Automatic Generation of Micro-kernels for Performance Portability of Matrix Multiplication on RISC-V Vector Processors

Second International workshop on RISC-V for HPC

Francisco D. Igual
Luis Piñuel
Universidad Complutense de Madrid

Héctor Martínez
Universidad de Córdoba

Sandra Catalán
Universitat Jaume I de Castelló

Adrián Castelló
Enrique S. Quintana-Ortí
Universitat Politècnica de València
Motivation

• High-performance BLAS implementations rely on simple micro-kernels
  • Adapted to the underlying architecture
  • Hand-written in assembly/intrinsics
  • Typically well-structured, semi-automatic development

• Automatic generation of GEMM micro-kernels for RVV
  • Basic building block for a complete Level-3 BLAS

• First experiences with C910/C906
  • Both supporting RVV 0.7.1
  • Necessary optimizations to improve performance vs. existing libraries (e.g. OpenBLAS)
Outline

1. Background on High-Performance GEMM

2. GEMM optimizations for RVV
   1. Hand-tuned
   2. Automatic generation

3. Experimental results

4. Conclusions
Background
Anatomy of a high-performance GEMM

\[ C = C + AB \]

\[ C: mxn; A: mxk; B: kxn \]
Automation of a high-performance GEMM

Cache Configuration Parameters (CCPs)

- Blocksize selection
  - $m_c, n_c, k_c$ (cache parameters)
  - $m_r, n_r$ (micro-kernel dimension)

- Analytical modeling [1]
  - From cache hierarchy features
  - Size, cacheline size, associativity
  - Matrix dimensions (skinny matrices)

High-performance GEMM (Level-3 BLAS)

GEMM micro-kernel

- Typically written in assembly/intrinsics
- Maximize register use, flops vs. memops
- Generic optimizations
  - Vectorization
  - Instruction mix/order
  - SW pipelining
  - Loop unrolling
- Automatic generation
  - Apache TVM [2]
  - Scripts


GEMM optimizations for RVV
Baseline ASM micro-kernel (4x4)

1. Vector load (vle) of column of Ar
2. Scalar load (flw) of elements of row of Br
3. Accumulation using vector-scalar (vfmac v.f)

Stage 1. Load micro-tile Cr to V.Regs.
Stage 2. Updates Cr at each iteration
Stage 3. Writes back Cr to main memory
Optimization 1: broadcasting (vfmv)

1. Vector load (vle) of a column of Ar
2. Scalar load (flw) of elements of row of Br
3. Broadcast (vfmv.v.f) to vector registers
4. Accumulation using vector-vector (vfmacc.vv)
Optimization 2: broadcasting (vrgather)

1. Vector load (vle) of a column of Ar
2. Vector load (vle) of a row of Br
3. Use vrgather.vi to splat individual elements of Br
4. Accumulation using vector-vector (vfmac.s.vv)

```
.macro LOOP_BODY_4x4
vle32.v A0, (Ar)     # Load the pr-th column of Ar into vector registers
vle32.v Btmp, (Br)   # Load the pr-th row of Br into vector registers
vrgather.vi B0, Btmp, 0 # Splat individual elements (lanes) of Br into vector registers.
vrgather.vi B2, Btmp, 2
vrgather.vi B3, Btmp, 3
vfmac.s.vv C00, A0, B0 # Vector-vector accumulation (Col. 0)
vfmac.s.vv C01, A0, B1 # Vector-vector accumulation (Col. 1)
vfmac.s.vv C02, A0, B2 # Vector-vector accumulation (Col. 2)
vfmac.s.vv C03, A0, B3 # Vector-vector accumulation (Col. 3)
.endm
```
Optimization 3: load order (B->A)

- Rearrange load order:
  - Elements of Br loaded before Ar

Stage 1. Load micro-tile Cr to V.Regs.
Stage 2. Updates Cr at each iteration
Stage 3. Writes back Cr to main memory
Optimization 4: general techniques

- Combined with previous optimizations:
  1. Loop unrolling
  2. Software pipelining
## Optimization summary

<table>
<thead>
<tr>
<th>Name</th>
<th>Load A</th>
<th>Load B</th>
<th>Load Order</th>
<th>Broadcast of B</th>
<th>Accumulation</th>
<th>Pipelining</th>
<th>Unroll</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTO-Baseline</td>
<td>Vector</td>
<td>Scalar</td>
<td>AB</td>
<td>–</td>
<td>VF</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>AUTO-Op1</td>
<td>Vector</td>
<td>Scalar</td>
<td>AB</td>
<td>VFMV</td>
<td>VV</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>AUTO-Op2</td>
<td>Vector</td>
<td>Vector</td>
<td>AB</td>
<td>Gather</td>
<td>VV</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>AUTO-Op3</td>
<td>Vector</td>
<td>Vector</td>
<td>BA</td>
<td>Gather</td>
<td>VV</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>AUTO-Op4</td>
<td>Vector</td>
<td>Vector</td>
<td>BA</td>
<td>Gather</td>
<td>VV</td>
<td>Yes</td>
<td>Yes (2)</td>
</tr>
</tbody>
</table>
Automatic micro-kernel generation

- Previous optimizations exhibit a very regular structure
- Python generator routines parametrized by:
  - Micro-kernel dimensions \((mr, nr)\)
  - Vector length \((vl)\)
  - Datatype
- Generator driver:
  1. Parametrized by \((mr, nr)\)
  2. Receives desired optimizations
  3. Applies analytical modeling for CCPs \((mc, nc, kc)\)
  4. Generates GEMM codes that apply partitioning + packing + optimized micro-kernel
Experimental results
Platforms and experimental setup

**XuanTie C910**
- T-HEAD 1520 SoC
  - 4 x C910@1.85GHz
  - 12-stage, out-of-order superscalar
  - 2 vector slices (pipelines), 128-bit (VLEN)
  - RVV 0.7.1
- **L1**: 64 KiB, 2-way. **L2**: 1 MiB, 16-way

**XuanTie C906**
- Allwinner D1 SoC
  - 1 x C906@1GHz
  - 5-stage, in-order
  - 1 vector slice, 128-bit (VLEN)
  - RVV 0.7.1
- **L1**: 32 KiB, 4-way

**Compiler:**
- GCC toolchain 10.2 (port by T-HEAD, versión 2.6.1)
- Flags: -march=rv64imafdcv0p7_zfh_xtheadc -mabi=lp64d, and -mtune=c910|c906

**OpenBLAS:**
- Commit 23693f0
- Two configurations: RVV generic, C910
Experimental conditions

1. FP32, single core
2. 8x8 and 16x4 micro-kernels (examples)
3. Square matrices ($m=n=k$)
4. Resnet-50 (rectangular matrices)

<table>
<thead>
<tr>
<th>Layer type id.</th>
<th>Layer numbers in ResNet50 v1.5</th>
<th>$m$</th>
<th>$n$</th>
<th>$k$</th>
<th>Layer type id.</th>
<th>Layer numbers in ResNet50 v1.5</th>
<th>$m$</th>
<th>$n$</th>
<th>$k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>001</td>
<td>1,605,632</td>
<td>64</td>
<td>147</td>
<td>11</td>
<td>080</td>
<td>100,352</td>
<td>256</td>
<td>512</td>
</tr>
<tr>
<td>2</td>
<td>006</td>
<td>401,408</td>
<td>64</td>
<td>64</td>
<td>12</td>
<td>083/095/105/115/125/135</td>
<td>25,088</td>
<td>256</td>
<td>2,304</td>
</tr>
<tr>
<td>3</td>
<td>009/021/031</td>
<td>401,408</td>
<td>64</td>
<td>576</td>
<td>13</td>
<td>086/098/108/118/128/138</td>
<td>25,088</td>
<td>1,024</td>
<td>256</td>
</tr>
<tr>
<td>4</td>
<td>012/014/024/034</td>
<td>401,408</td>
<td>256</td>
<td>64</td>
<td>14</td>
<td>088</td>
<td>25,088</td>
<td>1,024</td>
<td>512</td>
</tr>
<tr>
<td>5</td>
<td>018/028</td>
<td>401,408</td>
<td>64</td>
<td>256</td>
<td>15</td>
<td>092/102/112/122/132</td>
<td>25,088</td>
<td>256</td>
<td>1,024</td>
</tr>
<tr>
<td>6</td>
<td>038</td>
<td>401,408</td>
<td>128</td>
<td>256</td>
<td>16</td>
<td>142</td>
<td>25,088</td>
<td>512</td>
<td>1,024</td>
</tr>
<tr>
<td>7</td>
<td>041/053/063/073</td>
<td>100,352</td>
<td>128</td>
<td>1,152</td>
<td>17</td>
<td>145/157/167</td>
<td>6,272</td>
<td>512</td>
<td>4,608</td>
</tr>
<tr>
<td>8</td>
<td>044/056/066/076</td>
<td>100,352</td>
<td>512</td>
<td>128</td>
<td>18</td>
<td>148/160/170</td>
<td>6,272</td>
<td>2,048</td>
<td>512</td>
</tr>
<tr>
<td>9</td>
<td>046</td>
<td>100,352</td>
<td>512</td>
<td>256</td>
<td>19</td>
<td>150</td>
<td>6,272</td>
<td>2,048</td>
<td>1,024</td>
</tr>
<tr>
<td>10</td>
<td>050/060/070</td>
<td>100,352</td>
<td>128</td>
<td>512</td>
<td>20</td>
<td>154/164</td>
<td>6,272</td>
<td>512</td>
<td>2,048</td>
</tr>
</tbody>
</table>
Results - C910, 8x8 microkernel, square matrices

- **Auto-Baseline vs. OpenBLAS**
  - 1.72x improvement vs. OpenBLAS RVV Generic
  - Similar performance than OpenBLAS C910
- **Auto-Op1** (bcast)
  - 2.38x improvement vs. Auto-Baseline
- **Auto-Op2** (gather)
  - 2.62x improvement vs. Auto-Baseline
- **Auto-Op3** (load reorder)
  - 2.90x improvement vs. Auto-Baseline
  - 2.59x improvement vs. C910 OpenBLAS
- **Auto-Op4** (SW pipelining)
  - 2.88x improvement vs. Auto-Baseline
Results - C910, microkernel comparison, square matrices
Results - C910, microkernel comparison, Resnet-50
Results - C906, 8x8 microkernel, square matrices

- **Auto-Baseline vs. OpenBLAS**
  - 1.32x improvement vs. OpenBLAS RVV Generic
  - 1.51x improvement vs. OpenBLAS C910
- **Auto-Op1 (bcast)**
  - 0.91x improvement vs. Auto-Baseline
- **Auto-Op2 (gather)**
  - 0.86x improvement vs. Auto-Baseline
- **Auto-Op3 (load reorder)**
  - 0.87x improvement vs. Auto-Baseline
- **Auto-Op4 (SW pipelining)**
  - 0.78x improvement vs. Auto-Baseline
Results - C906, 8x8 microkernel, square matrices
Results - C906, microkernel comparison, Resnet-50
Conclusions
Conclusions

• The development of micro-kernels for vector architectures is a well-structured task, with potential for automation

• Yielding a rich family of optimized micro-kernels enables the use of the most suitable depending on the underlying architecture, even when all of them implement a common ISA (in our case, RISC-V + RVV)

• Performance results for the C910/C906 demonstrate remarkable performance benefits compared with state-of-the-art BLAS implementations (OpenBLAS)
Automatic Generation of Micro-kernels for Performance Portability of Matrix Multiplication on RISC-V Vector Processors

Second International workshop on RISC-V for HPC

Francisco D. Igual
Luis Piñuel
Universidad Complutense de Madrid

Héctor Martínez
Universidad de Córdoba

Sandra Catalán
Universitat Jaume I de Castelló

Enrique S. Quintana-Ortí
Universitat Politècnica de València

Adrián Castelló
Backup slides
C910. STREAM – Roofline model

STREAM – C910 vs. Nvidia ORIN (Cortex A78AE) vs. Intel Atom (x7425E)
Results - C910, 16x4 microkernel optimizations. Square matrices
Results - C910, microkernel comparison, Resnet-50
Results - C906, 16x4 microkernel optimizations. Square matrices

![Graph showing performance of SGEMM on C906@1.0Ghz for 16x4 microkernel with square matrices. The x-axis represents problem dimension (m=n=k) ranging from 0 to 3000, and the y-axis represents GFLOPS ranging from 0 to 2. The graph compares different implementations including Auto-Baseline, Auto-Op1: Broadcast, Auto-Op2: Gather, OpenBLAS - RVV Generic, and OpenBLAS - C906.]
Results - C906, microkernel comparison, Resnet-50