TECHNISCHE
UNIVERSITÄT
WIEN

Multidimensional Match, Transform and Replace: Tensorization in VADL

Benedikt Huber

Intro

- Single dimensional vectorizers are well established
  - SIMD instructions
- How can we make use of multidimensional optimized target implementations?
  - Optimized library functions like `gemm` in OpenBLAS
  - Matrix or tensor instructions like `TDPBUUD` in Intel AMX
    This is relevant for VADL.
- Accessed via compiler intrinsics or library calls
  - Not portable and error prone
- Goal of MMTR
  - Identify and transform nested loops and replace them
  - Only based on a VADL description of the optimized target implementation

Overview

- An optimized target implementation often is semantically equivalent to
  - A perfectly nested loop
  - With a dense and rectangular iteration domain
  - A constant step size
  - A loop invariant iteration count
  - A small loop body
  - Without complex control flow
- Suitable for
  - Polyhedral model
  - Pattern matching
- MMTR implemented as
  - Extension pass in Polly
  - LLVM 19
- Two modes for dimensions
  - parametrized
  - fixed Relevant for VADL
- Convenient Tablegen description

MMTR

- Each target specific MPattern consists of
  - InstrPat: Instruction Tree Pattern
  - Dimensions
  - MemAccPats: List of array accesses
  - ReplacementBlock: LLVM IR to be inserted

Example: Input

```
void matmul(unsigned l, unsigned m, unsigned n,
            TY z[restrict l][m],
            TY x[restrict l][n],
            TY y[restrict n][m]) {
  for (unsigned i = 0; i < l; i++)
    for (unsigned j = 0; j < m; j++) {
      // Polly reports this as Stmt_for_body3
      z[i][j] = 0;
      for (unsigned k = 0; k < n; k++)
        // Polly reports this as Stmt_for_body8
        z[i][j] += x[i][k] * y[k][j];
    }
}
```

Listing 1: Input Program: Matrix Multiplication

Example: MPattern of Vector Matrix Multiplication

```
def MACPat :
  Store<
    Add<
      Mul<Load<A>, Load<B>>,
      Load<C>>,
    C>;

// indices are j, k, l, ...
def LA : ReadAccess<A,  [k]>;
def LB : ReadAccess<B,  [k, j]>;
def LC : ReadAccess<C,  [j]>;
def SC : WriteAccess<C, [j]>;

def VectMatMulPattern :
  MPattern<
    MACPat
  , Fixed<[6, 4]>
  , [LA, LB, LC, SC]
  , Emitter<"VectMatMul", [C, A, B, empty, strideB]>>;
```

Listing 2: MPattern

Example: Algorithm

- Match LLVM IR Instructions with the InstrPat
  and capture the needed values
- Check size and shape of instance set as reported by Polly
- Match all memory accesses
- Determine strides as reported by ArrayInfo
- Move other statements out of the loop
  Use Polly's ScheduleTree for loop fission
- Loop Tiling for Fixed Dimensions
  Again with Polly's functionality
- Check if all data dependencies are still observed
- Set the replacement mark
- During emission replace the loop with the ReplacementBlock

```
Emitter<"VectMatMul", [C, A, B, empty, strideB]>
```

Example: Algorithm

```
for (int c0 = 0; c0 < l; c0 += 1) {
  for (int c1 = 0; c1 < m; c1 += 1)
    Stmt_for_body3(c0, c1);
  for (int c1 = 0; c1 < 6 * floord(m, 6); c1 += 6)
    for (int c2 = 0; c2 < 4 * floord(n, 4); c2 += 4)

      // REPLACE_LOOP
      for (int c3 = 0; c3 <= 5; c3 += 1)
        for (int c4 = 0; c4 <= 3; c4 += 1)
          Stmt_for_body8(c0, c1 + c3, c2 + c4);

  for (int c1 = 0; c1 < 6 * floord(m, 6); c1 += 1)
    for (int c2 = -(n % 4) + n; c2 < n; c2 += 1)
      Stmt_for_body8(c0, c1, c2);
  for (int c1 = -(m % 6) + m; c1 < m; c1 += 1)
    for (int c2 = 0; c2 < n; c2 += 1)
      Stmt_for_body8(c0, c1, c2);
}
```

Listing 3: Loop Tiling

Evaluation

- Optimized target implementation was OpenBLAS on x86-64
- MPatterns for
  - Matrix Mult `cblas_sgemm` (3D)
  - Vector Matrix Mult `cblas_sgemv` (2D)
  - Dot Product `cblas_sdot` (1D)
  - Vector Addition `cblas_saxpy` (1D)

- PolyBench
  - 7 out of 30 with matches

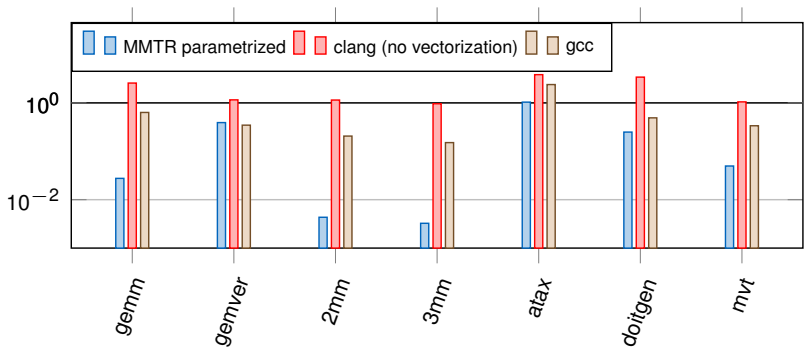

Figure: PolyBench Runtime relative to Clang with vectorization
logarithmic scale, smaller is better

- Visual Wake Words (VWW)
  - MLPerf Tiny
  - Classify images
- C code generated from TFLite model by TVM
- 10 MMTR matches
  4 in hot functions
- All matches are Vector Matrix Mult
- Reduces runtime by 66%

Vector Matrix Multiplication

```
100  constant WSize    = 4             // Bytes per Word
101  constant VEL      = 6             // Words in a Vector
102  constant VBytes   = VEL*WSize     // Bytes in a Vector
103  constant Rows     = 4             // Vector sized Rows in a Matrix
104
105  using Vector    = Word<VEL>
106  using Matrix    = Vector<Rows>
107
108  register V : Index -> Vector
109  register M : Index -> Matrix
110
111  instruction LV : frmt = { V(rd) := MEM< VBytes >(X(rs1)) }
112
113  instruction SV : frmt = { MEM< VBytes >(X(rd)) := V(rs1) }
114
115  // lm m0 x1 x2
116  instruction LM : frmt = {
117    M(rd) := forall i in (Row-1)..0 tensor
118      //              addr   + i * stride
119      MEM< VBytes >(X(rs1) + i * X(rs2))
120  }
121
122  // vmm v0 v1 m2
123  instruction VMM : frmt = {
124    let v = V(rs1) in                          // Input vector has 'Row' elements
125    let m = M(rs2) in
126    let mult = forall i in (Row-1)..0, j in (VEL-1)..0 tensor
127      v(i) as UInt<32> * m(i)(j) in
128    let mac =  forall j in (VEL-1)..0 tensor  // Output vector has 'VEL' elements
129              (forall i in (Row-1)..0 fold + with mult(i)(j)) in
130    V(rd) := mac
131  }
```

Conclusion

- Future work
  - Extend handling of access patterns
  - Generalize iteration domains
  - Better cost model
  - Emit MMTR description from OpenVADL LCB
- MMTR source code available at
  - `github.com/OpenVADL/llvm-project/tree/mmtr`
- Contact
  - `benedikt.huber@tuwien.ac.at`