIMD2
4倍精度乗算
SIMD3

12. 性能logからOpenMP指示行の作成

SR16000/M1 システムでは,チューニングのために、コンパイル時に性能関連情報が取得できます。これをOpenMPの指示行作成に利用できます。その一例をしめします。

```
      sum1=0.0q0
**
**  Parallel processing starting at loop entry
**  Parallel function: _parallel_func_2_MAIN
**  Parallel loop
**    D: TLOCAL variable
**    ZZ: TLOCAL variable
**    SUM3: TLOCAL variable
**    CNT4: TLOCAL variable
**    YY: TLOCAL variable
**    SUM2: TLOCAL variable
**    CNT2: TLOCAL variable
**    XX: TLOCAL variable
**    SUM1: reduction variable (SUM)
**    I3: TLOCAL variable
**    I2: TLOCAL variable
```

TLOCAL変数はprivateに置き換えます。

```
 do i1=1,n
     xx=x30(i1)*cnt0
     by=1.0q0-xx
     cnt2=by-ay
     sum2=0.0q0
**
     do i2=1,n
     yy=x30(i2)*cnt2
     bz=1.0q0-xx-yy
     cnt4=bz-az
     sum3=0.0q0
**
**  Continued parallel processing
**  Parallel processing finishing at loop exit
n the loop.
**
     do i3=1,n
     zz=x30(i3)*cnt4
     d = -xx*yy*s-tt*zz*(1.0q0-xx-yy-zz)+(xx+yy)*ramda**2+
    1    (1.0q0-xx-yy-zz)*(1.0q0-xx-yy)*fme**2+zz*(1.0q0-xx-yy)*fmf**2
     sum3=sum3+cnt0*cnt2*cnt4*(gw30(i1)/d)*(gw30(i2)/d)*gw30(i3)
     end do
     sum2=sum2+sum3*h
     end do
     sum1=sum1+sum2*h
     end do
```

!$OMP parallel do を最外側DO i1=1,nの前に挿入すれば良い事を表しています。

指示行挿入後のソース

```
!$OMP parallel do
!$OMP& reduction(+:sum1)
!$OMP& private(xx,yy,cnt2,cnt4,i2,i3,sum2,sum3,d,by,bz)
      do i1=1,n
      xx=x30(i1)*cnt0
      by=1.0q0-xx
      cnt2=by-ay
      sum2=0.0q0
      do i2=1,n
      yy=x30(i2)*cnt2
      bz=1.0q0-xx-yy
      cnt4=bz-az
      sum3=0.0q0
      do i3=1,n
      zz=x30(i3)*cnt4
      d = -xx*yy*s-tt*zz*(1.0q0-xx-yy-zz)+(xx+yy)*ramda**2+
     1    (1.0q0-xx-yy-zz)*(1.0q0-xx-yy)*fme**2+zz*(1.0q0-xx-yy)*fmf**2
      sum3=sum3+cnt0*cnt2*cnt4*(gw30(i1)/d)*(gw30(i2)/d)*gw30(i3)
      end do
      sum2=sum2+sum3*h
      end do
      sum1=sum1+sum2*h
      end do
!$OMP end parallel do
```

13. その他

計算機の性能では整数演算,論理演算,マスク演算,リスト演算は見逃がされている事が多く，整数演算による性能低下や,並列化効果の低下をもたらす事もありますので２つのお勧めの例を示しました。

(ア)mod関数はiand関数に変更

```
mod(I,2) => iand(I,1)
mod(I,4)=> iand(I,3)
```

(イ)if 文の削除(並列化実行時の演算量の均等化)

```
do it=1,nt
do iz=1,nz
do iy=1,ny
do ix=1,nx
if(mod(ix+iy+iz+it,2).eq.ieo) then
```

は以下の様に修正するのが良いでしょう。

```
do it=1,nt
do iz=1,nz
do iy=1,ny
 ip=iand(ix+iy+iz+it+ieo+1,1)
 do ix=1+ip,nx,2
```