Reprinted from the

Proceedings of the GCC Developers' Summit

June 17th–19th, 2008
Ottawa, Ontario
Canada

Middle-End Array Expressions

Richard Guenther
SUSE Labs
rguenther@suse.de

Abstract

We are implementing support for array objects as first-class citizens of the tree middle-end. This allows to retain expressions that operate on array objects, such as allowed by the Fortran 9x language, to survive until the high-level loop transformation passes. Those benefit from less work on loop and dependence analysis. Lowering those expressions leveraging work done by the GRAPHITE project allows to do a better job of what is currently done by the frontend and allows faster code to be generated.

As a design and implementation proposal we expect this to be a medium term project as for utilization it requires modifications of the Fortran frontend as well as merging GRAPHITE and/or fallback lowering code in the middle-end to handle all expressions on array objects as allowed by the Fortran standard.

We also present a proposal for a GCC extension to allow utilizing the new facility from C and C++ using VLA types and builtin functions.

1 Introduction

In recent years we have seen a increased focus on high-level loop optimization development within GCC. With the introduction of the Tree SSA framework [3] frameworks for the analysis of scalar variables in loops [2] and data dependency analysis [2] have been contributed to GCC.

High-level loop optimizations such as loop nest optimizations [2] heavily depend on these analyses. With the GRAPHITE project [4] the dependence on properly analyzable loops will increase more.

Programming languages such as Fortran90 feature high-level operations on whole arrays that are easy to analyze from a data-dependency point of view as loop structure and control flow is implicit in these operations. GFortran unfortunately is required to lower these high-level constructs to GIMPLE loops by means of scalarization, making the loop structure explicit and the data-dependencies possibly harder to analyze.

We are trying to address this shortcoming of the GIMPLE intermediate language by allowing these high-level operations to persist in a canonical form until after the high-level loop optimization phase. This should make data-dependence analysis easier and expose more loops to high-level loop transformations in the end resulting in faster code.

The goal is to be able to represent all of the Fortran90 high-level operations on arrays in the GIMPLE intermediate language (IL) including commonly used intrinsics. The representation should be simple and extensible and expose optimization opportunities to the existing scalar optimizers in GCC.

For experimentation purposes we are proposing a simple way to expose the new GIMPLE features to the C and C++ front-ends by means of introducing new GCC-specific builtin functions.

A lowering pass will perform the necessary scalarization at some point of the tree pass pipeline until the GRAPHITE framework can be leveraged to do this as part of its optimized GIMPLE code creation phase.

In Section 2 we will outline the proposal in detail. After outlining the current capabilities of the GIMPLE middle-end regarding arrays, we propose a first extension to allow full lowering of array type properties to GIMPLE introducing register temporaries of array type. After this we outline the specification for high-level operations on GIMPLE arrays. The lowering procedure is outlined in Section 3. The GCC specific builtins that make these features accessible from C and C++ are specified in Section 4.

In Section 5 we give an outlook to possible optimizations performed on the high-level representation and discuss the connection to the GRAPHITE project.

2 Arrays in GIMPLE

We propose to allow GIMPLE registers of array type. This allows expressing loads and stores of (sub-)arrays with a single statement and exposes the resulting array temporaries in the SSA web. As arrays in Fortran array expressions are generally of variable size, this includes variable length array (VLA) registers with all the complexity required to handle them.

We propose to represent expressions with array value and/or operands by introducing an indexing operator on an array that defines a scalar placeholder. These placeholders can be used to express element-wise computations. As complementary operator we propose a de-indexing operator that defines an array register by means of a scalar (placeholder) operand.

We finish the set of new operators by a contraction operator that reduces the array dimension by summing over all elements in a specified set of dimensions.

The proposed scheme resembles tensor component notation commonly used in mathematics and physics. For example, given a tensor[1] T of rank two, T_{ij} specifies the element which can be addressed by the indices i and j. Matrix multiplication is the tensor product UV of two second rank tensors U and V with two dimensions contracted to yield a second rank result. In component notation this is written as $\sum_k U_{ik}V_{kj}$ which contracts the dimensions indexed by k and computes $(UV)_{ij}$, the elements of the matrix product.

In the following subsections we elaborate on the parts of this proposal.

2.1 Variable Length Arrays as GIMPLE Registers

Arrays in the GIMPLE intermediate languages are treated as aggregates and thus are not eligible for being rewritten into SSA form. Array variables can be indexed by means of the `ARRAY_REF` operator and are suitable for aggregate assignments.

[1]Pragmatically we treat a tensor as a multidimensional array as this is the only property we are interested in.

Variable length arrays need special attention during the lowering procedure to GIMPLE and special support from the `ARRAY_REF` operator. In GENERIC all information about the shape of an array is encoded in its type, respective the individual sub-domain array types `TYPE_DOMAIN` and its element type `TYPE_SIZE`.

The `TYPE_DOMAIN` of an array type specifies the minimum and maximum index value valid for operand one of the `ARRAY_REF` operator. The array element type `TYPE_SIZE` specifies the stride.

As for variable length arrays, both the domain and the element size are not compile-time constants; their values need to be properly preserved and made accessible during optimization. The gimplification process for this reason transforms the information in the types to operand slots in the GIMPLE operators. For aggregate copies this is done using the `WITH_SIZE_EXPR` wrapper, and for the `ARRAY_REF` indexing operator this is done by filling in its third and fourth operands which are the lower bound of the valid indices and the array element size. Expressions needed to compute these are emitted to the GIMPLE instruction stream and kept live by means of the uses in the above operands.

While the scheme for lowering VLA information outlined above works well for both aggregate copies and array indexing already present in the input program, there is not enough information for a later pass to insert new `ARRAY_REF`s into the instruction stream.

To fix this deficiency we propose a new wrapper called `VLA_VIEW_EXPR` that makes the missing information available in the GIMPLE IL. `VLA_VIEW_EXPR` is used to build a temporary VLA register object from memory specified by its first operand, a VLA, and of a shape as specified by pairs of extent and stride arguments.

```
VLA_VIEW_EXPR <vla, extentI, strideI,
..., extentN, strideN>
```

The first extent, stride pair represents the fastest varying dimension. The dimensionality of the resulting array register is the same as that of the `vla` operand and the number of extent, stride operand pairs. `VLA_VIEW_EXPR` is dumped as `VLA` with the first operand (the data operand) in parentheses after the shape specification.

As an example consider

```
void func (float *x, float *y,
           int n, int m) {
  float areg[n][m];
  areg = VLA <m, 1, n, m> (*x);
  VLA <m, 1, n, m> (*y) = areg;
```

This represents an array load from `*x` to an VLA register `areg` that is exposed for further use in the IL. The second statement represents an array store to `*y` from the VLA register `areg`. The loads make a semantic detail explicit which is that in Fortran and GIMPLE in an expression involving arrays all loads are carried out before all stores. This gives expressions with overlapping source and destination arrays well-defined semantics[2].

`VLA_VIEW_EXPR` uses a variable length tree and thus joins `CALL_EXPR` in the `tcc_vl_exp` tree code class. The result type of `VLA_VIEW_EXPR` is a variable length array type with extents as specified by the tree operands. The result is a gimple register which is suitable for rewriting into SSA form.

You can also extract parts of an array into a gimple register by adjusting the VLA operand and the extent, stride pairs like for example with

```
float A[n][m];
float tmp[4][4];
tmp = VLA <4, 2, 4, 2 * m> (*&A[i][j])
```

which extracts a four times four piece of A starting at the position specified by i and j skipping every second element.

To index an array register a (recursive) `ARRAY_REF` tree has to be built indexing the original VLA object from the defining `VLA_VIEW_EXPR`. This can be done by looking at the defining statement of the array register SSA name. The `VLA_VIEW_EXPR` defining it contains the necessary information that can be put into the fourth operand of the `ARRAY_REF` trees.

Note that the array registers have zero-based indices, in particular selecting a part of an VLA with `VLA_VIEW_EXPR` includes encoding the origin in the VLA operand. If the need arises the offset of the origin could also be explicitly specified as operands of the `VLA_VIEW_EXPR` as using the third operand of the `ARRAY_REF` operator allows for this.

[2]The currently implemented scalarizer does not handle this case correctly.

2.2 High-Level Operations on Arrays

There are various kinds of high-level operations that we want to represent in a canonical way within GIMPLE. In particular the goal is to leverage existing infrastructure for scalar optimizations as they would apply to the lowered form with loops and scalar operations.

We concentrated on the following common operations:

1. Element-wise operations `A * B`, $(AB)_{ij} = A_{ij}B_{ij}$
2. Mixed scalar and array operations `s * A`, $(sA)_{ij} = sA_{ij}$
3. Contractions such as the scalar product of two vectors `DOT_PRODUCT(u,v)`, $uv = \sum_i u_i v_i$
4. Tensor products $(uv)_{ij} = u_i v_j$
5. Mixed tensor product and contraction such as matrix multiplication `MATMUL(A,B)`, $(AB)_{ij} = \sum_k A_{ik}B_{kj}$

Thus the following new operations are proposed for the GIMPLE intermediate language. First the operation to represent indexing of an array `VLA_IDX_EXPR`

```
Aij...n = VLA_IDX_EXPR <A, i, j, ..., n>
```

which takes an array register as its first operand and one integral register or constant for each rank as further operands, starting with the value for the fastest varying index. `VLA_IDX_EXPR` produces a scalar placeholder of array element type representing the array element indexed as specified during the loop iteration. `VLA_IDX_EXPR` is dumped as `VLA_IDX` with the array register operand in parentheses after the index specification.

The reverse operation `VLA_RIDX_EXPR` has the same constraints but takes a gimple value as its first operand and produces an array temporary object.

```
A = VLA_RIDX_EXPR <Aij...n, i, j, ..., n>
```

Note that the first operand can be a scalar placeholder representing different values in each loop iteration but also a constant or loop invariant value. Indices as specified in the index operands can be used to compute the value that represents the array elements. `VLA_RIDX_`

EXPR is dumped as VLA_RIDX with the data placeholder dumped in parentheses after the index specification.

Indices valid for VLA_IDX_EXPR are constants or registers that are affine combinations of indices introduced by VLA_RIDX_EXPR or the later-specified VLA_DELTA_EXPR.

The scalar placeholders can be operated on like on any other scalar GIMPLE temporary. In particular VLA_IDX_EXPR and VLA_RIDX_EXPR are sufficient to implement element-wise operations on arrays, mixed scalar and array operations, and tensor products. Note that to commit a temporary array result to memory a store via a VLA_VIEW_EXPR is still necessary.

For the remaining operations we require a contraction operator. If you re-write the tensor component notation to make the summation implicit following *Einstein summation convention* then the matrix multiplication simply becomes $(AB)_{ij} = A_{ik}B_{kj}$. As implicitly induced operations are not suitable for the GIMPLE IL we can re-write this slightly to $(AB)_{ij} = A_{ik}B_{lj}\delta_{kl}$ using the *Kronecker* tensor δ[3] to induce the contraction of the dimensions indexed by k and l—thus, expanding the above again, computing $(AB)_{ij} = \sum_k \sum_l A_{ik}B_{lj}\delta_{kl}$. Note that the double summation collapses to one caused by the properties of the δ tensor.

Following this scheme we introduce the VLA_DELTA_EXPR operator that takes a scalar placeholder as its first operand, the common extent of the contracted dimensions as its second operand, and any number of further gimple variables as indices to contract.

```
Bij...n = VLA_DELTA_EXPR <Aij...n,
                          extent, indices...>
```

Where Bij...n does not include indices specified in the list of indices to contract. VLA_DELTA_EXPR is dumped as VLA_DELTA with the scalar placeholder operand dumped in parentheses after the index specification.

A VLA_DELTA_EXPR operator represents exactly a single loop summing over all values the scalar placeholder represents with iterating over the indices specified multiplied by one if all indices are equal and zero otherwise, thus it represents a multiplication with the Kronecker delta $\delta_{ij...n}$. The extent operand is redundant to some point, as it can be obtained by looking for the definition statement of the placeholder and taking the extent from the array register operand of the VLA_IDX_EXPR. But replicating the extent here makes life simpler for the scalarization process.

[3]The Kronecker delta δ_{ij} is one for equal i and j and zero otherwise.

With these operators available translating tensor component expressions can be done literally for example for computing the trace of a two-dimensional *n*x*n* matrix A, $\sum_i A_{ii} = A_{ii}\delta_{ii}$, as in

```
Aii = VLA_IDX <i, i> (A);
trace = VLA_DELTA <n, i, i> (Aii);
```

where actually the second i operand to VLA_DELTA is superfluous and can be omitted. A similar example, the scalar-product of two vectors U and V of length n, $u_i v_j \delta_{ij}$ would look like

```
Vi = VLA_IDX <i> (V);
Uj = VLA_IDX <j> (U);
ViUj = Vi * Vj;
s = VLA_DELTA <n, i, j> (ViUj);
```

With the scalar result s and the scalar placeholders Vi, Uj, and ViUj. A mixed vector-product and contraction example is the multiplication of two matrixes as seen in Figure 2.

Similarly translating Fortran90 array assignments is straightforward. Consider the example

```
A(1:n-1) = B(0:n-2) + B(2:n)
```

which is equal to $A_i = B_{i-1} + B_{i+1}$ and thus can be translated to

```
A = VLA <n, 1> (*a);
B = VLA <n + 2, 1> (*b);
Bi1 = VLA_IDX <i> (B);
Bi2 = VLA_IDX <i+2> (B);
Ai = Bi1 + Bi2;
A = VLA_RIDX <i> (Ai);
```

Note that in the first example we omitted the loads and stores via VLA_VIEW_EXPR. Note that in the second example the register array B includes a ghost area and, as indices start at zero, what is $i-1$ in index notation is i in the VLA_IDX expression.

2.3 Scalar Optimizations

With the proposed scheme we are exposing loads of arrays

```
D.1629_72 = VLA <m_1, 1, n_10, m_1> (*U_24)
```

as statements the SSA memory optimizers can work with. In particular, fully redundant loads can be eliminated here. The same is true for stores and the possibility to eliminate dead ones, though in practice this seems unlikely to happen.

As all intermediate computation is done on scalar types, optimizations such as constant propagation or redundancy elimination naturally apply to these parts. Likewise if-conversion can be applied to conditional parts in the scalar computations.

Applying scalar optimizations may expose opportunities for loop fusion in case a computation is used by multiple array stores (which are the statements that make a loop live). Applying scalar optimizations may reduce the depth of the loop nest if a dimension is only ever indexed with a constant index.

2.4 Correctness

The ability to easily preserve correctness is a critical part for any extension to GIMPLE. First, ordering of loads and stores has to be preserved. This is ensured on the granularity of whole arrays by the fact that loads from and stores to arrays via `VLA_VIEW_EXPR` are memory operations that are properly processed by the alias analyzer and represented in the FUD chain of virtual operands. Thus we have

```
# VUSE <SMT.4_9(D)>
D.1567_3 = VLA <4, 1, 4, 4> (*B_2);
x_6 = VLA_IDX <i_1, j_1> (D.1567_3);
D.1568_7 = VLA_RIDX <i_1, j_1> (x_6);
# SMT.4_10 = VDEF <SMT.4_9(D)>
VLA <4, 1, 4, 4> (*B_2) = D.1568_7;
# VUSE <SMT.4_10>
D.1569_8 = VLA <4, 1, 4, 4> (*B_2);
# SMT.4_11 = VDEF <SMT.4_10>
VLA <4, 1, 4, 4> (*B_2) = D.1569_8;
```

where the array stores clobber the symbol representing the whole array and array loads use it.

At the point we are creating the scalar placeholders for arbitrary elements of the array registers things get more interesting. A placeholder can be interpreted as value-number for the index operation that is unambiguously specified by the array register and the index register arguments. Thus in the above example `VLA_IDX <i_1, j_1> (D.1567_3)` identifies `D.1567_3[j_1][i_1]`. This is guaranteed by the properties of the SSA form both the array registers and the index registers are in.

Via their dependency on the loop indices recursively all uses of results of `VLA_IDX_EXPR` are dependent on a particular loop nest iteration. Following that the loop nest iterations are necessarily independent if a loop nest iteration dependent value can be value-numbered the same. Thus scalarization only needs to make sure to scalarize all index-dependent expressions.

It should be noted that the validity or definedness of the GIMPLE IL cannot be decided on a per-statement level, but instead needs to consider all statements that finally will end up in a loop nest. Usually this includes all statements from the first load to the last store of an array expression specifying the loop nest.

At scalarization time the index registers are replaced by the respective loop induction variable and accesses to the real arrays living in memory are created. Thus ordering of these loads and stores relative to other aliasing loads and stores that are not part of the loop nest is not preserved. In particular, the following is undefined GIMPLE

```
# VUSE <SMT.4_10>
D.1569_8 = VLA <4, 1, 4, 4> (*B_2);
# SMT.4_11 = VDEF <SMT.4_10>
(*B_2)[2][3] = 0.0;
# SMT.4_12 = VDEF <SMT.4_11>
VLA <4, 1, 4, 4> (*B_2) = D.1569_8;
```

as scalarization will happily re-order the loads from the array with the single scalar store. Note that the above case cannot result from a valid transformation of a well-defined GIMPLE program but only created in this undefined form from the start. Thus it is necessary to create array expression IL in a compact form not interleaved with unrelated code, but this should be no surprise. With the proposed interface from C in Section 4 below it is still easy to produce undefined GIMPLE though. Likewise killing the SSA names used for indexing (`i_1` and j_1 in the above example) in-between the array load and the array store results in invalid GIMPLE as the indices in VLA_IDX_EXPR and VLA_RIDX_EXPR need to be mapped to each other.

Note that like in Fortran the evaluation order of the loop nest is unspecified, but all loads take place before all stores. But in particular overlapping source and destination are not (yet) properly handled by the scalarizer. Both varying the evaluation order and introducing a temporary array are possible solutions applied by the scalarizer of the GFortran frontend.

Control flow can in theory be handled fine by the proposed GIMPLE extension, though the implemented scalarizer only can deal with if-converted form.

2.5 Alternate Approaches

As one can imagine, the taken route is not the only possible one. In fact we tried several different approaches to the problem of preserving the high-level expressions the GFortran frontend offers.

The very first approach was to keep the expressions in GENERIC form, thus not splitting the complex expressions apart. A patch for this was posted with the initial proposal to the gcc mailing-list.

The second approach was to put array expressions in GIMPLE form, but to allow array registers and also rewrite them into SSA form. This is what we also do with the approach outlined in this paper. The difference was that the expressions were operating on arrays, not scalar placeholders. This complicated matters unnecessarily if you consider mixed scalar/array expressions or matrix multiplication. A variety of new tree (sub-)codes were invented to accommodate all the needs. On the side of the C bindings you were not able to bind any temporary results to identifiers which also made the source look really ugly.

With the approach of using scalar placeholders for all computations and the generic contraction operator all of the previous problems seem to be addressed.

3 Scalarization

Lowering of array expressions to loop form has been implemented to ensure all necessary information for this task is available and to serve as an eventual fallback replacement for the optimized GIMPLE creation phase of the GRAPHITE framework.

The lowering process is organized as follows. A new loop nest is created for each final store to memory, thus we walk all statements looking for VLA <...> = tmp_n. The left-hand side of this statement specifies the iteration domain of the outer loop nest and the right-hand side SSA name connects the whole expression by means of the use-def chain which ends in leafs that consist of array loads. Similar handling is implemented for tmp_n = VLA_DELTA <...> if it has a true scalar result, for example in case of computing a scalar-product of two vectors.

As a preparation step we walk the use-def chain of the stores right-hand side SSA name and mark all defined SSA names as to be lowered if the walk ends in a generating function, which is any of VLA_*_EXPR. This preparation makes it easy to avoid putting any loop invariant code inside the lowered loop.

First we create the outer loop nest as specified by the array store. In particular we create the loop nest with loop headers copied and hoisted out of the complete loop nest. We fill the innermost basic block recursively by walking the use-def chain of the right-hand side creating code on-the-fly for all statements found to need lowering. This on-the-fly code generation includes adding loops for the contractions induced by the VLA_DELTA_EXPR operator.

While recursing through VLA_IDX_EXPR, VLA_RIDX_EXPR, and VLA_DELTA_EXPR we keep track of which index variables are mapped to which induction variable which is needed for proper lowering and allows the index variables to be used as loop dependent data input.

In Figure 3 you can see the lowered loop form of the matrix multiplication GIMPLE IL from Figure 2. In particular you can see how we build the array references from the first operands to VLA_VIEW_EXPR and fill in the variable length array parts as third and fourth operands of the ARRAY_REF operators. This stresses the fact that the frontends will be required to build proper VLA types for the machinery to work. You can also see the assumption that all array indices are zero-based, which is a restriction that can be easily lifted if the need arises.

```
void matmul(float *w, float *u, float *v, int n, int m) {
  float (*U)[n][m] = (float (*)[n][m])u;
  float (*V)[m][n] = (float (*)[m][n])v;
  float (*W)[n][n] = (float (*)[n][n])w;
  int i, j, k, l;
  float Ukj, Vil, VUij;
  Ukj = __builtin_array_idx (__builtin_array_select (U, m, 1, n, m), k, j);
  Vil = __builtin_array_idx (__builtin_array_select (V, n, 1, m, n), i, l);
  VUij = __builtin_array_delta (Vil * Ukj, m, k, l);
  __builtin_array_store (W, __builtin_array_ridx (VUij, i, n, j, n), n, 1, n, n);
}
```

Figure 1: Matrix multiplication using the C bindings

Figure 5 shows x86 assembly for the more complicated example in Figure 4 which implements the matrix-matrix-vector multiplication $V_{jk}U_{ki}W_j$. You can see how the outer loop of the matrix-matrix multiplication is fused with the inner matrix-vector multiplication loop.

The scalarizer currently misses data-dependence analysis to correctly lower array expressions with overlapping sources and destination.

4 C and C++ Language Interface

We created a simple interface to the new GIMPLE operations. The main constraints are minimal modifications to the frontends and exposing all features of the middle-end array framework.

The choice fell to support middle-end arrays by using GCC builtin functions that are lowered to the new GIMPLE operations during gimplification. As both C and C++ support VLA objects those serve as building blocks for communicating memory layout.

Let us start with an example. The piece of code in Figure 1 implements matrix multiplication of two matrices U and V which have sizes mxn and nxm. The result is to be stored to the nxn matrix W.

Here is an overview of the existing builtin functions:

1. `__builtin_array_select` maps to `VLA_VIEW_EXPR` and takes either a pointer to (variable length) array or an array argument. The following arguments are the extent, stride pairs as in the `VLA_VIEW_EXPR` specification. Note that the fastest varying dimension comes first as opposed to the C array notation where in `A[n][m]` m is the fastest varying index. The result of `__builtin_array_select` is of array type, so you cannot bind it to a C identifier.

2. `__builtin_array_store` is the counterpart to `__builtin_array_select` and necessary to not require array store handling in the frontends. It translates to an assignment with the left-hand side being a `VLA_VIEW_EXPR` as specified by the arguments to `__builtin_array_store`. In addition to the arguments required for `__builtin_array_select` a second argument is inserted that has to be of array type and serves as the right-hand side of the resulting assignment statement.

3. `__builtin_array_idx` directly maps to `VLA_IDX_EXPR`; its first argument needs to be of array type and the result is a scalar which you can bind to a C identifier.

4. `__builtin_array_ridx` maps to `VLA_RIDX_EXPR`, but for ease of gimplification the index arguments come in pairs of `index, extent` to be able to build a correct (variable length) array object that is returned. The first argument is a scalar. You cannot bind the result to a C identifier.

5. `__builtin_array_delta` maps to `VLA_DELTA_EXPR` and takes a scalar as first argument, the common extent as second argument and any indices that are to be summed over as following arguments. As this also has a scalar result, you can bind it to a C identifier as well.

Implementation-wise the builtins are type-generic vari-

```
int i, j, k, l;
float D.1629[0:n - 1][0:m - 1];
float D.1631[0:m - 1][0:n - 1];
float D.1633[0:n - 1][0:n - 1];
float x, y, s;

U_24 = (float[0:D.1588][0:D.1581] *) u_23(D);
V_46 = (float[0:D.1604][0:D.1597] *) v_45(D);
W_68 = (float[0:D.1620][0:D.1613] *) w_67(D);
# VUSE <SMT.10_89(D)>
D.1629_72 = VLA <m_1(D), 1, n_10(D), m_1(D)> (*U_24);
Ukj_75 = VLA_IDX <k_73(D), j_74(D)> (D.1629_72);
# VUSE <SMT.10_89(D)>
D.1631_79 = VLA <n_10(D), 1, m_1(D), n_10(D)> (*V_46);
Vil_82 = VLA_IDX <i_80(D), l_81(D)> (D.1631_79);
D.1632_83 = Vil_82 * Ukj_75;
VUij_84 = VLA_DELTA <m_1(D), k_73(D), l_81(D)> (D.1632_83);
D.1633_85 = VLA_RIDX <i_80(D), j_74(D)> (VUij_84);
# SMT.10_90 = VDEF <SMT.10_89(D)>
VLA <n_10(D), 1, n_10(D), D.1630_78> (*W_68) = D.1633_85;
return;
```

Figure 2: Matrix multiplication in GIMPLE SSA before lowering.

adic functions. To be able to handle different return types depending on arguments we invented a new function attribute, *covariant return*, which makes the frontend ignore mismatches in assignments and function calls. Unfortunately this makes the machinery somewhat fragile in that program errors easily turn into ICEs during gimplification or later.

In Figure 2 you can see the result of gimplifying the C example from Figure 1 in its state right before the lowering process. In particular you can see the array temporaries `D.1629`, `D.1631`, and `D.1633` which are in SSA form. You can also see that the extra extent parameters to the `__builtin_array_ridx` function are dropped in the `VLA_RIDX_EXPR`.

In its current form, the interface to C and C++ is not suitable for exposing high-level array operations to a wider user-base. Instead either the frontends need to be taught of the covariantness of the builtins properly, or new expressions need to be added to the languages. With C++ one can imagine wrapping the GCC builtins as in a properly templated and overloaded set of functions.

5 Future Work

If we settle on the proposed scheme as middle-end representation for high-level array operations, further steps need to be taken to fully leverage the potential of this proposal.

First a verifier for the expressions needs to be written. We expect some high-level interface for walking array expressions and indices as the result of this. This should serve as a start to teach the dependence analyzer about the array expressions as well.

To be able to efficiently lower array expressions, an SSA form is needed, which currently is not available without optimization. As this is likely to change, we didn't bother to implement a fallback solution for lowering at -O0.

The GFortran frontend needs to be changed to emit array expressions via the proposed scheme, not doing scalarization in the frontend.

As part of this the data dependence analysis required to decide if a temporary is needed in order to make loop iterations independent needs to be moved to the middle-end and implemented as a first lowering step before the scalarizer. This is needed to make the scalarization result correct in the case of overlapping of the source and destination arrays in an array expression.

At the point of lowering it needs to be decided if outputting inline code is profitable, or if a call to the GFor-

```
    U_24 = (float[0:D.1588][0:D.1581] *) u_23(D);
    V_46 = (float[0:D.1604][0:D.1597] *) v_45(D);
    W_68 = (float[0:D.1620][0:D.1613] *) w_67(D);
    ivtmp.19_88 = n_10(D);
    ivtmp.19_87 = n_10(D);
    if (ivtmp.19_88 == 0)
      goto <bb 3>;
  <bb 4>:
    if (ivtmp.19_87 == 0)
      goto <bb 3>;
  <bb 6>:
    # ivtmp.19_77 = PHI <0(4), ivtmp.19_76(8)>
  <bb 7>:
    # ivtmp.19_91 = PHI <0(6), ivtmp.19_92(10)>
  <bb 5>:
    D.1678_94 = m_1(D);
  <bb 9>:
    # ivtmp.19_95 = PHI <0(5), ivtmp.19_97(9)>
    # elttmp.20_96 = PHI <0.0(5), elttmp.20_101(9)>
    x_99 = (*U_24)[ivtmp.19_91]{lb: 0 sz: D.1628_71 * 4}[ivtmp.19_95];
    y_100 = (*V_46)[ivtmp.19_95]{lb: 0 sz: D.1630_78 * 4}[ivtmp.19_77];
    elttmp.20_98 = x_99 * y_100;
    elttmp.20_101 = elttmp.20_96 + elttmp.20_98;
    ivtmp.19_97 = ivtmp.19_95 + 1;
    if (D.1678_94 > ivtmp.19_97)
      goto <bb 9>;
  <bb 10>:
    elttmp.20_93 = elttmp.20_101;
    (*W_68)[ivtmp.19_77]{lb: 0 sz: D.1630_78 * 4}[ivtmp.19_91] = elttmp.20_93;
    ivtmp.19_92 = ivtmp.19_91 + 1;
    if (ivtmp.19_88 > ivtmp.19_92)
      goto <bb 7>;
  <bb 8>:
    ivtmp.19_76 = ivtmp.19_77 + 1;
    if (ivtmp.19_76 < ivtmp.19_87)
      goto <bb 6>;
  <bb 3>:
    return;
```

Figure 3: Matrix multiplication in GIMPLE SSA after lowering.

tran runtime is better. For this reason a pattern recognition pass should transform the middle-end array operations back to library calls. The canonical form of the contractions should make this easy.

In the same line a pass converting loops back to middle-end array operations may be worth looking at. Allen and Kennedy [1] go into great depth here in the course of exposing fine-grained parallelism.

The remaining middle-end array operations ideally will be lowered by the GRAPHITE framework that should output highly optimized scalar code. OpenMP-aware scalarization is also something that looks worth looking into.

The C++ valarray class looks like a obvious candidate for using the middle-end array operations infrastructure, but a proper type-safe language extension is needed before exposing this to the programmer.

6 Conclusion

A variety of approaches for making arrays a first-class citizen of the GCC middle-end have been explored and

```
void fancy (float *res, float *u, float *v, float *w, int n, int m) {
  float (*U)[n][m] = (float (*)[n][m])u;
  float (*V)[m][n] = (float (*)[m][n])v;
  float (*W)[n] = (float (*)[n])w;
  float (*R)[n] = (float (*)[n])res;
  int i, j, k;
  float Uki, Vjk, Wj, VUij, Ri;
  Uki = __builtin_array_idx (__builtin_array_select (U, m, 1, n, m), k, i);
  Vjk = __builtin_array_idx (__builtin_array_select (V, n, 1, m, n), j, k);
  Wj = __builtin_array_idx (__builtin_array_select (W, n, 1), j);
  VUij = __builtin_array_delta (Vjk * Uki, m, k, k);
  Ri = __builtin_array_delta (VUij * Wj, n, j, j);
  __builtin_array_store (R, __builtin_array_ridx (Ri, i, n), n, 1);
}
```

Figure 4: Matrix and vector multiplication using the C bindings.

```
        xorl    %r13d, %r13d
        xorl    %ecx, %ecx
        movl    %r13d, -36(%rsp)
        movss   -36(%rsp), %xmm3
.L5:
        movq    %r9, %rax
        movl    %r13d, -36(%rsp)
        xorl    %edx, %edx
        imulq   %rcx, %rax
        movss   -36(%rsp), %xmm2
        leaq    (%rbp,%rax,4), %rsi
.L4:
        movaps  %xmm3, %xmm1
        movq    %rsi, %r11
        xorl    %r10d, %r10d
.L3:
        movq    %r8, %rax
        imulq   %r10, %rax
        addq    $1, %r10
        addq    %rdx, %rax
        movss   (%rbx,%rax,4), %xmm0
        mulss   (%r11), %xmm0
        addq    $4, %r11
        cmpq    %r10, %r9
        addss   %xmm0, %xmm1
        ja      .L3
        mulss   (%rdi,%rdx,4), %xmm1
        addq    $1, %rdx
        cmpq    %rdx, %r8
        addss   %xmm1, %xmm2
        ja      .L4
        movss   %xmm2, (%r12,%rcx,4)
        addq    $1, %rcx
        cmpq    %rcx, %r8
        ja      .L5
```

Figure 5: X86 assembly for the matrix and vector multiplication.

the best has been selected and presented as a proposal for future integration into GCC. The advantages are that it will be possible to keep the high-level information that Fortran array expressions offer throughout the early optimization phases until the high-level loop-optimization phase.

It has been shown that the changes required to the compiler are small and that most of the optimization framework can be shared for optimizations on scalar and array expressions. With the implementation of a lowering pass we successfully showed that the proposal is complete and able to correctly preserve program semantics.

The array GIMPLE IL will also serve as a connection between the GFortran frontend and the GRAPHITE high-level loop optimization framework.

References

[1] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002. ISBN 1-55860-286-0.

[2] Daniel Berlin, David Edelsohn, and Sebastian Pop. High-level loop optimizations for gcc. In *GCC Developers' Summit*, June 2004.

[3] Diego Novillo. Design and implementation of tree ssa. In *GCC Developers' Summit*, June 2004.

[4] Sebastian Pop, Albert Cohen, Cedric Bastoul, Sylvain Girbal, Georges-Andre Silber, and Nicolas Vasilache. Graphite: Polyherdral analyses and optimizations for gcc. In *GCC Developers' Summit*, June 2006.