Abstract

Sparse matrix formats implemented in a low-level imperative language can be hard to code, debug and understand. Furthermore, it is impractical to verify such implementations. Conversion into sparse formats and multiplication of matrices can be thought of as a sequence of data layout transformations. We present translation into low-level code of a high-level functional language that is restricted enough to be amenable to verification and efficient compilation, yet expressive enough to capture the dataflow of format conversion and matrix-vector multiplication. Our generated sequential code for SpMV in CSR format has \( 1.1 \times \) median slowdown compared to a hand-written reference code, and the automatically generated parallel code, which uses OpenMP, provides up to \( 1.78 \times \) speedup on 2 cores and up to \( 2.43 \times \) speedup on 4 cores.

1. Introduction

Sparse matrix-vector multiplication (SpMV) is an operation intensely used in diverse applications in scientific computing, engineering, economic modeling, data mining, as well as in other domains [4]. A variety of formats exists in order to compress sparse matrices and reduce the memory footprint of matrices while increasing the performance of this memory-bound operation.

Consider for instance the conversion from a dense representation of a matrix to one of the more complex sparse storage formats, the jagged diagonal format (JAD). Following is the C code for achieving this task. It is clear that this code doesn’t let one understand easily the JAD data structure.

```c
/* JAD compression. */
lenperm (M, P); /* obtain row permutation */
for (d = k = 0; d < n; d++) {
    kk = k;
    for (i = 0; i < n; i++) {
        for (j = nz = 0; j < m; j++)
            if (M[P[i]][j])
                if (++nz > d) break;
        if (j < m) {
            J[k] = j;
            V[k] = M[P[i]][j];
            k++;
        }
    }
    if (k == kk) break;
    D[d] = k;
}
```

As an alternative, conversion to JAD can be expressed as a sequence of high-level transformations, as illustrated in Figure 1. The transition from Figure 1(a) to Figure 1(b) corresponds to applying a simpler compression format which is CSR; the next transformation, which leads to Figure 1(c) consists in sorting the compressed rows by length, along with providing the permutation \( P \); finally, the compressed rows are transposed to obtain the compressed diagonals.

The following program captures the transformations mentioned above. ; is the function composition operator, which composes the sequence.

```python
def jad:
    csr; lenperm; (fst, trans(snd))

lenperm, which implements the second of the three transformations is defined as:

def lenperm:
    map ((len, (#, id))); sort; rev; map (snd); unzip
```

The rest of the report is structured as follows: Section 2 presents an overview of a subset of the language constructs using the CSR SpMV example, translation from high-level programs to C is elaborated in Section 3, Section 4 presents evaluation results, Section 5 discusses future work and Section 6 concludes.

2. Language overview

Our high-level functional language consists of a limited number of higher-order functions (functions that take other functions as arguments) such as map and filter, a number of built-in library functions, and the function composition operator ;.

We next present the high-level data transformations corresponding to the CSR compression and SpMV, and the language constructs that are needed to express these in our language.

Figure 2 illustrates the phases of data transformation from a dense representation to the CSR representation. CSR consists of compressing each row of the matrix by filtering out zero elements and pairing each remaining element with its column index. Conversion to CSR can thus be seen as a sequence of two transformations, first enumerating each row of the dense matrix, which, given Figure 2(a) yields Figure 2(b), and then filtering out the zero elements in each row, giving the final result in Figure 2(c).

The first transformation, enumerating all rows, is a sequence mapping. A sequence mapping is achieved via the higher-order function map, which applies a function to every element of an input list. This phase of CSR conversion is therefore expressed as

```python
map ((len, (#, id))); sort; rev; map (snd); unzip
```
map (enum), where enum is a built-in function that, given a list, pairs each of its elements with its index.

The second transformation requires the introduction of another higher-level function, the filter construct. Given an input list, this function evaluates a predicate on each element of the list to keep only those that satisfy the predicate. In order to obtain the final result, we filter out in each row those pairs whose second element is zero. This amounts to the following code:

```
def csr: map (enum; filter (snd != 0))
```

Consecutive maps can be merged together, the resulting code is:

```
def csr: map (enum; filter (snd != 0))
```

Moving on to the CSR SpMV, we describe the operation as:

(a) for each element, multiply the value by the corresponding entry of the source vector; (b) sum all the resulting products in a row.

This is captured by the following program.

```
def csrsv A, x: A; map (map (snd * x[fst]); sum)
```

### 3.3 Optimizations

Inspecting the naïvely-generated code, we were able to come up with a number of optimizations in order to approach the performance of a hand-written reference code for a given example. These optimizations can be summarized as follows:

- Reuse of allocated buffers inside loops corresponding to map and filter constructs.
- Upon encountering a map(f) followed by a reduction function, e.g. a sum or a product over all elements of a list, instead of producing the output list of the map and then reducing it, the result is accumulated inside the loop as the map progresses.

### 3.4 Parallelism

Automatic parallelization follows from the fact that map constructs are naturally data-parallel. Our compiler currently generates parallelized versions for each map that is not contained in another, and whose output length is known in advance. Parallelism is achieved by using OpenMP parallel regions. Upon entry to a parallel map region, a partition of elements of the input list is computed, such that each thread in the work group
is assigned to apply the sequence mapping to one part. The partitioning function can be specialized for different types to achieve load balancing.

In our current implementation of the compiler, the generated code for CSR SpMV achieves data parallelism by parallelizing the outermost map construct, which corresponds to dividing the matrix into row blocks. The partition is computed such that each row block contains a nearly equal number of nonzeros.

4. Evaluation

We evaluate the performance of our generated code for CSR SpMV, comparing it to a hand-written naive CSR SpMV implementation. Following is the reference code in C that implements the operation.

```c
/* CSR multiplication. */
void ref_csrmv (int m, double *value, int *col_idx,
               int *row_start, double *x, double *y)
{
    int i, jj;

    /* loop over rows */
    for (i = 0; i < m; i++) {
        double y_i = 0.0;
        /* loop over non-zero elements in row i */
        for( jj = row_start[i]; jj < row_start[i+1];
            jj++, col_idx++, value++ ) {
            y_i += value[0] * x[col_idx[0]];
        }
        y[i] = y_i;
    }
}
```

For our performance study, we consider a set of 8 unstructured non-symmetric matrices among those studied by Williams et al. [5]. The basic properties of these matrices can be observed in Figure 3.

The benchmarking environment is a 2.3 GHz single socket quad-core AMD Opteron processor, with 8GB of memory.

The performance results are presented in Figure 4. The generated sequential code has a maximum 1.17x slowdown compared to the hand-written code, with a 1.09x median slowdown. On 2 cores, the generated code has up to 1.73x speedup with a median speedup of 1.68x. On 4 cores, up to 2.43x speedup is achieved, with a median speedup of 2.11x.

5. Future work

The next step in code generation will be experimenting with register blocking and cache blocking in CSR SpMV, which are sequential optimizations [1, 2]. Register blocking proves useful when the input matrix has a natural dense subblock structure, and cache blocking is beneficial for the cases where the source vector is too large to reside entirely in cache.

Register blocking can be implemented using the Blocked CSR (BCSR) format. In our high-level abstract representation, a matrix in BCSR format will be of type `[(int, [[float]])]`, i.e. a two-dimensional list of pairs of indices and dense subblocks. Figure 5 illustrates the phases of construction for this format: (a) the dense matrix is partitioned into fixed-size register blocks, small enough to permit the whole block to reside in register while it is being processed; (b) the subblocks are enumerated and the ones that don’t contain any non-zero elements are filtered out.

Consider conversion to BCSR as a sequence of high-level transformations: It amounts to invoking the built-in block function to transform a dense matrix $A$ of size $m \times n$ to an $m \times n$ matrix of $r \times c$ dense blocks followed by pairing these blocks with their column indices using `map(enum)` and filtering out the all-zero ones.

```c
def bcsr(A):
    block (r, c, A);
    map (enum;
        filter (snd; map (id != 0); disj); disj))
```

Given a $r \times c$-BCSR matrix $A$ and a dense vector $x$, multiplication can be conceptually described as follows: first, we bind the names $B$ and $l$ to each dense block and its index respectively, then we perform dense matrix-vector multiplication (denoted `densemv`) on each block and the corresponding $c$-sub-vector of $x$. The latter is obtained by breaking $x$ into a list of $c$-vectors using `block (c, x)` and selecting the appropriate one. The result vectors in a row block are summed, and the final result is obtained by concatenating the result sub-vectors from all row blocks.

```c
def bcsrsv (A, x):
    A; map (map (l, B; (B, block (c, x[l])); densemv); sum); concat
```

A further improvement to register-blocked CSR format consists in reordering the columns of the matrix to permute its nonzeros into contiguous locations, the problem being formulated as the well-known traveling salesman problem [3]. This optimization can be integrated in our case by providing the permutation as input to the program.

6. Conclusion

In this report, we presented low-level code generation for high-level functional sparse matrix codes, which allow to capture operations such as sparse matrix construction and multiplication as a sequence of high-level transformations.

Our translation to C code of naïve CSR SpMV has 1.09x median slowdown compared to a hand-written version, and automatic parallelization of the code helps attain up to 1.73x speedup on 2 cores and up to 2.43x on 4 cores.

References

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Rows</th>
<th>Columns</th>
<th>Nonzeros</th>
<th>Nonzeros/Row</th>
</tr>
</thead>
<tbody>
<tr>
<td>mac_econ_fwd500</td>
<td>206500</td>
<td>206500</td>
<td>1273389</td>
<td>6.1</td>
</tr>
<tr>
<td>mc2depi</td>
<td>525825</td>
<td>525825</td>
<td>2100225</td>
<td>3.9</td>
</tr>
<tr>
<td>qcd5_4</td>
<td>49152</td>
<td>49152</td>
<td>1916928</td>
<td>39.0</td>
</tr>
<tr>
<td>rail4284</td>
<td>4284</td>
<td>1092610</td>
<td>11279748</td>
<td>2632.9</td>
</tr>
<tr>
<td>rma10</td>
<td>46835</td>
<td>46835</td>
<td>2374001</td>
<td>50.6</td>
</tr>
<tr>
<td>scircuit</td>
<td>170998</td>
<td>170998</td>
<td>958936</td>
<td>5.6</td>
</tr>
<tr>
<td>webbase-1M</td>
<td>1000005</td>
<td>1000005</td>
<td>3105536</td>
<td>3.1</td>
</tr>
</tbody>
</table>

**Figure 3.** Matrix suite

\[
\begin{pmatrix}
\begin{pmatrix} a & 0 \\ b & c \end{pmatrix} & \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \\
\begin{pmatrix} 0 & 0 \\ 0 & d \end{pmatrix} & \begin{pmatrix} 0 & 0 \\ 0 & e \end{pmatrix}
\end{pmatrix}
\left(\begin{pmatrix} 0, \\ a \\ b \\ c \end{pmatrix}\right)
\begin{pmatrix} 0, \\ 0 \\ 0 \\ d \end{pmatrix}
\left(\begin{pmatrix} 1, \\ 0 \\ 0 \\ e \end{pmatrix}\right)
\]

**Figure 5.** Example $2 \times 2$ blocked matrix representation.