Experience with memcpy, hand-written SVE and compiler optimizations on A64FX

WENBIN LU

iACS INSTITUTE FOR ADVANCED COMPUTATIONAL SCIENCE

Exascallab
Exascale Programming Models Laboratory
https://you.stonybrook.edu/exascallab/

The STREAM Benchmark

- A synthetic benchmark that measures sustainable memory bandwidth for four simple vector kernels.

```
-----------------------------------------------------------
name        kernel                  bytes/iter      FLOPS/iter
-----------------------------------------------------------
COPY:       a(i) = b(i)                 16              0
SCALE:      a(i) = q*b(i)               16              1
SUM:        a(i) = b(i) + c(i)          24              1
TRIAD:      a(i) = b(i) + q*c(i)        24              2
-----------------------------------------------------------
```

Strange Case of the Copy Loop

- Compile STREAM with `gcc -mcpu=a64fx -O3` and run on Ookami, here are the results (GCC 11.2).

```
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:           16316.8     0.033060     0.032903     0.042272
Scale:          38238.7     0.014087     0.014040     0.014924
Add:            43389.5     0.018689     0.018560     0.023295
Triad:          43396.2     0.018668     0.018557     0.019446
```

A Footgun

- Let's compare the assembly of the COPY loop and the SCALE loop (`gcc -S -fverbose-asm`).

```
// stream.c:315:          c[j] = a[j];
    mov x1, x26 //, tmp546
    mov x2, 46080   //,
    mov x0, x21 //, tmp549
    movk    x2, 0x4c4, lsl 16   //,,
    bl  memcpy      //
```

```
// stream.c:325:          b[j] = scalar*c[j];
    ld1d    z0.d, p0/z, [x21, x0, lsl 3]
// stream.c:325:          b[j] = scalar*c[j];
    fmul    z0.d, z0.d, z1.d    // vect__11
// stream.c:325:          b[j] = scalar*c[j];
    st1d    z0.d, p0, [x20, x0, lsl 3]  //
// stream.c:324:      for (j=0; j<STREAM_ARRA
    incd    x0      // ivtmp_354
    whilelo p0.d, w0, w19   // loop_mask_34
    b.any   .L51    //,
```

Disable GCC Built-in C Functions

- Compile & run with `gcc -mcpu=a64fx -O3 -fno-builtin`.

```
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:           38091.9     0.014152     0.014094     0.014918
Scale:          38243.9     0.014097     0.014038     0.014866
Add:            43410.7     0.018624     0.018551     0.019372
Triad:          43429.1     0.018623     0.018543     0.019369
```

Vendor Compilers

Function	Best Rate MB/s	Avg time	Min time	Max time
LLVM/Clang 15-git (-fno-builtin)				
Copy:	39239.5	0.013750	0.013682	0.014482
Scale:	39310.8	0.013742	0.013657	0.017214
Add:	42747.0	0.018955	0.018839	0.023647
Triad:	42735.1	0.018939	0.018844	0.022270
Armclang 21.1				
Copy:	37946.8	0.014219	0.014148	0.015234
Scale:	38025.0	0.014227	0.014119	0.017280
Add:	43642.9	0.018568	0.018452	0.022444
Triad:	44472.7	0.018240	0.018108	0.023159
Cray 21.03				
Copy:	40217.9	0.013409	0.013349	0.014305
Scale:	40460.7	0.013327	0.013269	0.014220
Add:	44919.2	0.018002	0.017928	0.018874
Triad:	44739.5	0.018085	0.018000	0.018951
Fujitsu 4.5				
Copy:	47862.8	0.011290	0.011217	0.016070
Scale:	47730.9	0.011321	0.011248	0.015752
Add:	54196.7	0.014911	0.014859	0.016392
Triad:	54276.8	0.014884	0.014837	0.016877

What if My Code Uses memcpy/std::copy?

- Most compilers wouldn’t replace function calls
  - We’re stuck with Glibc
  - Armclang is the exception, at least for memcpy

- Or write your own memcpy with SVE!
  - Lots of videos/articles/slides online
  - Sample code provided at the end of this presentation

Impact on Intra-Node Communication

- Eg. shared-memory transports of MPI implementations
  - Glibc's memcpy is slower than inter-node RDMA writes!

- Communication libraries are still catching up
  - UCX proposal to use optimized memcpy for different ISAs
  - MPI: Using Arm Scalable Vector Extension to Optimize OpenMPI
  - Compile your MPI implementation with armclang?

Take-aways

- OSS compilers could do better in SVE-awareness
  - Try `-fno-builtin` and friends if you must use vanilla GCC/Clang
- Try the vendor compilers & benchmark if possible!
  - Fujitsu/Cray could generate super-performant code
  - Armclang is smarter for some cases
- Hand-written SVE routines could help in some cases

Sample Code: SVE memcpy

```
#ifdef __ARM_FEATURE_SVE
#include <arm_sve.h>
void *memcpy_sve(void *dest, const void *src, size_t len)
{
    uint8_t *dest_u8       = (uint8_t*) dest;
    const uint8_t *src_u8 = (uint8_t*) src;
    uint64_t i             = 0;
    svbool_t pg            = svwhilelt_b8_u64(i, (uint64_t)len);

    do {
        const svuint8_t byte_vec = svld1_u8(pg, &src_u8[i]);
        svst1_u8(pg, &dest_u8[i], byte_vec);
        i += svcntb();
        pg = svwhilelt_b8_u64(i, (uint64_t)len);
    } while (svptest_first(svptrue_b8(), pg));

    return dest;
}
#endif  /* __ARM_FEATURE_SVE */
```