Modeling and tuning of SpMV and a lattice QCD kernel on the A64FX

Christie Alappat, Jan Laukemann, Thomas Gruber, Georg Hager, Gerhard Wellein, Nils Meyer, Tilo Wettig

Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany
Erlangen National High Performance Computing Center (NHR@FAU)
University of Regensburg, Germany
Single-core analysis

ECM model

Registers

L1

L2

MEM
Single-core ECM model

Execution-Cache-Memory (ECM) model helps us to understand and analyze the single-core performance.

In-core prediction

Application knowledge

STREAM TRIAD
\[ a[i] = b[i] + s \cdot c[i] \]

```
.L18:
    ld1d z4.d, p5/z, [x21, x9, lsl 3]
    ld1d z5.d, p5/z, [x20, x9, lsl 3]
    fmad z5.d, p5/m, z2.d, z4.d
    st1d z5.d, p5, [x19, x9, lsl 3]
    add x8, x9, 8
    whilelo p5.d, w8, w7
    b.any .L18
```

2cy / VL

Machine knowledge

<table>
<thead>
<tr>
<th>Reservation Stations</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSE0 20 entries</td>
</tr>
<tr>
<td>RSE1 20 entries</td>
</tr>
<tr>
<td>RSA0 10 entries</td>
</tr>
<tr>
<td>RSA1 10 entries</td>
</tr>
<tr>
<td>RSBR 19 entries</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Execution Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int ALU</td>
</tr>
<tr>
<td>FP arith.</td>
</tr>
<tr>
<td>FMA</td>
</tr>
<tr>
<td>FP DIV</td>
</tr>
<tr>
<td>VEC, addr, calc</td>
</tr>
<tr>
<td>FP ST</td>
</tr>
</tbody>
</table>

2cy

1 cy

FMA

LD

LD

ST

2cy

A64FX Symposium | Georg Hager

2021-08-12
In-core prediction

Application knowledge

STREAM TRIAD

\[ a[i] = b[i] + s \times c[i] \]

.L18:

1d1d z4.d, p5/z, [x21, x9, lsl 3]
1d1d z5.d, p5/z, [x20, x9, lsl 3]
fmad z5.d, p5/m, z2.d, z4.d
st1d z5.d, p5, [x19, x9, lsl 3]
add x8, x9, 8
whilelo p5.d, w8, w7
b.any .L18

Machine knowledge

Reservation Stations
- RSE0: 20 entries
- RSE1: 20 entries
- RSA0: 10 entries
- RSA1: 10 entries
- RSBR: 19 entries

Execution Units

Static analysis and prediction of in-core contribution

https://github.com/RRZE-HPC/OSACA
# In-core prediction

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Reciprocal Throughput [cy]</th>
<th>Latency [cy]</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld1d</td>
<td>0.5</td>
<td>11</td>
</tr>
<tr>
<td>st1d</td>
<td>1.0</td>
<td>–</td>
</tr>
<tr>
<td>fadd</td>
<td>0.5</td>
<td>9</td>
</tr>
<tr>
<td>fmad</td>
<td>0.5</td>
<td>9</td>
</tr>
<tr>
<td>faddv</td>
<td>11.5</td>
<td>49</td>
</tr>
</tbody>
</table>

## Static analysis and prediction of in-core contribution

https://github.com/RRZE-HPC/OSACA
Data transfer for STREAM triad

Machine knowledge (FX700)

- Registers
  - 128 B/cy
  - 64 B/cy

- L1
  - 64 B/cy
  - 32 B/cy

- L2
  - 117 B/cy
  - 64 B/cy

- MEM

Application knowledge

STREAM triad

\[ a[i] = b[i] + s*c[i] \]

ECM prediction?

STREAM triad on A64FX

- Registers
  - LD
  - RD
  - WR

- L1
  - RD
  - WR

- L2
  - RD
  - WR

- MEM
Overlap hypotheses for A64FX

Large pages required!
Model validation (FX1000, large pages)

ECM validation for in-memory data sets (single-core)

Lower is better

Runtime [cy/VL]

<table>
<thead>
<tr>
<th>Test Case</th>
<th>ECM Prediction</th>
<th>Measured</th>
</tr>
</thead>
<tbody>
<tr>
<td>COPY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DAXPY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRIAD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCHOENAUER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stencil 2DSPT - LC+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stencil 2DSPT - LC-</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Measured Runtime (cy/VL)
Multicore (in-memory data set)

Sufficient unrolling is crucial (but sometimes it's not enough)
SpMV
Sparse Matrix-Vector Multiplication
Sparse Matrix-Vector Multiplication (SpMV) : \( b = Ax \)

\[
\begin{align*}
\mathbf{b}[: &]= \mathbf{b}[:]+ \mathbf{A}[:, :] \ast \mathbf{x}[:] \\
= & + N_r \\
\end{align*}
\]

General case: some indirect addressing required!

In Compressed Row Storage (CRS) format

\[
\text{for } i = 0: \text{nrows-1} \quad \text{//Long outer loop} \\
\quad \text{for } j = \text{row_ptr}[i]: \text{row_ptr}[i+1]-1 \quad \text{// Short inner loop} \\
\quad \quad \mathbf{b}[i] = \mathbf{b}[i] + \mathbf{A}[j] \ast \mathbf{x}[\text{col_idx}[j]]
\]

Minimum code balance:

\[
B_{cmin} = 6 \frac{\text{byte}}{\text{flop}}
\]
- dRECT: 4000-column tall & skinny dense matrix ($N_{n z r} = 4000$)
- HPCG: matrix from HPCG benchmark ($N_{n z r} = 27$), $128^3$ rows
Assembly of the short inner-loop

In Compressed Row Storage (CRS) format

```
for i = 0:nrows-1  // Long outer loop
    for j = row_ptr[i]:row_ptr[i+1]-1  // Short inner loop
```

ECM model predicts maximum bandwidth of 100 GB/s → No saturation

Assembly of the short inner-loop

```
.L6:
ld1sw  z0.d, p0/z, [x17, x20, lsl 2]
ld1d   z2.d, p0/z, [x18, x20, lsl 3]
ld1d   z3.d, p0/z, [x30, z0.d, lsl 3]
add    x20, x20, 8
fmla   z1.d, p0/m, z3.d, z2.d
whilelo p0.d, x20, x14
b.any  .L6
faddv  d4, p1, z1.d
```

FMA: Update z1.d
Latency: 9 cycles
Loop length : 27 HPCG matrix
Horizontal add of 512-bit register
latency = 49 cycles
The problem with SpMV on A64FX

We need both:
- SIMD vectorization
- Modulo Variable Expansion (MVE)

With CRS, both must be implemented in the inner loop. The partial sums accumulation adds to the overhead.

Can we get rid of the partial sums accumulation \textit{and} separate SIMD from MVE?
CRS $\rightarrow$ SELL-C-$\sigma$

Change data storage format
SELL-C-\(\sigma\)

**Idea**

- Sort rows according to length within sorting scope \(\sigma\)
- Store nonzeros column-major in zero-padded blocks of height \(C\)

\[\beta = \frac{N_{nz}}{\sum_{i=0}^{N_c} C \cdot l_i}\]

\(l_i\): width of chunk \(i\)

“Chunk occupancy”:

- \(N_{nz}\): number of nonzeros
- \(N_c\): number of chunks
- \(C\): height of blocks
- \(l_i\): width of chunk \(i\)

Zero padding
SELL-C-\(\sigma\) kernel

Example \(C = 4\) without further unrolling \(\rightarrow\) longer inner loop, but still an LCD

```c
for (i = 0; i < N/4; ++i)
{
    for (j = 0; j < cl[i]; ++j)
    {
        y[i*4+0] += val[cs[i]+j*4+0] * x[col[cs[i]+j*4+0]];
        y[i*4+1] += val[cs[i]+j*4+1] * x[col[cs[i]+j*4+1]];
    }
}
```

\(C = 4\)
How to choose the parameters?

- $C$
  - $n \times$ SIMD width to allow good utilization of SIMD units
  - $n > 1$ useful for hiding ADD pipeline latency

- $\sigma$
  - As small as possible, as large as necessary
  - Large $\sigma$ reduces zero padding (brings $\beta$ closer to 1)
  - Sorting alters RHS access pattern $\rightarrow \alpha$ depends on $\sigma$

- **SELL-C-\(\sigma\)** separates SIMD from sum reduction
- \(C>8\) allows for reduction of fmla latency impact

![Graph](image-url)

**Performance [Gflop/s]**

![HPCG-128\(^3\)](image-url)

- GCC SELL-8-1
- GCC SELL-16-1
- FCC SELL-16-1
- FCC CRS
SpMV performance with SELL-C-σ (full chip)

foreach m in $matrices:
    apply RCM reordering if helpful
    try row-based vs. nonzero-based load balancing
    scan σ from 1 … 4096
Domain Wall (DW) kernel

from Quantum Chromodynamics (QCD)
Context

- Lattice QCD simulates the strong interaction
- Iterative multigrid techniques on regular (4D or 5D) lattices
- Core component: Apply Dirac operator $D$ to quark-field vector $\Psi$
- Domain Wall (DW) formulation: quark field lives on 4D boundary of a 5D space-time volume $V_4 \times L_S$

\[(D\psi)(n, s)_{aa} = \sum_{\mu=1} U_\mu(n)_{ab}(1 + \gamma_\mu)_{\alpha\beta} \psi(n + \mu, s)_{\beta b} + U^\dagger_\mu(n - \mu)_{ab}(1 - \gamma_\mu)_{\alpha\beta} \psi(n - \hat{\mu}, s)_{\beta b}\]
#define x_p 1 // x-plus direction
#define x_m 2 // x-minus direction
#define y_p 3 // y-plus direction

...#pragma omp parallel for schedule(static)
for \( \{ t, z, y, x \} = 1: \{ L_t-2, L_z-2, L_y-2, L_x-2 \} \) // collapsed loop over 4d space-time
{
  for (int s=0; s<L_s; ++s) // loop over 5th dimension
  {
    O[t][z][y][x][s] = R(x_p) \cdot U[x_p][t][z][y][x] \cdot P(x_p) +
                      R(x_m) \cdot U[x_m][t][z][y][x] \cdot P(x_m) +
                      R(y_p) \cdot U[y_p][t][z][y][x] \cdot P(y_p) +
                      R(y_m) \cdot U[y_m][t][z][y][x] \cdot P(y_m) +
                      R(z_p) \cdot U[z_p][t][z][y][x] \cdot P(z_p) +
                      R(z_m) \cdot U[z_m][t][z][y][x] \cdot P(z_m) +
                      R(t_p) \cdot U[t_p][t][z][y][x] \cdot P(t_p) +
                      R(t_m) \cdot U[t_m][t][z][y][x] \cdot P(t_m) +
  }
Complex numbers data layout choice

RIRI (standard)  

R R R R R R R R I I I I I I I I I I R R ...

RRII  

R I R I R I R I R I ...
Observed performance

- Starting point: RIRI layout, ACLE intrinsics, GCC/FCC
- 1320 flops/LUP (theoretical)
- Measured code balance: 1500 byte/LUP

**A64FX (FX1000):** $B_m = 0.25 \frac{\text{byte}}{\text{flop}} \Rightarrow$ expect memory bound

\[
B_c \approx 1.14 \frac{\text{byte}}{\text{flop}}
\]
Layer Conditions (LC) analysis

- LC determine traffic to/from different caches \((i)\) of size \(s_i\)
- \(w = 2\): write-allocate factor (would be 1 if WA evasion applies)
- RIRI (RRII) layout has \(d = 1\) (\(d = 2\))

<table>
<thead>
<tr>
<th>Name</th>
<th>(V_j) in byte/LUP</th>
<th>(s_i) &gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>no reuse</td>
<td>((8 \cdot 9 + 8 \cdot 12 + w \cdot 12) \cdot 16)</td>
<td>0</td>
</tr>
<tr>
<td>(LC_s)</td>
<td>((8 \cdot 9/L_s + 8 \cdot 12 + w \cdot 12) \cdot 16)</td>
<td>2880</td>
</tr>
<tr>
<td>(LC_x)</td>
<td>((8 \cdot 9/L_s + 7 \cdot 12 + w \cdot 12) \cdot 16)</td>
<td>(2 \cdot L_s(8 \cdot 9/L_s + 8 \cdot 12 + 12) \cdot 16)</td>
</tr>
<tr>
<td>(LC_y)</td>
<td>((8 \cdot 9/L_s + 5 \cdot 12 + w \cdot 12) \cdot 16)</td>
<td>(2 \cdot L_s \cdot L_s(8 \cdot 9/L_s + 7 \cdot 12 + 12) \cdot 16)</td>
</tr>
<tr>
<td>(LC_z)</td>
<td>((8 \cdot 9/L_s + 3 \cdot 12 + w \cdot 12) \cdot 16)</td>
<td>(2 \cdot L_s \cdot L_x \cdot L_y(8 \cdot 9/L_s + 5 \cdot 12 + 12) \cdot 16)</td>
</tr>
<tr>
<td>(LC_t)</td>
<td>((8 \cdot 9/L_s + 1 \cdot 12 + w \cdot 12) \cdot 16)</td>
<td>(2 \cdot L_s \cdot L_x \cdot L_y \cdot L_z(8 \cdot 9/L_s + 3 \cdot 12 + 12) \cdot 16)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>512-bit vectorized code</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11520</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(d \cdot 8 \cdot L_s(8 \cdot 9/L_s + 8 \cdot 12 + 12) \cdot 16)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(d \cdot 8 \cdot L_s \cdot L_x(8 \cdot 9/L_s + 7 \cdot 12 + 12) \cdot 16)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(8 \cdot L_s \cdot L_x \cdot L_y(8 \cdot 9/L_s + 5 \cdot 12 + 12) \cdot 16)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(4 \cdot L_s \cdot L_x \cdot L_y \cdot L_z(8 \cdot 9/L_s + 3 \cdot 12 + 12) \cdot 16)</td>
</tr>
</tbody>
</table>
SIMD-friendly data layout

- One partition per SIMD lane
- Partition size:
  
  \[ L_x \times L_y \times \frac{L_z}{2} \times \frac{L_t}{2} \times L_s \]
- → SIMD makes LCs more stringent (need more cache to fulfill)
LC effect

Measurement of code balance with $L_s = 8$

![Graph showing memory traffic and lattice size]
Summary of optimizations for DW

- **Software prefetching** decreases L2 data volume
- `-O1` makes compiler obey the ordering hints in the computational kernel (more efficient OoO execution)
- **RRII data layout**
  - Prevents use of complex arithmetic instructions `fcmla/fcadd`
  - Removes imbalance between FLA and FLB ports in the core
  - Some register spills occur, but still better than RIRI
  - Measurement falls short of ECM prediction by 2.3x (GCC) or 3.1x (FCC)
DW kernel optimizations and ECM model

The diagram shows the L2 traffic, memory traffic, and performance for GCC and FCC under different implementations:

- **L2 traffic [byte/LUP]**
  - GCC
  - FCC
  - RIRI baseline
  - RIRI prefetch
  - RIRI prefetch+01
  - RIIII prefetch+01
  - LC prediction

- **Memory traffic [byte/LUP]**
  - GCC
  - FCC
  - RIRI baseline
  - RIRI prefetch
  - RIRI prefetch+01
  - RIIII prefetch+01
  - LC prediction

- **Performance [Gflop/s]**
  - GCC
  - FCC
  - RIRI baseline
  - RIRI prefetch
  - RIRI prefetch+01
  - RIIII prefetch+01
  - LC prediction

<table>
<thead>
<tr>
<th>Implementation</th>
<th>GCC</th>
<th>FCC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T_{c,OL}$</td>
<td>$T_{L1,LD}$</td>
</tr>
<tr>
<td>RIRI-prefetch</td>
<td>168.0</td>
<td>25.6</td>
</tr>
<tr>
<td>RIIII-prefetch</td>
<td>70.8</td>
<td>34.4</td>
</tr>
</tbody>
</table>
CMG performance RIRI vs. RRII

- RRRII saturates already at 8 cores
- Sharing across cores in L2 gives slight increase @ 12 cores
Comparison with other architectures
Summary

- **ECM model constructed** for single-core performance of A64FX

- **Partially overlapping** memory hierarchy → high single-core memory bandwidth (even more so with large pages)

- If performance is bad, the single-core performance is usually the culprit

- SpMV requires proper data format for efficient single-core execution

- DW kernel benefits from prefetching and OoO improvements

- **Performance modeling is invaluable** for navigating optimization efforts
Thank You.