Optimizations Applied by the ExaStencils Code Generator

Stefan Kronawitter, Christian Lengauer

Faculty of Informatics and Mathematics, University of Passau
{kronast,lengauer}@fim.uni-passau.de

Technical Report, Number MIP-1502
Faculty of Informatics and Mathematics
University of Passau, Germany
January, 2015

Optimizations Applied by the ExaStencils Code Generator

Stefan Kronawitter and Christian Lengauer

Faculty of Informatics and Mathematics,University of Passau
{kronast,lengauer}@fim.uni-passau.de

Abstract. ExaStencils is a DFG-funded project with the charter to provide a dedicated infrastructure for the domain-specific engineering and optimization of multigrid stencil codes. The vision is that application scientists provide an abstract, mathematical formulation of the problem that should be solved by formulating, e.g., a partial differential equation or an energy functional which should be minimized. Another input to the infrastructure is the description of target platform which includes both hardware, e.g., the type of CPU, memory hierarchy or accelerators, and software, e.g., the compiler or the MPI implementation. With these inputs, the infrastructure will automatically generate target code which is especially tuned for the application and platform at hand. A central feature of such an infrastructure is its code generator. We describe the code transformations and optimizations performed by the ExaStencils code generator.

1 Introduction

Multigrid methods [7] are widely used in scientific applications, especially in physics and chemistry simulations to solve partial differential equations (PDEs). The basic idea of multigrid methods is two-fold. First, the high-frequency error of an approximation is smoothed with a few steps of an iterative method like Jacobi or Gauß-Seidel. Second, a smooth function can be approximated (and solved) on a coarser grid with much fewer discretization points. These are embodied in a recursive solver: a multigrid cycle scales the problem down by coarsening the grid, until it can be solved directly, and prolongates the result back to the finest grid.

The process of implementing multigrid methods is extremely time-consuming, especially if high performance is required. Even if a solver is already available, just moving it to a larger machine may require further optimizations to load it to capacity. Project ExaStencils [5] provides an infrastructure for programming numerical solvers in a fairly abstract domain-specific language (DSL), called ExaSlang, and employs an optimizing code generator to obtain automatically target code that is highly tuned to the specificities of the solver and execution platform [6]. The DSL unburdens the application scientist from having to deal with algorithmic or implementational aspects that would require knowledge which is not part of the application domain. The code generator uses domain-specific

knowledge made available by the application scientist and by the execution platform to create target code tuned to the specific scenario at hand. To port the application to a different machine, one simply invokes the generator again with an updated hardware specification.

After some background, given in Section 2, the set of transformations available to speed up the resulting code is presented in Section 3. Section 4 concludes and proposes future research.

2 Background

2.1 Polyhedron Model

The customary way of optimizing a loop nest is by manipulating the source code directly via indivial transformations like loop permutation, loop skewing, loop fusion, loop distribution, etc. This approach leaves the trade-off between different choices of transformation opaque and carries the danger that the more suitable transformations may be less obvious or more difficult to apply. The polyhedron model supports unbiased and automatic optimizing search across the space of possible loop transformations.

In the polyhedron model [3], a nest of d loops is represented by a d-dimensional integer polyhedron, the so-called iteration domain. Each point in the polyhedron represents a statement instance, i.e., one specific execution of a statement. Its coordinates $\boldsymbol{i}$, the so-called iteration vector, correspond to the values of the iteration variables of the surrounding loops at this instance.

In addition to the loop variables, the model contains structure parameters, which typically correspond to the problem size. For example, the grid size of a stencil code, which also describes the size of the iteration domain, is a structure parameter. It is unknown at compile time, since the size typically varies for different invocations, but constant at run time.

An integer polyhedron, or $\mathbb{Z}$-polyhedron, is the intersection of a (rational) polyhedron with $\mathbb{Z}^d$ or a lattice. It contains only integral points. This compact representation is independent of the number of loop iterations or the run time values of structure parameters. The consequence is one major benefit of the polyhedron model: a loop nest can be optimized without instantiating the variables in its loop bounds, or its structure parameters.

Optimizations are expressed as transformations of the iteration domain. In order to guarantee correctness, a transformation must respect all data dependences that involve a write access. That is, the ordering of all statement instances participating in such dependence must not change. Dependences can only be computed if all loop bounds and array subscripts are affine in the loop variables and structure parameters.

A code region which can be represented and optimized in the polyhedron model is called a static control part (SCoP). The data flow and data dependences of a SCoP can be computed statically, i.e., at compile time and both can be described by a finite set of affine expressions.

2.2 Integer Set Library

The Integer Set Library (isl) [8] is the most recent, and currently most popular, C library that supports the polyhedron model. It offers efficient operations on integer sets and relations bounded by affine inequalities, like set/map union, intersection, or projection.

The library is also able to deal with symbolic constants and existentially quantified variables. The latter are, e.g., used to model holes in a $\mathbb{Z}$-polyhedron. Consider the simple loop:

```
for (int i=0; i<n; i+=2)
S:  A[i] = 0;
```

A first attempt to model the iteration domain of statement `S` is with the constraint $0 \leq \mathtt{i} < \mathtt{n}$, but this covers also the odd values of `i`.

Actually, a description of the above iteration domain requires an extension of the model. The isl offers existentially quantified variables that have no effect on the shape of the polyhedron, but on the integral points inside it. In this case, a valid iteration domain requires the constraints:

$$(0 \leq \mathtt{i} < \mathtt{n}) \wedge (\exists\, \mathtt{a} : \mathtt{a} \in \mathbb{Z} : 2 \cdot \mathtt{a} = \mathtt{i})$$

In isl syntax, the iteration domain of `S` can be specified as follows:

```
[n] -> { S[i] : 0<=i<n and (exists a : 2*a=i) }
```

A transformation of the iteration domain can be represented by an isl map. Following the example above, a transformation that reduces the loop stride from 2 to 1 looks as follows:

```
[n] -> { S[i] -> S[o] : 2*o=i }
```

In addition to some basic methods for the inspection and manipulattion of sets and maps, isl offers also higher-level functions for important steps of the optimization, like a dependence analysis or the computation of an optimal schedule under the given constraints. Since all computations are performed with GMP,[1] the library can deal with arbitrarily sized integers.

2.3 Code Generator

All optimizations presented here have been implemented in our code generator. Its input is a problem specification in our domain specific language ExaSlang. ExaSlang has four layers of abstraction, ExaSlang 1–4, each addressing the needs of a different user group. The most abstract layer, ExaSlang 1, is directed at application scientists and engineers who have little or no experience in programming. The subsequent layers introduce, successively mathematical and computational structures necessary for an implementation. The most concrete layer, ExaSlang 4, has all the makings for an executable and platform-appropriate implementation [6].

[1] GNU Multiple Precision Arithmetic Library `https://gmplib.org/`

The optimizations described below address an even more concrete level: that of an intermediate representation (IR), which resembles C or C++ but also contains some of the (more abstract) features of ExaSlang 4. The latter help in the analysis of the code structure, but ExaSlang 4 alone is not expressive enough for some lower-level optimizations. For example, ExaSlang 4 provides a `loop over` statement for iterations across the computational domain of a given field, independent of its dimensionality. This construct is useful in the abstract specification of a smoother, but it does not allow modifications of the iteration ordering. Therefore, the generator ultimately emits code, in which `loop over` loops have been replaced by an appropriate nest of C `for`-loops.

3 Automatic Optimizations

Without optimizations, the automatically generated code is relatively naïve but also very inefficient. It pays to generate not the fastest possible code outright, but rather a correct version that is simple enough to enable the automatic application of all required optimizations subsequently. This approach has two basic advantages:

- A progression in small transformation steps simplifies the maintenance of the correctness of the code.
- A search for the most suitable sequence of optimizations and their parameters becomes possible.

The set of optimizations implemented in our code generator can be divided into polyhedral and traditional optimizations.

3.1 Polyhedral Optimizations

All polyhedral optimizations are currently encapsulated in a single strategy, which consists of the following steps. For each function defined in a multigrid cycle (smoother, restriction, etc.) the generator creates multiple methods, one for each grid level of the cycle, customized for the grid size at that level. The number of SCoPs is usually relatively large and the majority of the run time is spent in computing on the finer grids. The generator can be customized to emphasize optimization at the finer grid levels.

Depending on the configuration parameters and the optimization level, only a subset of the steps below may take hold.

Extract Models. In the first step, a polyhedral representation of all `loop over` statements is extracted. The abstract nature of the `loop over` iterator simplifies the extraction process. Also, the grid accesses are not yet linearized, which ensures their affinity even in case of a triangular grid.

Merge Models. The extractor generates one polyhedron per loop nest. If beneficial, polyhedra of adjacent models are merged.

Compute Dependences. Next, the data depencendes inside each polyhedron are computed. E.g., the flow-dependences are discovered by composing the write access relation with the inverse of the read access relation and intersecting the resulting map with the sequential ordering of the iteration domain. isl provides especially good support for this crucial operation.

Not only the flow-, anti-, and output-dependences, which must be respected because each involves at least one write access, but also the input-dependences are computed, since the polyhedral scheduler may want to use them later on to reduce the data reuse distance.

Eliminate Dead Code. The first optimizing transformation inside the polyhedron model is to detect and remove statement instances that have no effect on the code's output. Dead statement instances satisfy two requirements:

- They do not contain the, in the sequential execution, latest write to a memory location, which is going to be accessed after the loop.
- They are not source of any flow-dependence whose target is not dead.

Ignore Reduction Dependences. If there is a reduction in the loop nest, the data dependences specified by it effectively sequentialize the entire SCoP. But a reduction can be parallelized easily by ignoring its dependences. The code generated later will perform independent, sequential reductions in each thread and then a parallel tree reduction over the accumulators of all threads.

Search a Parallel Schedule. The central polyhedral optimization is the search for an optimal schedule with respect to the given objective function. Currently, the isl scheduler, which is based on the PLuTo algorithm [2], is used to optimize the schedule of the current SCoP. It attempts to increase data locality by minimizing the distance of input-dependences, while respecting all flow-, anti-, and output-dependences. For a sequence of stencil applications, this leads to a temporal blocking, which reduces the bandwidth requirements and, thus, may increase the performance. Alternative scheduling algorithms will be investigated in the future.

Tile Dimensions. Another popular optimization that curbs the memory bandwith requirement is tiling [4]. The generator uses a method provided by the isl to apply this transformation to an optimized SCoP. That is, tiling is performed inside the polyhedron model, rather than, as is commonly done, on the text of the source loop nest. This relieves the generator from the worry whether the number of iterations is or is not evenly divisible by the tile size. E.g., a transformation that tiles the second of a three-fold nested loop with a block size of 32 can be specified as follows:

```
{ S[z,y,x] -> S[t,z,y,x] : t = floord(y,32) }
```

where `floord` denotes an integer division rounded towards negative infinity.

Recreate the Abstract Syntax Tree. At this point, we have IR code for all statements in the loop nest plus the polyhedral description of the loop nest. The AST builder of the isl now generates an abstract syntax tree (AST) of C-like code that scans all points in the iteration domain as precsribed by the schedule discovered previously. Subsequently, the tree is transformed to IR code.

3.2 Traditional Optimizations

What remains are the non-polyhedral optimization techniques. Each is implemented in a separate strategy and, thus, can be disabled if desired. Additionally, there are some other small strategies on which the optimizations described below depend. These simply prepare the syntax tree, or collect other information required, and are omitted here.

Color Splitting. In case of a red-black Gauß-Seidel smoother, the first optimization is a partitioning of each field into two blocks, the first containing the red and the second the black points. This can be achieved by first doubling the range of the outermost dimension and then adding a summand that maps accesses to the black points to the newly created upper half, while the accesses to the red points remain as are. E.g., for two-dimensional fields, the added summand has the form

$$(d0 + d1)\%2 * N \tag{1}$$

where di is the i-th expression of the old index access vector and N is the old range. Now, the red and black points are separated but, to prevent that only every second array element is accessed, the size of the innermost dimension is halved and the expression for this dimension is modified to perform additionally a division by 2 at run time.

If a loop is known to update only data points of one color, as in the case of both parts of a single smoothing step, this information can be used to precompute the value of Eq. (1) statically.

Address Precalculation. Another, more general optimization of linearized array accesses inside a loop nest is to precompute a maximally sized part of the index expression outside the innermost loop. This is a standard compiler optimization implemented in production compilers [1], but it is not always applied since other transformations performed previously by our generator can stand in its way. Consequently, we implemented a more advanced version directly, as explained in the following example.

Consider the two field accesses `a[z][y][x]` and `a[z+1][y+1][x+1]`. Linearized versions may look as follows:

```
a[( z   *512 +  y   )*512 +  x]
a[((z+1)*512 + (y+1))*512 + (x+1)]
```

where `x` is the iteration variable of the innermost loop that surrounds both expressions. In a first step, each index expression is simplified and normalized in order to reduce the number of pointers that have to be precomputed later as much as possible. After simplification, the accesses become:

```
a[z*262144 + y*512 + x]
a[z*262144 + y*512 + x + 262657]
```

Now, both accesses can share the same base pointer outside the innermost loop:

```
a_p = &a[z*262144 + y*512];
for (...) {
  ... a_p[x] ...
  ... a_p[x + 262657] ...
}
```

This reduces the computational overhead inside the loop, which may lead to a performance increase unless the memory bandwidth is reached.

```
   .125*s[y/2       ][x/2       ]
 + .125*s[y/2       ][x/2 + x%2]
 + .125*s[y/2 + y%2][x/2       ]
 + .125*s[y/2 + y%2][x/2 + x%2]
```

Fig. 1: Example expression which is generated as part of a 2-dimensional prolongation code.

Unrolling Prolongation. Beside the smoother, another run time-intensive part of a multigrid application is the prolongation, i.e., the interpolation of a correction computed on a coarser grid to obtain a finer grid.

Take the example of the subexpression of an interpolation code for a 2-dimensional grid in Figure 1. The modulo expressions of its array subscripts evaluate to 1 in only one fourth of all iterations. Conversely, three fourth of the iterations perform semantically equivalent load operations, as well as unnecessary arithmetic instructions, in order to accumulate identical values multiple times.

An obvious optimization would be to unroll all loops involved and precompute the modulo expressions. This increases the code size exponentially in the number of dimensions, but it reduces the number of operations required to compute the actual memory locations, and it also results in some statically identical memory accesses. These duplicate loads can be collapsed by the following optimization.

Arithmetic Simplifications. Generated expressions like, e.g., depicted in Figure 1, benefit from further mathematical optimizations. In this example, it would be reasonable to apply the law of distributivity in order to prevent multiple floating-point multiplications. Another transformation would be to prevent a

multiple load of the same array element from memory or to convert a division by a constant to a multiplication by its inverse.

In order to provide such arithmetic transformations, we implemented a strategy that first extracts a normalized representation of each floating-point expression similar to the one used in the address precalculation strategy. During the extraction, multiple accesses to the same array element are factored out, if possible, which reduces the number of load instructions. This representation can be viewed as a large sum, and all constants are moved as closely as possible to the variables and array accesses. In a subsequent step, when recreating an abstract syntax tree, all summands with a common coefficient are collected and the law of distributivity is applied as above.

Vectorization. Most contemporary compilers provide automatic vectorizers, whose purpose is to generate vector instructions in order to use the processors SIMD units. But the compiler is not always able to emit efficient SIMD code, since the generated C++ code turns out to be quite complex after applying all previous transformations. Therefore, our generator vectorizes explicitly as many loops as possible. While doing so, it can take advantage of more information than is available to the target compiler later on. For example, the data flow analysis that is part of the polyhedral optimization phase tells it whether a loop can be vectorized safely. Also, the generator can choose an alignment of the memory allocations that avoids unaligned load and store operations.

Unrolling. The final optimization is the simplest: it unrolls the innermost loop of a loop nest. Unrolling reduces the overhead of evaluating the exit condition, as well as the number of jumps that must be performed. It also increases the number of instructions inside the body, which helps the target compiler and the processor to optimize the loop body by reordering its instructions.

In addition to inserting copies of the entire loop body as is, the generator has the option of interleaving the statements of all unrolled blocks, provided this does not violate data dependences. That is, if the original body consisted of two statements `A(i); B(i)`, a normally unrolled version with an unrolling factor of 2 would look like `A(i); B(i); A(i+1); B(i+1)`, while an interleaved one is `A(i); A(i+1); B(i); B(i+1)`. Generating the latter may be suitable for in-order architectures like, e.g., for BlueGene/Q.

4 Conclusions and Future Work

We have presented a number of automatic optimizations and transformations to bring stencil codes to highest performance. Their prototypical implementation in the ExaStencils code generator is concluded; further refinements are work in progress.

We plan to add optimizations to support other processor architectures such as BlueGene/Q, or even accelerators such as graphics processing units (GPUs).

While the former requires only moderate adaptions, e.g., specific transformations for various vector intrinsics, the latter also require new objective functions as optimization targets, like specific, explicit access patterns to achieve best performance on GPU memories.

5 Acknowledgements

This work is part of project ExaStencils in the DFG priority programme SPPEXA, grant no. LE 912/15-1.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers – Principles, Techniques and Tools. Addison-Wesley, 2nd edn. (2007), Section 6.4.3
2. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. In: Proc. ACM SIGPLAN 2008 Conf. on Programming Language Design and Implementation (PLDI). pp. 101–113. ACM (2008)
3. Feautrier, P., Lengauer, C.: Polyhedron model. In: Padua, D.A. (ed.) Encyclopedia of Parallel Computing, pp. 1581–1592. Springer (2011)
4. Irigoin, F., Triolet, R.: Supernode partitioning. In: Proc. 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 319–329. ACM (1988)
5. Lengauer, C., Apel, S., Bolten, M., Größlinger, A., Hannig, F., Köstler, H., Rüde, U., Teich, J., Grebhahn, A., Kronawitter, S., Kuckuk, S., Rittich, H., Schmitt, C.: ExaStencils: Advanced stencil-code engineering. In: Lopes, L., et al. (eds.) Euro-Par 2014: Parallel Processing Workshops, Part II. Lecture Notes in Computer Science, vol. 8806, pp. 553–564. Springer (2014)
6. Schmitt, C., Kuckuk, S., Hannig, F., Köstler, H., Teich, J.: ExaSlang: A domain-specific language for highly scalable multigrid solvers. In: Proc. 4th Int. Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC). pp. 42–51. ACM (Nov 2014)
7. Trottenberg, U., Osterlee, C.W., Schuller, A.: Multigrid. Academic Press (2000)
8. Verdoolaege, S.: *isl*: An integer set library for the polyhedral model. In: Proc. 3rd Int. Congress on Mathematical Software (ICMS). pp. 299–302. Springer (2010)