A64fx アーキテクチャ向けアプリケーションチューニング**-**差分法を例に

牧野淳一郎
神戸大学惑星学専攻/惑星科学研究センター/理研 **R-CCS**
2020/9/3 第 **14** 回アクセラレーション技術発表討論会

概要

- 何が問題か？ — 規則格子陽解法コードの「京」、富岳での効率
- 何ができるか？ — キャッシュブロッキングは有効か？
- 単純な実験の結果
- 単純な実験からの検討、改善されたコード生成
- 結果と考察

ボトムライン

A64fx では陽解法差分法で **10-15%** 以上の効率は
色々がんばっても実現できなかった。

何が問題か？ — 規則格子陽解法コードの「京」、富岳での効率

- 「京」、「富岳」での、規則格子陽解法コードの実行効率はあまり高くない。知られているもので **15%** を超えるものはない。
- **B/F** は **0.36** とか **0.5** あり、理論限界はもっと高い「はず」である。
- **PEZY-SC2** でも、テンポラルブロッキング使って **18%**、使わなくても **15%** くらいでた。**B/F=0.03** しかない。
- メモリバンド幅 **10** 倍あるんだからもうちょっとなんとかならないか？

「京」、「富岳」での規則格子陽解法コードの典型的実装

比較的単純な **DO** ループ、例えば

```
      do k=1,nnz
        do j=1,nny
          do ii=1,nx
            i=ii+2
              r_x(i,j,k) = ((r_p(i-2,j,k)-r_p(i+2,j,k))
     *         -8*(r_p(i-1,j,k)-r_p(i+1,j,k)))*c
          enddo
        enddo
      enddo
```

こんな感じのものを多数並べる。一つのループは、配列の全要素をなめる。

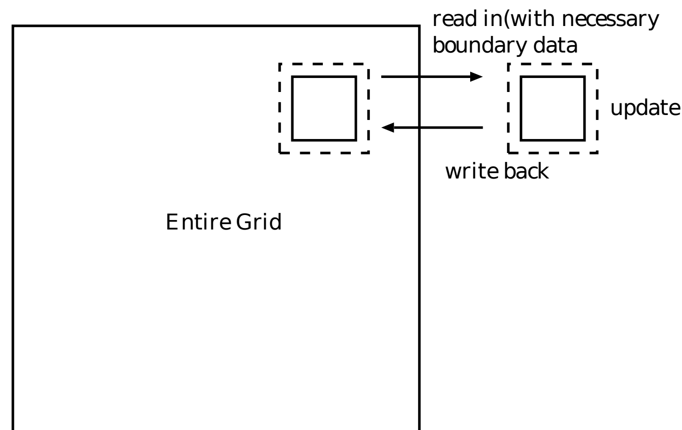
何故こうするか？

- **1980** 年代のベクトルプロセッサではこれが一番よかった。
- **B/F=4** とかあればこれでピーク性能でる。
- **NEC SX-8** までは **B/F=4** あったので全然問題なかった。
- 完全に全部主記憶アクセスしたら、 **B/F=0.5** だと効率 **5%**くらいまで落ちる。
- ある程度 **L1D**、**L2D** でデータ再利用が効くので、 **10-15%** になる。

もうちょっとなんかできないか？
—キャッシュブロッキング

キャッシュブロッキングの基本的考え

- 格子点のブロックをキャッシュに読み込んだら、時間積分に必要な全ての操作をする。これは **3** 次元流体で数千オペレーションある。
- **5** 変数、**1** 度読んで書く、で **80** バイトアクセス、**2000** 演算なら **B/F** は **0.04** でいい。
- **PEZY-SC2** とか **GRAPE-PFN2** 「以外」ならみんな **B/F=0.05** くらいはあるのでこれで **OK**。



A64fx で色々実験してみた

- 実験 1
- 実験 2

実験1

流体のオイラー方程式

```
r_t = -r*u_x - r*v_y - r*w_z - r_x*u - r_y*v - r_z*w
u_t = -p_x*(r)**(-1) - u*u_x - u_y*v - u_z*w
v_t = -p_y*(r)**(-1) - u*v_x - v*v_y - v_z*w
w_t = -p_z*(r)**(-1) - u*w_x - v*w_y - w*w_z
p_t = -gm*p*u_x - gm*p*v_y - gm*p*w_z - p_x*u - p_y*v - p_z*w
```

(人工粘性なし**)** を、空間**4**次精度補間、時間方向**3**次精度**(**エルミート積分**)**の**SL4TH3**スキームで積分**(PEZY-SC2** でやったのと同じ**)**

$$
\begin{aligned}
q_x &= \frac{1}{12h}(q^{(-2)} - 8q^{(-1)} + 8q^{(+1)} - q^{(+2)}) + O(h^4), \\
q_{xx} &= \frac{1}{12h^2}(-q^{(-2)} + 16q^{(-1)} - 30q^{(0)} + 16q^{(+1)} - q^{(+2)}) + O(h^4)
\end{aligned}
$$

コード生成

上の方程式表現から **F77** のコードを生成する。
こんな感じのコードがでる。

```
      r_t =-r_z*w_p(i,j,k)-r_y(i,j,k)*v_p(i,j,k)
     *       -r_x(i,j,k)*u_p(i,j,k)-r_p(i,j,k)*w_z
     *       -r_p(i,j,k)*v_y(i,j,k)-r_p(i,j,k)*u_x(i,j,k)
      u_t =-u_z*w_p(i,j,k)-u_y(i,j,k)*v_p(i,j,k)
     *       -u_p(i,j,k)*u_x(i,j,k)-p_x(i,j,k)*rinv
```

- データが **L2** にはいるように **16x16x16** の格子でやった。
- 富士通コンパイラも **gfortran** もちゃんと **SIMD** 化したいい感じの機械語生成した。

パーサ

再帰下降パーサ(**code)**

使った。**Yacc** とかじゃなくてちょっと古典的**(Wirth** の**Algorithms amd Data Structure** にあるもの**)**

元々**Introduction of numerical calculation using Crystal** 用に作ったもの。

性能

- 私のこのノートで **10Gflops (Skylake without AVX512. Peak clock: 3.2GHz)**
- **FX700** だと **4Gflops (**富岳と同じ **A64fx**、 **512-bit SVE, 2GHz)**

私のノートで理論ピークの **1/3** 程度。**(**名目ピークからは **20%**。掛け算少ないので、、、**)**
FX700 は理論ピークの **6%**。非常に低い。

機械語の例

AVX2

```
movq     -1704(%rbp), %rdx
vmovapd (%rdx,%rax), %ymm11
movq     -1712(%rbp), %rdx
vsubpd  (%rdx,%rax), %ymm11, %ymm11
movq     -1688(%rbp), %rdx
vmovapd (%rdx,%rax), %ymm0
movq     -1696(%rbp), %rdx
vsubpd  (%rdx,%rax), %ymm0, %ymm0
movq     -1736(%rbp), %rdx
vmovapd (%rdx,%rax), %ymm14
movq     -1744(%rbp), %rdx
vsubpd  (%rdx,%rax), %ymm14, %ymm14
vfnmadd132pd    %ymm7, %ymm0, %ymm11
movq     -1720(%rbp), %rdx
vmulpd  %ymm12, %ymm11, %ymm8
vmovapd (%rdx,%rax), %ymm0
movq     -1728(%rbp), %rdx
vmovapd %ymm8, -176(%rbp)
```

アドレス計算あるけどそんなに多くない。**rdx**、**rax** にそれぞれ配列のベースアドレス、添字がはいっていると思われる。

機械語の例

SVE

```
ld1d    {z0.d}, p0/z, [x0, 0, mul vl]
mov     x0, x8
movk    x0, 65173, lsl #16
movk    x0, 6968, lsl #0
add     x0, x5, x0
add     x10, x0, x3
mov     x0, 23724032
movk    x0, 53184, lsl #0
mov     x3, x8
movk    x3, 64810, lsl #16
add     x0, x10, x0
fmla    z4.d, p0/m, z2.d, z1.d
fsub    z0.d, z0.d, z4.d
fmul    z0.d, z0.d, z3.d
st1d    {z0.d}, p0, [x0, 0, mul vl]
```

整数命令ばっかり。浮動小数点計算はどこ？みたいな。これは、アンロールされた都合もある **(SVE load/store は ld1d/st1d)**

命令セットの違い

	AVX	**SVE**
メモリオペランド	あり	なし
アドレスオフセット	＞ **20** ビット	**8** ビット

- **ARM** はロード・ストアアーキテクチャなので、必要な命令数多い。
- 何故かよくわからないが、アドレス計算の命令も増えている。
- とはいえ、**AVX** でも効率低い。

実験2

- 変数1つ、単純化した方程式。

実験2: コード本体

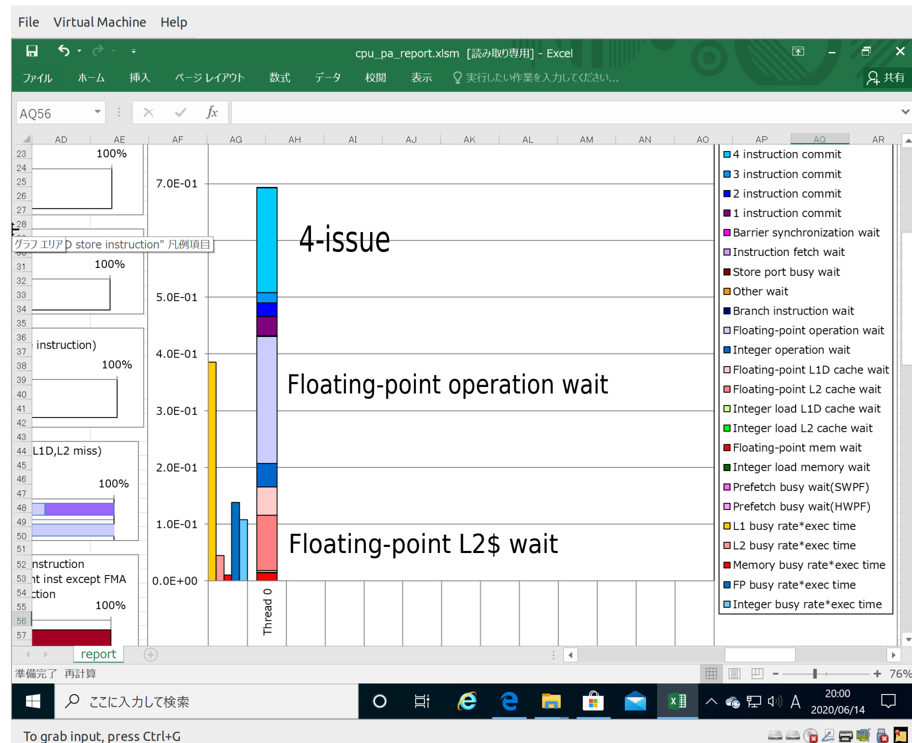
```
      do jj=1,ny
        j = jj+2
        do ii=1,nx
          i = ii+8
          u0xx = (-(u00(i-2,j,k0)+u00(i+2,j,k0)) +16*(u00(i-1,j,k0)+u00(i+1,j,k0)) -30*(u00(i,j,k
          u0yy = (-(u00(i-2,j,k0)+u00(i+2,j,k0)) +16*(u00(i-1,j,k0)+u00(i+1,j,k0))-30*(u00(i,j,k0
          u0xy = (+(u0x(i,j-2,k0)-u0x(i,j+2,k0)) -8*(u0x(i,j-1,k0)-u0x(i,j+1,k0)))*c
          u0z = (+(u00(i,j,km2)-u00(i,j,kp2))     -8*(u00(i,j,km1)-u00(i,j,km1)))*c
          u0xz = (+(u0x(i,j,km2)-u0x(i,j,kp2))    -8*(u0x(i,j,km1)-u0x(i,j,kp1)))*c
          u0yz = (+(u0y(i,j,km2)-u0y(i,j,kp2))    -8*(u0y(i,j,km1)-u0y(i,j,kp1)))*c
          u0zz = (-(u00(i,j,km2)+u00(i,j,kp2))+16*(u00(i,j,km1)+u00(i,j,kp1))-30*(u00(i,j,k0)))*c
          u0(i,j,k) = u00(i,j,k0)+ 0.1*(u0x(i,j,k0)+ u0y(i,j,k0)+ u0z) + 0.01*(u0xx+u0yy+u0zz)
*              + 0.001*(u0xy+u0xz+u0yz)
        enddo
      enddo
```

Note that u_x and u_y are calculated in separate loops to reduce the calculation cost.

実験2: 性能

- 私のノート **PC (Core i7-8565U) : 18Gflops**
- **Xeon Gold 6140: 10 Gflops**
- **FX700: 8 Gflops**
- **Core i7** と **Xeon** の性能は主記憶リミット。
- **FX700** はそうじゃない。何故？

性能が低い本当の理由



- 一番大きいのは、「浮動小数点演算待ち」。**A64fx** の演算レイテンシが大きいので、パイプラインが埋まらない。
- **L2** キャッシュ待ちも大きい。

ではどうするか？

演算レイテンシを可能な限り隠すような機械語コードをコンパイラを通さないで生成することを考える。例えば、

```
                r_x(i,j,k) = ((r_p(i-2,j,k)-r_p(i+2,j,k))
     *           -8*(r_p(i-1,j,k)-r_p(i+1,j,k)))*c
```

は、定数 **(8** と **c)** のためのレジスタ **2** 個、積算のためのレジスタ **1** つ、あともう **2** つで書けるので、**32** 個のアーキテクチャレジスタを使って **8** 重にアンロールできる。

Assembly code for A64fx

なんかこんなのをだす。

```
.arch armv8-a+fp16+sve
.file "testasm.c"
.text
.align 2
.p2align 3,,7
.global svefadd
.type svefadd, %function
svefadd:
.LFB0:
.cfi_startproc
ptrue   p7.d, all
ld1d z0.d, p7/z, [x0]
ld1d z1.d, p7/z, [x1]
fadd z0.d, z1.d, z0.d
st1d z0.d, p7,   [x0]
ret
.cfi_endproc
.LFE0:
.size svefadd, .-svefadd
.ident "GCC: (GNU) 8.2.1 20180905 (Red Hat 8.2.1-3)"
.section .note.GNU-stack,"",@progbits
```

実際に生成したコードの例

```
ld1d    z0.d, p7/z, [x2, -8, mul vl]  fsub    z0.d, z1.d, z0.d
ld1d    z2.d, p7/z, [x2, -7, mul vl]  fsub    z2.d, z3.d, z2.d
ld1d    z4.d, p7/z, [x2, -4, mul vl]  fsub    z4.d, z5.d, z4.d
ld1d    z6.d, p7/z, [x2, -3, mul vl]  fsub    z6.d, z7.d, z6.d
ld1d    z8.d, p7/z, [x2, 0, mul vl]   fsub    z8.d, z9.d, z8.d
ld1d    z10.d, p7/z, [x2, 1, mul vl]  fsub    z10.d, z11.d, z10.d
ld1d    z12.d, p7/z, [x2, 4, mul vl]  fsub    z12.d, z13.d, z12.d
ld1d    z14.d, p7/z, [x2, 5, mul vl]  fsub    z14.d, z15.d, z14.d
ld1d    z1.d, p7/z, [x5, -8, mul vl]  ld1d    z1.d, p7/z, [x3, -8, mul vl]
ld1d    z3.d, p7/z, [x5, -7, mul vl]  ld1d    z3.d, p7/z, [x3, -7, mul vl]
ld1d    z5.d, p7/z, [x5, -4, mul vl]  ld1d    z5.d, p7/z, [x3, -4, mul vl]
ld1d    z7.d, p7/z, [x5, -3, mul vl]  ld1d    z7.d, p7/z, [x3, -3, mul vl]
ld1d    z9.d, p7/z, [x5, 0, mul vl]   ld1d    z9.d, p7/z, [x3, 0, mul vl]
ld1d    z11.d, p7/z, [x5, 1, mul vl]  ld1d    z11.d, p7/z, [x3, 1, mul vl]
ld1d    z13.d, p7/z, [x5, 4, mul vl]  ld1d    z13.d, p7/z, [x3, 4, mul vl]
ld1d    z15.d, p7/z, [x5, 5, mul vl]  ld1d    z15.d, p7/z, [x3, 5, mul vl]
```

実際に生成したコードの例

```
fmla     z0.d, p7/m, z1.d, z31.d       fmls    z0.d, p7/m, z1.d, z31.d
fmla     z2.d, p7/m, z3.d, z31.d       fmls    z2.d, p7/m, z3.d, z31.d
fmla     z4.d, p7/m, z5.d, z31.d       fmls    z4.d, p7/m, z5.d, z31.d
fmla     z6.d, p7/m, z7.d, z31.d       fmls    z6.d, p7/m, z7.d, z31.d
fmla     z8.d, p7/m, z9.d, z31.d       fmls    z8.d, p7/m, z9.d, z31.d
fmla     z10.d, p7/m, z11.d, z31.d     fmls    z10.d, p7/m, z11.d, z31.d
fmla     z12.d, p7/m, z13.d, z31.d     fmls    z12.d, p7/m, z13.d, z31.d
fmla     z14.d, p7/m, z15.d, z31.d     fmls    z14.d, p7/m, z15.d, z31.d
ld1d    z1.d, p7/z, [x4, -8, mul vl]   fmul    z0.d, z0.d, z30.d
ld1d    z3.d, p7/z, [x4, -7, mul vl]   fmul    z2.d, z2.d, z30.d
ld1d    z5.d, p7/z, [x4, -4, mul vl]   fmul    z4.d, z4.d, z30.d
ld1d    z7.d, p7/z, [x4, -3, mul vl]   fmul    z6.d, z6.d, z30.d
ld1d    z9.d, p7/z, [x4, 0, mul vl]    fmul    z8.d, z8.d, z30.d
ld1d    z11.d, p7/z, [x4, 1, mul vl]   fmul    z10.d, z10.d, z30.d
ld1d    z13.d, p7/z, [x4, 4, mul vl]   fmul    z12.d, z12.d, z30.d
ld1d    z15.d, p7/z, [x4, 5, mul vl]   fmul    z14.d, z14.d, z30.d
```

性能

- **16x16x128** 格子で **9.2Gflops**。実行効率 **14%**。
- すごくがんばって全部アセンブラにしてプリフェッチもいれてあらゆることしても **15%** 超えられなかった。
- 性能低下の主要な要因**: L2** 待ちにはなった。**8** 重アンロールが効いて演算待ちは **L2** 待ちの半分以下になった。また、アドレス計算はベクトルアクセス用の型式を使うことで浮動小数点命令より少なくできた。
- **L2** アクセスは基本的には **512** バイト連続だが、かなりの部分が **128** バイト単位になる。これは **x, y** 方向の格子サイズが小さく、袖もあるため。
- **x** 方向大きくすると **L1** にはいらない。現在のサイズでも、中間変数含めると最内側ループでも **L1** にはいらないので、 **L1** ミスレートは結構高い **(L1** ヒットは **90%)**

性能を決める要因

- 可能な限りアンロールしないと、演算レイテンシを隠蔽できない。**(9** サイクルある**)**
- アンロールすると、 **L1** キャッシュからあふれる
- **L2** キャッシュの、連続アクセスではない時のバンド幅が非常に低い。連続コピーだと **3.6GW/s**、 リードが **512** バイト単位のランダムアクセスだと**3GW/s** だが、**128** バイト単位だと **2GW/s** まで落ちる **(**プリフェッチちゃんといれて**)**。
- この **L2** の速度で、性能はほぼ説明できる。

もうちょっとなんとかできる可能性はあるか？

- **L1** のヒットレートをあげる。このためには **unroll** 回数を減らす必要があり、浮動小数点演算待ちが増えるのでなかなか難しい。
- **L2** のアクセス単位を大きくする。問題は **x** 方向の格子点数なので、簡単ではないが、**1** つの方法は、 **x** と **y** の間に物理変数をもってくる **AoSoA** 構造にすることである。これで空間微分計算についてはアクセス単位を **5** 倍にできる。しかし、中間変数に対してはアクセス単位が小さくなる
- **x** 方向の格子点数を増やす。**L2** アクセス単位は大きくなるが **L1** ヒットレートは下がるので、あまりよくない。

ハードウェアとしては？

- 演算レイテンシが長い
- **L1** が小さい
- **L2** レイテンシが大きく、バンド幅も小さい

というあたりが厳しい理由。

SC2 では、相対的に巨大なローカルメモリの **B/F** が高いために容易に性能がでた。

まとめ

- **A64fx** で、規則格子陽解法コードのキャッシュブロッキングによる性能向上を試みた。
- 非常に頑張ったが、効率は **14%**程度ですごく高いとはいえない。
- 演算レイテンシ、**L2** レイテンシが大きく、**L1** が小さく、**L2** バンド幅が小さい、ということで性能が制限される。
- 今回採用した、主記憶アクセスを極限まで減らせるスキームより、通常の **flux splitting** をして方向毎に主記憶アクセスする代わりに中間変数を減らすスキームのほうが、**L1** ヒットレートをあげられる可能性があり、その検討が必要である。