A Source-to-Source Vectorizer for the Wide Connex SIMD Processor

Alex Şuşu, Politehnica University of Bucharest
Motivation

- Wide SIMD is popular
  - Intel Xeon Phi: 512 bits (16 x 32)
  - ARM SVE: at most 2048 bits (64 x 32)
  - NVIDIA GP-GPU: a warp has 1024 bits (32 x 32)
- Reasons:
  - data parallel architectures scale easily – can keep pace with Moore’s law
  - better compilation support these days
  - energy efficient – smaller control unit, compiler issues vector code
Key Idea

- Compile **efficiently** for Connex SIMD processor
  - simple applications of **arbitrary** sizes
    - an array has size specified as a program variable
  - with full automation: from C/C++ sequential programs to coordinated vector assembly
  - for Connex of arbitrary width.
The Connex SIMD Processor #1/2

- **reconfigurable** SIMD processor designed to run efficiently BLAS
  - 32 - 4096 lanes (EUs)
    - ~50 times wider than current SIMD processors
  - hw support for sum reduction
  - 16-bit lanes --> emulate 32-bit int and float operations
    - possible to have 32-bit EUs, also FP hw support
  - easy to tweak Verilog code

- implemented in FPGA
  - also as ASIC – CA1024 for HDTV
The Connex SIMD Processor #2/2

- Connex loosely coupled with CPU - similar to NVIDIA GP-GPU
- separate memory space
  - scratchpad memory: Local Storage (LS)
  - Instruction and Reduction FIFOs
- predictable performance
  - processor handles each instruction in 1 cycle
  - shift vector unit
<table>
<thead>
<tr>
<th>Category</th>
<th>Opineca Connex Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( R(d) = R(s_1) + R(s_2); // add )</td>
</tr>
<tr>
<td></td>
<td>( R(d) = R(s_1) - R(s_2); // sub )</td>
</tr>
<tr>
<td></td>
<td>( R(d) = R(s_1) + R(s_2) + carry; // addc )</td>
</tr>
<tr>
<td></td>
<td>( R(d) = R(s_1) - R(s_2) + carry; // subc )</td>
</tr>
<tr>
<td></td>
<td>( R(s_1) \times R(s_2); // mult, initialize multiplication )</td>
</tr>
<tr>
<td></td>
<td>( R(d_0) = \text{MULT}_\text{LOW}() ) and</td>
</tr>
<tr>
<td></td>
<td>( R(d_n) = \text{MULT}_\text{HIGH}() )</td>
</tr>
<tr>
<td></td>
<td>/* multi or multi, get 16-bit lower and higher part of result of multiplication */</td>
</tr>
<tr>
<td>sum-reduce</td>
<td>( \text{REDUCE}(R(s)); // red, result has 32 bits )</td>
</tr>
<tr>
<td>bitwise</td>
<td>( R(d) = \neg R(s); // not )</td>
</tr>
<tr>
<td>logical</td>
<td>( R(d) = R(s_1) \mid R(s_2); // or )</td>
</tr>
<tr>
<td></td>
<td>( R(d) = R(s_1) &amp; R(s_2); // and )</td>
</tr>
<tr>
<td></td>
<td>( R(d) = R(s_1) \oplus R(s_2); // xor )</td>
</tr>
<tr>
<td></td>
<td>( R(d) = R(s_1) \ll R(s_2); // shl )</td>
</tr>
<tr>
<td></td>
<td>( R(d) = R(s_1) \gg R(s_2); // shr )</td>
</tr>
<tr>
<td></td>
<td>( R(d) = \text{SHRA}(R(s_1), R(s_2)); // shra )</td>
</tr>
<tr>
<td></td>
<td>( R(d) = \text{ISHRA}(R(s), \text{imm}); // ishra )</td>
</tr>
<tr>
<td></td>
<td>( R(d) = \text{POPCNT}(R(s)); // popcount, bits sum )</td>
</tr>
<tr>
<td>logical</td>
<td>( R(d) = R(s_1) == R(s_2); // eq )</td>
</tr>
<tr>
<td></td>
<td>( R(d) = R(s_1) &lt; R(s_2); // lt )</td>
</tr>
<tr>
<td></td>
<td>( R(d) = \text{ULT}(R(s_1), R(s_2)); // ult )</td>
</tr>
<tr>
<td>shift vector</td>
<td>/* load in shift register register ( R(s_1) ), then shift left/right by ( R(s_2) ) positions */</td>
</tr>
<tr>
<td></td>
<td>( \text{CELL_SHL}(R(s_1), R(s_2)); // cellshl )</td>
</tr>
<tr>
<td></td>
<td>( \text{CELL_SHR}(R(s_1), R(s_2)); // cellshr )</td>
</tr>
<tr>
<td></td>
<td>/* load in ( R(d) ) the current value of the shift register */</td>
</tr>
<tr>
<td></td>
<td>( R(d) = \text{SHIFT_REG}; // ldsh )</td>
</tr>
<tr>
<td>load/store</td>
<td>( R(d) = \text{LS}[	ext{imm}]; // \text{read}, imm-addr load )</td>
</tr>
<tr>
<td></td>
<td>( R(d) = \text{LS}[R(s)]; // \text{read}, indirect load )</td>
</tr>
<tr>
<td></td>
<td>( \text{LS}[\text{imm}] = R(s); // \text{write}, imm-addr store )</td>
</tr>
<tr>
<td></td>
<td>( \text{LS}[R(s)] = R(s); // \text{write}, indirect store )</td>
</tr>
<tr>
<td></td>
<td>( R(d) = \text{INDEX}; // idx, load index of each lane )</td>
</tr>
<tr>
<td></td>
<td>( R(d) = \text{imm}; // \text{load}, load immediate )</td>
</tr>
<tr>
<td>predication</td>
<td>( \text{EXECUTE_IN_ALL}(...); // endwhere )</td>
</tr>
<tr>
<td></td>
<td>( \text{EXECUTE_WHERE_EQ}(...); // whereeq )</td>
</tr>
<tr>
<td></td>
<td>( \text{EXECUTE_WHERE_LT}(...); // wherelt )</td>
</tr>
<tr>
<td></td>
<td>( \text{EXECUTE_WHERE_CRY}(...); // wherecry )</td>
</tr>
<tr>
<td>loop with</td>
<td>( \text{REPEAT}(\text{imm}); // set/e. \text{mm} \in [0..1023] )</td>
</tr>
<tr>
<td>counter</td>
<td>( \text{END_REPEAT}; // jmp/end址 )</td>
</tr>
<tr>
<td></td>
<td>( \text{NOF}; // nop )</td>
</tr>
</tbody>
</table>
Opincaa Programming Model

• Opincaa - run-time C++ assembler and coordination library for Connex [3]
  – expresses naturally more general computation for the accelerator
    • Connex ASM limitations: no loop nesting, no scalar registers, no function call mechanism
  – write in one C++ program: nonvectorized code together with Connex assembly (in C++).

• Tasks of Opincaa:
  – transfer data to/from the LS memory
  – assemble at runtime and dispatch kernels to Connex
  – brings reduction results from Connex on CPU (also synchronization).
Example Compiled Program

```
short SumReduce(short *C, int N) {
    short sum = 0;

    for (int i = 0; i < N; ++i)
        sum += C[i];

    return sum;
}
```

Listing 1. C source program for array reduction

```
short SumReduce(short *C, int N) {
    short sum = 0;

    int numElemsAccessedC = N;

    // Blocking DMA transfer from CPU mem. to Connex LS mem.
    connexGlobal->writeDataToConnexPartial(C,
        /* numVectors */
        (int)ceil((float)numElemsAccessedC) / CONNEX_VL,
        /* actual num elems written */ numElemsAccessedC,
        /* offset */ 0);

    _BEGIN KERNEL(BatchNumberGlobal);
    EXECUTE_IN_ALL(
        R(0) = 0;
        R(2) = 1;
        R(1) = R(0);
        R(3) = R(0);
        REPEAT_X_TIMES((N / CONNEX_VL) + ((N % CONNEX_VL) > 0));
            R(4) = LS[R(3)];
            R(3) = R(3) + R(2);
            R(1) = R(4) + R(1);
        END_REPEAT;
        REDUCE R(1);
    );
    _END KERNEL(BatchNumberGlobal);

    connexGlobal->executeKernel(
        TEST_PREFIX + to_string(BatchNumberGlobal));
    sum = connexGlobal->readReduction();

    return sum;
}
```

Listing 2. Opincaa program compiled from Listing 1. Note that it works for arbitrary N and CONNEX_VL.
Opincaa LLVM Compiler

- **Back-end**
  - extend LLVM to support more than 32 lanes
  - emulation of arithmetic operations for i32, fp16
  - support symbolic scalar immediate operands

- **Extend LoopVectorize module**
  - generate Opincaa coordination calls
  - use SRA to retrieve number of array elements accessed; SCEV for loop trip counts
    - recover them as symbolic scalar expressions from LLVM IR to C/C++
  - simple symbolic memory allocator
  - generate efficient 2-level loop nests.
Experiments

- Connex-128 (128 i16 EUs at 100 MHz on Xilinx Zynq-7000 EPP FPGA) w.r.t. 1 ARM core (667 MHz, 128-bit NEON) [2]
  - Note: Connex uses Zynq-7000 DSPs for i16 multiplication

Performance speedups of kernels with element types i16 and i32 for Connex with 128 lanes w.r.t. 1 ARM core

Performance speedups when running on Connex processors with number of lanes 32-1024 w.r.t. 1 ARM core
Tools To Get

• Opincaa library
  – also cycle-accurate simulator for Connex of custom width
    • prototype new Connex functionality
  – source code available at
    http://gitlab.dcae.pub.ro/research/ConnexRelated/opincaa

• OpincaaLLVM compiler
  – http://gitlab.dcae.pub.ro/research/ConnexRelated/OpincaaLLVM
Future Work

• Memory optimizations:
  – loop tiling
  – array allocation with graph coloring

• Compiler support for the shift vector unit

• Compile non-regular code on Connex.
Acknowledgments

- prof. Gheorghe M. Ștefan
- Radu Hobincu, Lucian Petrică, Călin Bîră
Thank you! Questions?
References


