Formura – –

Jun Makino

Jun 17, 2020 Internal seminar

Talk overview

- What is the problem? — The efficiency of explicit, regular-grid calculations on K and Fugaku
- What can be done — cache blocking or more?
- “Simple” experiment and result
- The real reason for the low efficiency
- What can/should be done next?

What is the problem? — The efficiency of explicit, regular-grid calculations on K and Fugaku

- Efficiency of all known regular-grid codes on both K or Fugaku is not very high. 15% or less.
- This is MUCH lower than the theoretical limit determined by the memory bandwidth.
- A very well-designed code on PEZY-SC2 can reach to 18%. (achieved by temporal blocking)
- Relative memory bandwidth (B/F) of SC2: 0.03, K:0.5, Fugaku:0.36.

Typical implementation of regular-grid codes on K/Fugaku

Implement the numerical scheme as a large number of DO (or for) loops, each represents relatively simple operation. For example, something like

```
 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
```

which loops over all elements of the grid assigned to one MPI process.

Why in this form?

- The best way to get good performance on vector supercomputers of 1980s and 1990s.
- With sufficient memory bandwidth (B/F=4) , the loops of this style can achieve near-peak performance.
- Worked very well on vector machines (up to NEC SX-8)
- Not very efficient on machines with cache hierarchy — efficiency will be 5-10% if B/F ~ 0.5.
- Actually, the data in L1 cache is reused a few times, resulting in somewhat better efficiency.

Efficiency estimate

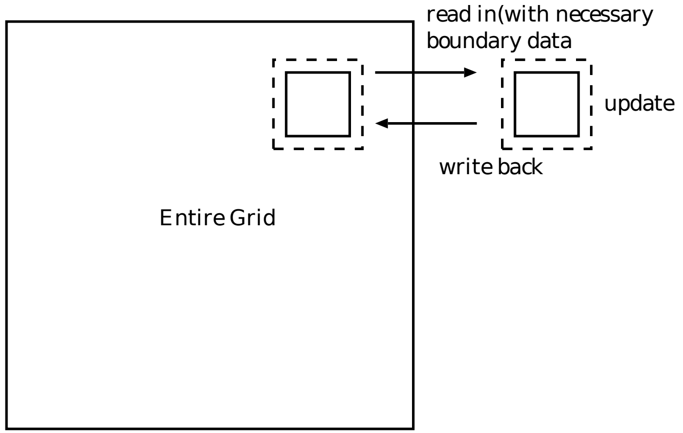
```
              r_x(i,j,k) = ((r_p(i-2,j,k)-r_p(i+2,j,k))
     *         -8*(r_p(i-1,j,k)-r_p(i+1,j,k)))*c
```

- **Looked as a vector operation, this statement requires one vector load, one vector store and five operations. (or one FMA and three non-FMA operations).**
- **Necessary B/F value to reach the peak performance = 16/8 = 2. Efficiency = 16% if B/F=0.5**
- **Finite difference in y-direction might hit L1 cache**
- **Finite difference in z-direction would not hit L1 cache**
- **Thus, overall efficiency $\sim$ 10%**

What can be done — cache blocking or more?

The basic idea of cache blocking:

- **If we read-in some block of grid point into the cache (usually the last-level cache) and apply all operations for one timestep, we can apply something like 2000 operations per grid point.**
- **Thus, 80 bytes (5 variable, read and write) for 2000 operations: required B/F = 0.04.**
- **ANY machine other than PEZY-SC2 and MN-Core has much more than enough memory bandwidth.**



“Simple” experiment and result

- First experiment
- Second experiment

First experiment

Euler equations of fluid

```
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
```

(artificial viscosity not included) integrated with SL4TH3 scheme (space 4th order collocation , time 3rd order Hermite).

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

Code generation

FORTRAN77 program is generated from Formura-like input (used in the yet-unfinished SP4TH3 paper). The equations in the previous slide is from that input file.

Example of the generated code:

```
            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
```

- Test run for 16x16x16 box, so that all data can fit to L2 cache
- Both Fujitsu FORTRAN compiler and gfortran generate reasonable machine code

The parser

A simple recursive descent parser
code
written for
Introduction of numerical calculation using Crystal

Performance measured

- Around 10Gflops on my note (Skylake without AVX512. Peak clock: 3.2GHz)
- Around 4Gflops on FX700 (The same processor as Fugaku. 512-bit SVE, 2GHz)

Performance of my note is still low. The theoretical peak (with FMUL ratio taken into account) = 28Gflops.

FX700 performance surprizingly low

Machine code example

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)
```

Looks reasonable (not ideal)

Machine code example

SVE

```
ldr     x2, [sp, 856]
ldr     q0, [x28, x1]
ldr     q2, [x2, x1]
ldr     q14, [x30, x1]
ldr     x2, [sp, 864]
str     q2, [sp, 688]
fsub    v14.2d, v14.2d, v0.2d
ldr     q23, [x2, x1]
ldr     q0, [x6, x1]
ldr     x2, [sp, 1536]
str     q0, [sp, 208]
```

```
ldr     q0, [x2, x1]
ldr     x2, [sp, 1664]
ldr     q1, [x26, x1]
str     q0, [sp, 240]
ldr     q22, [x2, x1]
ldr     x2, [sp, 888]
ldr     q4, [x27, x1]
ldr     q11, [x2, x1]
ldr     x2, [sp, 896]
fsub    v4.2d, v4.2d, v1.2d
```

large number of integer load/store???

partly because instruction rescheduling

Instruction set difference

	AVX	SVE
Memory operand	yes	no
Address offset	$>$ 20 bits	8 bits

- With AVX, the use of memory operand for arithmetic operation and large integer offset for memory address reduces the need to change values of address registers.
- With ARM SVE, address registers should be updated for practically all memory accesses...
- However, even with AVX2, the efficiency is still low. I have tried a simplified test.

Second test

- One variable (actually can change the number of variable to test cache performance)
- simplified equation

Second test: the code body

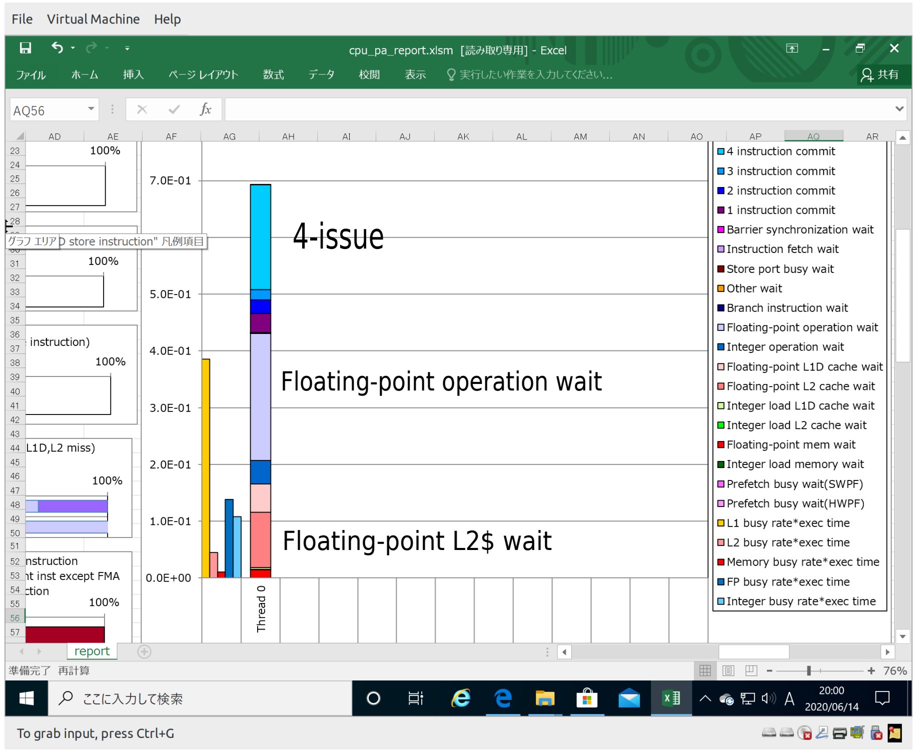
```
   do jj=1,ny
     j = jj+2
     do ii=1,nx
       i = ii+8
       u0xx = (-(u00(i-2,j,k0)+u00(i+2,j,k0)) +16*(u00(i-1,j,k0)+u00(i+1,j,k0)) -30*(u00(i,j
       u0yy = (-(u00(i-2,j,k0)+u00(i+2,j,k0)) +16*(u00(i-1,j,k0)+u00(i+1,j,k0))-30*(u00(i,j,
       u0xy = (+(u0x(i,j-2,k0)-u0x(i,j+2,k0)) -8*(u0x(i,j-1,k0)-u0x(i,j+1,k0)))*c
       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)))
       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.

Second test: the performance

- My note PC (Core i7-8565U) : 18Gflops
- Our Xeon at KU (Xeon Gold 6140): 10 Gflops
- FX700: 8 Gflops
- Core i7 and Xeon numbers are limited by the memory bandwidth (number of floating-point operations per grid point is small)
- FX700 number is not.

The real reason for the low efficiency



- **Dominant source of wait is “Floating-point operation wait”. The large latency of A64fx arithmetic unit prevents its efficient use.**
- **L2$ latency also visible...**

What can/should be done next?

We might be able to control the behavior of the arithmetic unit by generating the assembly code directly from the equation, bypassing the compiler. For example, the following statement

```
                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
```

can be written with two registers for constants, one accumulator, one additional register. So eight-way unrolling is possible. 16-way (effectively 15) might be better.

This unfortunately means we need to generate assembly code directly...

Assembly code for A64fx — My first try

```
.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)"
```

Assembly code for A64fx — My first try

This code actually worked, both with Fujitsu compiler and gcc.
Adding two sve variables give the expected result.

```
x 0.000e+00 1.000e+00 2.000e+00 3.000e+00 4.000e+00 5.000e+00 6.000e+00 7.000e+00
y 1.000e-01 2.000e-01 3.000e-01 4.000e-01 5.000e-01 6.000e-01 7.000e-01 8.000e-01
x after call 1.000e-01 1.200e+00 2.300e+00 3.400e+00 4.500e+00 5.600e+00 6.700e+00 7.800e+00
```

Summary

- **Performance of explicit regular-grid code on A64fx (and K) is usually limited by its memory bandwidth, because they are written in the way the most efficient on vector machines in 1980-90s.**
- **We can generate codes which try to make use of L1/L2 caches, and such code works fine on x86.**
- **The performance of such code is still low on A64fx, because of large latency of arithmetic pipeline and L2 cache**
- **By generating assembly code directly from the original equation and finite difference formulae, we** *might be* **able to improve the performance.**