QUARC: An Array Programming Approach to High Performance Computing

Diptorup Deb, Robert J. Fowler, and Allan Porterfield

Department of Computer Science,
University of North Carolina at Chapel Hill, USA
diptorup@cs.unc.edu,
{rjf,akp}@renci.org
http://cs.unc.edu/

Abstract. We present QUARC, a framework for the optimized compilation of domain-specific extensions to C++. Driven by needs for programmer productivity and portable performance for lattice QCD, the framework focuses on stencil-like computations on arrays with an arbitrary number of dimensions. QUARC uses a template meta-programming front end to define a high-level array language. Unlike approaches that generate scalarized loop nests in the front end, the instantiation of QUARC templates retains high-level abstraction suitable for optimization at the object (array) level. The back end compiler (CLANG/LLVM) is extended to implement array transformations such as transposition, reshaping, and partitioning for parallelism and for memory locality prior to scalarization. We present the design and implementation.

Keywords: array-programming, domain-specific languages

1 Introduction

QUARC is an embedded C++14 domain-specific compilation framework for optimizing expressive high-level C++ template code. It addresses performance and productivity challenges in lattice quantum chromodynamics (LQCD) in exploration of new physics and new algorithms. QUARC provides a compact, high-level notation with support for aggressive optimization and performance portability across architectures and machine implementations. QUARC provides notation and mechanisms to solve partial differential equations over complex vector fields discretized on structured lattices . While the design choices for QUARC are driven by the needs of LQCD, we plan to generalize QUARC to other domains.

It is increasingly difficult to extract high levels of portable performance from today's high-end systems. A single node of a current-generation HPC system has features such as deeply nested cache hierarchies, multi-core parallelism, and short-vector SIMD units. Domain-specific and architecture-specific knowledge and labor are required to design efficient concrete data layouts and code. The resulting hand-optimized codes bear little resemblance to the original abstract concepts and they are difficult to debug and to maintain.

These issues spring from weaknesses in architecture-neutral abstract parallel programming frameworks. Libraries such as Intel TBB [7] and Kokkos [2] address some of the challenges. Increasingly, languages such as C/C++ are the choice for HPC programming, but they lack support for abstract arrays as *first-class* objects. Various libraries and domain specific-languages (DSLs) [20], [6] extend the expressiveness of C++ using template meta-programming techniques like expression templates (ETs). These suffer performance problems because the concrete implementation of the array expressions, particularly *scalarization* of loops, occurs at the time of template instantiation. This makes it difficult or impossible for the compiler to retain enough context to infer the programmer's intent or to infer properties such as lack of aliasing or side effects. Subsequent compiler-driven analysis and optimization are thwarted.

1.1 The LQCD Problem Domain

QCD is the theory of the *strong* force, one of the four fundamental forces in nature. LQCD discretizes space and time on a four-dimensional lattice. Each lattice site is represented by at least one 12-dimensional complex vector (*spinors*) and eight (3×3) $SU(3)$ matrices (*gauge links*). The lattice usually is represented using a nest of array and structure types using as much as 2 kilobytes per site. In production, the lattice sizes can be as large as $128^3 \times 256$.

LQCD programs typically involve stencil computations. Often, a stencils is applied once per iteration of an implicit solver. Every stencil computation involves multiple short matrix-vector products, like the one shown in Listing 1.2, that can touch up to 3K bytes per lattice site, leading to poor memory locality and a low computational intensity. These characteristics contraindicat stencil optimization strategies like *time-tiling* LQCD thus requires strategies for optimization that have proven hard to automate. Recent performance studies [9] have highlighted this increasing *software gap* by comparing hand optimized LQCD kernels to QDP++ [20], an existing C++ ETs-based LQCD DSL. Reported numbers show an $8\times$ performance difference on Intel's Xeon Phi accelerators and a $2.6\times$ gap on regular Intel Xeon processors.

1.2 The QUARC Approach

QUARC optimizes kernels like that shown in Listing 1.1. It supports dynamic arrays of arbitrary rank as first-class objects. The intermediate representation preserves array semantics, allowing QUARC to use existing analysis and optimization passes in LLVM, as well as to add domain-specific transformations. The main innovations are:

- It provides a loop-less declarative syntax that makes arrays first-class objects, and provides a framework for defining array operators.
- To define stencils, QUARC uses a *generalized shift* (`gshift`) operation providing a multi-dimensional view of the array accesses to the compiler. Enabling exact dependence and reuse-distances analyses, and avoiding issues

```
//======== Basic lattice QCD data types ----===//
typedef std::complex<double> c;
// 3-D complex vector
typedef std::array<c, 3> su3Vec;
// 3x3 complex matrix
typedef std::array<su3Vec, 3> su3Mat;
// Packed array of 8 SU3Matrices
typedef std::array<su3Mat, 8> wG;
// 12-D complex vector
typedef std::array<su3Vec, 4> wS;
// 4-D lattice of 12-D complex vectors
typedef quarc::mdarray<wS, 4, PERIODIC> wSLattice;
// 4-D lattice of packed 3x3 complex matrices
typedef quarc::mdarray<wG, 4, PERIODIC> wGLattice;
int main () {
    wSLattice s_in(16,16,16,16), s_out(16,16,16,16);
    wGLattice g(16,16,16,16);
    // ... intializations

    //========= An abridged QCD stencil ----===//
    // operator* : su3_mult_op mkernel (Listing 1.2)
    // operator+ : complex vector addition
    // gshift    : described in Section 2.2
    // adj()     : complex adjunct
    s_out = g.get<0>() * s_in.gshift<1,0,0,0>()
          + g.get<1>() * s_in.gshift<0,1,0,0>()
          + g.get<2>() * s_in.gshift<0,0,1,0>()
          + g.get<3>() * s_in.gshift<0,0,0,1>()
          + adj(g.get<4>()) * s_in.gshift<-1,0,0,0>()
          + adj(g.get<5>()) * s_in.gshift<0,-1,0,0>()
          + adj(g.get<6>()) * s_in.gshift<0,0,-1,0>()
          + adj(g.get<7>()) * s_in.gshift<0,0,0,-1>();
    return 0;
}
```

Listing 1.1. A lattice QCD stencil written in QUARC syntax

such as *delinearization* [13]. The `gshift` operator cleanly separates stencil-related accesses from those occurring inside the pointwise operations.

- QUARC defers loop generation (*late scalarization*) of array expressions to the compiler. Late scalarization facilitates optimizations such as common subexpression elimination or expression fusion to array expressions. This opens the possibility of generating domain- and architecture-specific loop constructs after incorporating other optimizations.
- It provides uniform support for data transformations including tiling for shared-memory parallelism, partitioning for distributed parallelism, improving memory locality, and aligning data for vectorization. QUARC includes classical array transformations like *reshape, transpose* and *catenate* [14], [15].

```
template<typename T1, typename T2>
auto su3_mult_op(T1 m,T2 v){
  T2 r;
  for(int i=0;i<3;i++) {
    r[i][0]=0.0;r[i][1]=0.0;
    for(int j=0;j<3;j++) {
      r[i][0] += m[i][j][0] * v[j][0];
      r[i][0] -= m[i][j][1] * v[j][1];
      r[i][1] += m[i][j][0] * v[j][1];
      r[i][1] += m[i][j][1] * v[j][0];
    }
  }
  return r;
}
```

Listing 1.2. `Mkernel` defining a pointwise SU3 matrix-vector product

These enable the modification of array properties such as *rank* (number of dimensions) and *shape* (extent of each dimension). Combining such transformations with dependence- and reuse-distance analyses makes it possible to derive data layout transformations such as *structure of arrays* (SoA) to *arrays of structure of arrays* (ASoA) required for vectorizing LQCD kernels on short-vector SIMD machines.

2 An Array Programming Approach to Parallelism

Compilers for data-parallel programming languages like HPF [16] have focused on loop-centric transformations that alter the execution schedule of loop iterations to remove true dependence, improve cache-locality, and introduce parallelism. Without a data-centric view of the array expressions data-layout transformations become very challenging.

As a domain-specific compilation framework, QUARC can exploit inherent guarantees that allow us to take a radically different approach. QUARC statements are guaranteed to be data-independent, with all arrays having the same rank and shape (refer Sections 3.3, 3.4). This allows QUARC to be fully data-centric and to make loop generation a final step in the optimization process. In addition to traditional *loop-tiling* optimizations, QUARC can do data-layout transformations to support short-vector SIMD units.

2.1 QUARC Array Transformations

Array operations have been defined formally [14], [15] for APL [8] and similar array-programming frameworks. Such operations can alter the structural properties of arrays, and offer the necessary semantics for defining data-reordering within arrays. Mainstream procedural languages, like C/C++, have offered very limited support for such array operations.

Notations for defining array properties. We use Λ to denote an $\boldsymbol{n}$-dimensional LQCD lattice defined using QUARC arrays. Upper-case Roman characters used in a postfix notation denote array properties. Lower-case Greek letters denote array operations. Operations are written using C-like function call notation.

The dimensionality of the arrays is denoted by N and the extent of each dimension by B_i. The shape vector, made up of the dimensional extents, is represented as S and an index coefficient vector holding the cumulative sizes for each dimension is referred to as I_c. We initialize S and I_c as

$$S_{initial} = \{B_i | N < i <= 0\}, \tag{1}$$

$$I_{c\_initial} = \{ \prod_{i=N-2}^{0} B_i, \prod_{i=N-3}^{0} B_i, \ \ldots \ , 1\}. \tag{2}$$

A set of abstract array operations are used to model the data transformations. Selecting an element from a list is done using the (ι) operator. Reshaping array dimensions is done via the (ρ) operator. Reshaping is defined as

$$S_{new} = (\rho(\Lambda S, R_f)), \tag{3}$$

where R_f denotes a vector containing the reshape factors for all of the dimensions. Reshaping introduces padding only if $\iota R_f i$, for a given dimension does not divide the original B_i evenly. The new extents are

$$B_{i\_new} = \frac{\iota(\Lambda S, i)}{\iota(R_f, i)} = \begin{cases} B_i, & \text{if } \iota(R_f, i) == 1 \\ \{\iota(R_f, i), \lceil \frac{\iota(\Lambda S, i)}{\iota(R_f, i)} \rceil\} & \text{otherwise} \end{cases}$$
$$, \text{where } 0 < \iota(R_f, i) < B_i. \tag{4}$$

Transpose (Φ) generalizes two-dimensional matrix transpose to transpose an array about any diagonal, and catenation (κ) is used to merge or to linearize two adjacent dimensions into one. For both operatoions the required dimensions are specified as a two-tuple argument.

QUARC representation of array expressions. We introduce additional terminology for explaining the QUARC program structure. A QUARC kernel (Q_k) is a single array statement inside a QUARC program. Conceptually, it is an abstract countable loop over all values of the index set of the arrays referenced in the statement. Mini-kernel (`mkernel`) is a pointwise array operator or second-order array function. The iteration domain of a Q_k is enoted as AI_s. It the set of all the execution instances that need to be completed when processing the Q_k. In QUARC, the AI_s geometrically represents an $\boldsymbol{n}$-orthotope or *hyperrectangle*, with origin as the lower bound and upper bounds equal to the corresponding B_i. Each point in AI_s is termed an iteration point and is identified by an $\boldsymbol{n}$-tuple coordinate. Finally, the index space or the data domian is represented as D_s. It is the set of all array elements accessed by the Q_k. Although arrays are stored

in a one-dimensional linearized address space, D_s is an $\boldsymbol{n}$-dimensional space. We only consider monolithic addressing (Section 3.2) of QUARC arrays, therefore AI_s and D_s are always equivalent for every Q_k.

2.2 An Array-Transformation Mechanism

The present array-transformations in QUARC are driven by a reuse-distance based algorithm to derive SIMD friendly data-layouts for LQCD stencils. Reuse-distance is defined as the measure of non-unique data referenced between two successive uses of a given array reference. Various well-known canonical cache-blocking optimizations are based on reuse-distance, such as those provided by Wolf and Lam [24]. Henretty *et al.* [5] introduced a novel data-layout transformation for short-vector SIMD also using reuse-distance analysis to identify SIMD vector-stream alignment conflicts (SACs). Their algorithm uses the SAC metric to define $\Phi\rho$ transforms on the innermost dimension of multi-dimensional arrays to enhance vectorizability.

The QUARC array-transformation algorithm expands on Henretty *et al.*'s algorithm. We incorporate κ along with $\Phi\rho$ and apply the transformation to any dimension of the array. The technique derives the *gather-scatter* data-layout transformation and the required data mappings. Extending the transformation to outer dimensions can lead to an exhaustive search for the best layout. To reduce the search space, we use a LQCD-specific transformation. Most LQCD configurations use three equal-sized spatial dimensions and a time dimension twice the extent of the others. Thus, QUARC usually can ensure that the longest dimension is always innermost before starting layout transformations.

Step 1: Analyze outer accesses. The first step evaluates the accesses at the outermost nesting level and identifies SACs. We then apply $\kappa\Phi\rho$ to the innermost dimension, and proceed outwards until SACs are removed. Algorithm 1 provides an outline of the QUARC array-transformation algorithm using the kernel in Listing 1.1 as the input. The transformations are applied to both S and I_c. The final state of S provides the new shape with an innermost vector dimension, and the final state of the I_c gives the mapping to the old index space.

Step 2: Analyze mkernels. Along with analysis on the outer array accesses the `mkernels` are also analyzed for vectorizibility. For example, the `mkernel` in Listing 1.2 has no vectorizable loops, but has interleaved data accesses.

Step 3: Finalizing data-layout. In the final step the analyses from the earlier steps are combined to derive the data-layout for the complete Q_k. For Listings 1.1 and 1.2 after creating a vector dimension from the outermost dimensions the inner nested dimensions are permuted out.

2.3 Parallel Code Generation

The output of the array-transformation phase of the QUARC analysis is a mapping from D_s to the new data space, D'_s. These spaces can be of different dimensionality, as the transforms can change the rank of the arrays. D'_s gets broken into multiple split index sets to handle different boundary regions, and each set

Algorithm 1: QUARC array transformation outline

Input : $\Lambda\sigma$, where $\Lambda N = 4$
Input : Dimensional Reuse Distance Vector
Input : Linearized Reuse Distance Vector
Input : Vector Length (V_l)
Output: Index set transformation map

permute dimensions to ensure $B_0 \geq B_1 \geq B_2 \geq B_3$
if *More than one dimension has a SAC* **then**
  abort ; // Λ too small to benefit from layout transforms
else
  if $B_0 > V_l$ **then**
    $R_f =< 1, 1, 1, V_l >$;
    $S_1 = \rho(\Lambda S, R_f)$; // $< B_3, B_2, B_1, V_l, \frac{B_0}{V_l} >$
    $S_2 = \Phi(S_1, (1, 0))$; // $< B_3, B_2, B_1, \frac{B_0}{V_l}, V_l >$
  else
    factorize V_l to $(\frac{V_l}{2}, 2)$;
    $R_f =< 1, 1, \frac{V_l}{2}, 2 >$;
    $S_1 = \rho(\Lambda S, R_f)$; // $< B_3, B_2, \frac{V_l}{2}, \frac{B_1}{\frac{V_l}{2}}, 2, \frac{B_0}{2} >$
    $S_2 = \Phi(S_1, (1, 0))$; // $< B_3, B_2, \frac{B_1}{\frac{V_l}{2}}, \frac{V_l}{2}, \frac{B_0}{2}, 2 >$
    $S_3 = \Phi(S_2, (2, 1))$; // $< B_3, B_2, \frac{B_1}{\frac{V_l}{2}}, \frac{B_0}{2}, 2, \frac{V_l}{2} >$
    $S_4 = \kappa(S_3, 1, 0))$; // $< B_3, B_2, \frac{B_1}{\frac{V_l}{2}}, \frac{B_0}{2}, V_l >$
  end
end
create a mapping function from D_s *to* D'_s

is materialized into actual loop nests. For the set operations and the loop generation, we use the integer set operations and a polyhedral code generator from the Integer Set Library (*isl*) [22]. (See Section 5.3.) After transforming the arrays, we annotate different dimensions with the parallelization strategy to be used. Typically, the innermost dimension is designated as a vector dimension, and the outermost is parallelized using threads or MPI. We propagate this metadata into the *isl*-generated loops using existing LLVM infrastructure.

3 QUARC Language Design

QUARC uses C++14 template meta-programming to implement a DSL interface that generates annotations recognized by the compiler. Figure 1 presents an abridged BNF grammar for the QUARC DSL. By definition, QUARC programs are valid C++14 code compilable by any C++14 compiler. The language semantics are close to the C++ ETs idiom [21]. The ETs idiom uses overloaded operators and proxy expression objects to build array expressions without intermediate containers. ETs have been used widely, in various scientific computing DSLs and BLAS libraries DSLs [20], [6], [17] to embed array semantics in C++.

```
⟨quarc_kernel⟩       ::= ⟨mdarray_terminal⟩ = {
                         | ⟨bin_expr⟩ | ⟨gshift_expr⟩ | ⟨unary_expr⟩ |
                           ⟨terminal_expr⟩ } {...}
⟨gshift_expr⟩        ::= ⟨mdarray_terminal⟩ , ⟨integers⟩{...}
⟨bin_expr⟩           ::= { ⟨terminal_expr⟩ | ⟨binary_expr⟩ | ⟨shift_expr⟩ |
                           ⟨unary_expr⟩ } {2}, ⟨bin_op⟩
⟨unary_expr⟩         ::= ⟨terminal_expr⟩ | ⟨bin_expr⟩ |
                         | ⟨shift_expr⟩ | ⟨unary_expr⟩, ⟨unary_op⟩
⟨terminal_expr⟩      ::= ⟨mdarray_terminal⟩ | ⟨scalar_terminal⟩
⟨bin_op⟩             ::= ⟨is_arithmetic⟩{2}, ⟨bin_mkernel⟩
⟨unary_op⟩           ::= ⟨is_arithmetic⟩, ⟨unary_mkernel⟩
⟨bin_mkernel⟩        ::= ⟨is_arithmetic⟩ ⟨id⟩
                         | ( ⟨is_arithmetic⟩ ⟨id⟩, ⟨is_arithmetic⟩ ⟨id⟩ )
⟨unary_mkernel⟩      ::= ⟨is_arithmetic⟩ ⟨id⟩ ( ⟨is_arithmetic⟩ ⟨id⟩ )
⟨mdarray_terminal⟩   ::= ⟨mdarray⟩
⟨scalar_terminal⟩    ::= ⟨is_arithmetic⟩
⟨mdarray⟩            ::= ⟨is_arithmetic, rank, boundary_fn, shape⟩
```

Fig. 1. QUARC array syntax pseudo-BNF

QUARC differs from conventional ETs. First, with the aforementioned system of annotations, we embed extended type information in the syntax to extend the type system abstractly and to make QUARC arrays first-class objects. The annotations are transparent to the end user and need no manual intervention while programming in the QUARC DSL. Second, the late-scalarization technique pushes loop generation from the template-instantiation phase into the compiler back end. These design choices enable the QUARC optimizer (QOPT) to derive non-trivial low-level optimizations. In the next section, we describe the QUARC DSL syntax and API semantics.

3.1 QUARC Arrays

QUARC's `mdarray` data type is an abstract composite type that is represented using a four-tuple : <`type, rank, boundary-function, shape`>. The `type` specifies the C++ data type of the array elements. The current implementation limits the types to those matching the C++14 type trait `is_arithmetic`. The `rank` property is the number of dimensions of the array. `Boundary-function` is a user-definable index function to handle boundary conditions, and `shape` defines the extent of each dimension. Of these properties, `element-type`, `rank` and `boundary-function` are compile-time constants, specified as template arguments. The `shape` property is specified using C++14's variadic template feature. A combination of static and run-time checks is used for full type inference.

3.2 Array Addressing Modes

QUARC provides two addressing modes for the `mdarray` instances. Monolithic addressing operates on entire arrays and is used in array expressions. Elemental

addressing is similar to C++ subscript operation. In this paper, we focus on the monolithic addressing mode. Monolithic addressing of the eschews explicit subscripts. Allowing only an $\boldsymbol{n}$-tuple address offsets or "shifts", where $\boldsymbol{n}$ is the rank of the array. By default the shifts are all generated as "0s". Non-zero shifts are specified using the `gshift` operator.

There are two significant benefits of this approach. By design, the programmer uses whole-array subscripts, and the address linearization happens after performing optimizations. Thus, we do not have to deal with the *delinearization problem* of recovering a multi-dimensional view of the array accesses [13]. All references except the boundary values use the same index function, differing only in the constant term. Such references are termed *uniformly generated references* [24]. Moreover, every subscript implicitly describes an affine function, with a single index variable (SIV). This practice makes it possible to compute exact dependence distance vectors as well as reuse-distances. Together, these features support optimizations that otherwise have been hard to implement in C++ ETs-based array programs.

3.3 Array Operators

QUARC array operators are higher-order functions that take a callback function (`mkernel`) in a template argument to do the actual elemental array operations. The design cleanly differentiates the stencil operations, defined using `gshifts`, from the `mkernels`. Allowing us to derive data-layouts after analyzing both operations. The `mkernel`s are required to be "pure" or "side-effect free", such that every QUARC expression induces a completely statically determinable control flow. The language semantics allow `mkernel`s to operate on different types. For example, as shown in Listing 1.1 in QUARC it is possible to define arrays of matrices and arrays of vectors, and then to create a multiplication operator to operate on them, producing another array of vectors.

3.4 Array Statements

QUARC array-statement semantics are similar to those of other high-level languages supporting array objects, such as Fortran 90 and HPF. The right-hand side (RHS) is evaluated completely without side effects and only then is the result written into the left-hand side (LHS). We disallow the use of the same array on both sides. In the future, we intend to remove this restriction by using data-dependence analysis to identify the intersecting hyperplane in the index space between the left- and right-hand sides, and to introduce a temporary minimal-size variable. All arrays in an expression are assumed to be non-aliasing. We enforce the restriction that they have the same rank and shape.

4 The QOPT Architecture

QOPT, QUARC's underlying optimization framework, is built on top of the LLVM compiler infrastructure. It uses *isl* for set operations and loop genera-

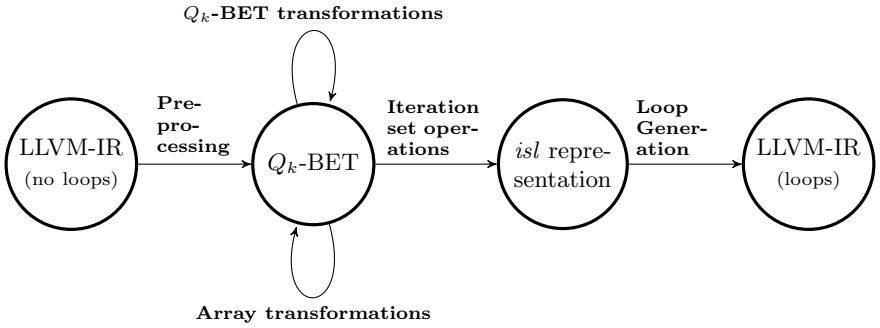

Fig. 2. The QOPT architecture

tion. The optimization workflow is a five-step process, as depicted in Figure 2. First, a preprocessing step detects all Q_ks in a procedure. After, preprocessing an abstract binary expression tree (Q_k-BET) representation is generated for each Q_k. The possibility of early transformations is explored using the Q_k-BET representation, and involves potentially combining the trees of multiple Q_ks.

After early transformations on the Q_k-BET, QOPT evaluates the applicability of array transformations for memory locality and SIMD-friendly data-layouts. The array transformations may lead to data layout changes. If so, an abstract map from the old to the new layout is generated to build the required *gather-scatter* code during the code-generation phase.

Following the array transformations, QOPT converts the Q_k-BETs into multiple iteration sets using *isl*. The iteration sets separate the iterations that require boundary-value computations from iterations that process only inner (non-boundary) elements of the arrays. The iteration sets and the corresponding loop-bounds are determined by the shape of the arrays and by the shifts specified in the array accesses. In the final step, QOPT scalarizes the iteration sets into actual loops in the LLVM IR language.

5 Array Expressions to Optimized Code

5.1 Preprocessing

The preprocessor recognizes QUARC annotations and applies program transformations that reduce code complexity while maintaining the semantic structure for further analysis and transformation passes. For example, it inlines all functions generated by QUARC templates other than the `mkernel` calls. This step significantly prunes the call graph, yet retains the separation of high-order stencil operators and the pointwise `mkernel` operations.

The preprocessor also annotates the LLVM IR to enable the construction of the Q_k-BETs. Listing 1.3 shows an abridged state of the IR after preprocessing a binary array expression that has a single `gshift` access. Each `quarcc_build_*_expr` call represents the creation of the proxy expression objects. The

```
/* Original code : a1 = a2.gshift<1,0>() + a2; */
  %1 = call __quarcc_build_gshift_expr__(%a2)
  %2 = call __quarcc_build_bin_expr__(%1, %a2)
  call __quarcc_kernel_dispatch__(%a1, %2)
```

Listing 1.3. State of IR after preprocessing

`quarcc_kernel_dispatch` is the call to the actual Q_k function. In the code-generation phase, the proxy objects are removed completely, while the Q_k call is transformed into inline loop nests.

5.2 Q_k Expression Tree Generation and Early Optimizations

The Q_k-BET is the intermediate representation that QOPT uses for all analysis and transformations. Generation of the Q_k-BET is also a two-step process. In the first step, QOPT analyzes the `quarcc_kernel_dispatch` function to build an abstract expression tree that does not contain the actual array references used in a particular instance of the Q_k. The `quarcc_kernel_dispatch` takes two parameters: the LHS subexpression that is always a single `mdarray` reference, and the RHS subexpression. To build the tree, QOPT recursively uses *def-use* chain analysis of the RHS subexpression parameter. Specifically, it looks for two specially annotated functions: `mkernel` and `access`. These two are the nodes of the tree, with the `access`es forming the leaves and the `mkernel`s forming the internal nodes. The access function, as described in Section 3.2, contains only the shift values. These are then extracted using LLVM's ScalarEvolution analysis.

After building the expression tree, QOPT *materializes* the actual Q_k-BET by building a second tree, a data structure that we call the "expression-builder-tree". The expression-builder-tree is constructed using successive *def-use* analyses of the arguments passed to the `quarcc_build_*_expr` calls, immediately preceding the `quarcc_kernel_dispatch`. The leaves of the expression-builder-tree store the actual array references to be used in the Q_k. QOPT builds a complete binary expression tree for every Q_k by matching these two trees.

Q_k-BET Merging QOPT looks for opportunities to *fuse* adjacent Q_ks to enhance memory locality in the body of a potentially fused loop nest. It limits fusion to adjacent Q_ks that share at least one array reference. Because all arrays in a Q_k have the same shape, the fused loop iteration space is the same as the original abstract iteration space of each Q_k. This strategy was used to simplify code generation in the current implementation.

We currently restrict fusion to kernels that are completely data-independent. QOPT does not try to fuse two kernels where the LHS of one kernel is accessed using a non-zero shift on the RHS of the other kernel. The fusion of the Q_ks is done using the Q_k-BET representation, thus merging the expression trees into a single tree. Scalarization then builds a single loop body for the fused tree.

```
/* Original a1 = a2.gshift<1,0>() + a2; */
  // No boundary operations needed
  for (int c0 = 0; c0 < D0 - 1; c0 += 1)
    for (int c1 = 0; c1 < D1; c1 += 1)
      a1[c0][c1] = a2[c0+1][c1] + a2[c0][c1];
  // Requires boundary function(PERIODIC) call
  if (D0 >= 1)
    for (int c1 = 0; c1 < D1; c1 += 1)
      a1[D0][c1] = a2[PERIODIC(D0+1)][c1] + a2[c0][c1];
```

Listing 1.4. Code generated after late scalarization

5.3 Late Scalarization

Late scalarization is the phase in which QOPT concretizes the abstract Q_k-BET representation. To help explain the process, we formally define an **out-of-bound set** (OB_s) as the the subset of D_s for which a shifted array access in the Q_k leads to an out-of-bound access. Every dimension can have two OB_s, each corresponding to the lower and upper bounds of that dimension. Thus, there can be a maximum of $\boldsymbol{2n}$ OB_ss for a given Q_k. Geometrically, the out-of-bound sets represent faces or boundaries of the $\boldsymbol{n}$-orthotope.

To compute the OB_s for a given Q_k, QOPT first calculates the maximal positive and negative shifts for every dimension. The OB_s for a given dimension, i, are computed by subtracting the maximal negative shift from the lower bound, then subtracting the positive shifts from the B_i. Thus, no OB_s are generated if the maximal shift in a given direction is 0.

Index-Set Splitting Once QOPT generates the OB_s it proceeds to split D_s into disjoint subsets to separate all of the iterations for which a boundary function call is required. To build these split sets, QOPT successively finds all possible combinations of adjacent facets of the $\boldsymbol{n}$-orthotope. For every combination, the OB_s corresponding to each facet in the combination is intersected with D_s, and all other OB_s not in that particular combination are subtracted from D_s.

In the worst case, where each dimension has a non-zero shift in both directions, the process is equivalent to computing each lower dimensional *facet* or $\boldsymbol{k}$-orthotope of the original $\boldsymbol{n}$-orthotope, where $\boldsymbol{k} = (0..n]$. Since each $\boldsymbol{k}$-orthotope in turn has $\boldsymbol{2k}$ facets, the total number of split sets generated is S, where

$$S = \sum_{k=0}^{n-1} {}_nC_k 2^{n-k} + 1. \tag{5}$$

It can be shown that S equals 3^n. This is because each facet must have its center as a valid I_p, and the set of all the centers is the set of points each of whose coordinates can have only three possible values $\{0, \lfloor B_i/2 \rfloor, B_i\}$. Thus, the total number of centers, and by corollary the number of hyperrectanges, has to be 3^n. Hence, in the worst case the number of split sets is exponential in the number of dimensions. Listing 1.4 shows the generated loop nests for the

example introduced in Lisiting 1.3. We show the equivalent C++ code for what QOPT generates in the LLVM IR language.

6 Related Work

C++ ETs optimizations. Various approaches to array semantics in C++ using the C++ ETs idiom have been explored. Iglberger *et al.* [6] and Härdtlein *et al.* [4] presented techniques to improve the sequential performance of ETs. The Boost.SIMD package [3] provides an abstract interface built using ETs that automate generation of SIMD intrinsics to enable vectorized code generation. These designs do not have a compiler-based component. Winter *et al.* [23] designed a just-in-time compilation framework for ETs to optimize GPU kernels. None of these approaches addresses the optimization of multiple statement. There are no provisions for data layout transformations or for cache-blocking.

DSL compilation strategies. Compiler-driven techniques with goals similar to ours have also been attempted. The ROSE [26] compiler framework was originally designed as a preprocessor generator that could do automatic property discovery and optimizations from C++ ETs. The telescoping languages [11] design was also an influential proposal addressing many of these issues.

Stencil compilers. Special-purpose stencil compilers have been the target of many research efforts. The Rice dHPF compiler allowed compilation of stencil codes for distributed memory systems [16]. More recently, Datta *et al.* [1], Kamil *et al.* [10] and Tang *et al.* [19] offered solutions for shared memory multi-core platforms. Henretty *et al.* [5] built a stencil compiler incorporating data-layout transformations for short-vector SIMD machines.

Compiler driven data-layout optimizations. Automating data-layouts selection for vectorization has been addressed by number of recent works. Majeti *et al.* [12] offered an automated solution for SoA to AoS transformations targetting heterogeneous platforms. Sung *et al.* [18] provided a transformation technique for structured grid applications on GPUs. Xu and Gregg [25] designed a pragma based semi-automatic technique that also transforms SoA to AoS.

7 Status and Work in Progress

Currently, QUARC can process simple examples end-to-end to generate single-threaded X86_64 executables. We currently support multi-dimensional arrays, but do not yet support arrays nested at each lattice site to support the SU(3) algebra used in LQCD. Ongoing work is addressing the extension of the semantics to nested arrays with the objective of generating optimized code for non-trivial LQCD applications. This work will relax the current type restriction (Section 3.1) on the `mdarray`s.

We are in the process of integrating the late scalarization module with LLVM's parallel code generation framework to support OpenMP outlining and vector code generation. We are also extending the array transformation framework to support data partitioning at the level of MPI nodes.

Acknowledgement. This work was supported in part by the DOE Office of Science SciDAC program on grants DE-FG02-11ER26050/DE-SC0006925 and DE-SC0008706.

References

1. Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., Yelick, K.: Stencil Computation Optimization and Auto-tuning on State-of-the-art Multicore Architectures. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing. pp. 4:1–4:12. SC '08, IEEE Press, Piscataway, NJ, USA (2008), http://dl.acm.org/citation.cfm?id=1413370.1413375
2. Edwards, H.C., Trott, C.R.: Kokkos: Enabling performance portability across manycore architectures. In: Proceedings of the 2013 Extreme Scaling Workshop (Xsw 2013). pp. 18–24. XSW '13, IEEE Computer Society, Washington, DC, USA (2013), http://dx.doi.org/10.1109/XSW.2013.7
3. Estérie, P., Gaunard, M., Falcou, J., Lapresté, J.T., Rozoy, B.: Boost.SIMD: Generic Programming for Portable SIMDization. In: Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques. pp. 431–432. PACT '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2370816.2370881
4. Härdtlein, J and Pflaum, C and Linke, A and Wolters, C H: Advanced expression templates programming. Comput. Visual Sci. 13(2), 59–68 (2009), http://dx.doi.org/10.1007/s00791-009-0128-2
5. Henretty, T., Veras, R., Franchetti, F., Pouchet, L.N., Ramanujam, J., Sadayappan, P.: A stencil compiler for short-vector SIMD architectures. Proceedings of the 27th international ACM conference on International conference on supercomputing - ICS '13 p. 13 (2013), http://dl.acm.org/citation.cfm?doid=2464996.2467268
6. Iglberger, K., Hager, G., Treibig, J., Rüde, U.: Expression Templates Revisited: A Performance Analysis of Current Methodologies. SIAM J. Sci. Comput. 34(2), C42–C69 (2012), http://dx.doi.org/10.1137/110830125
7. Intel Corporation: Intel Threading Building Blocks (2016)
8. Iverson, K.E.: Notation As a Tool of Thought. Commun. ACM 23(8), 444–465 (aug 1980), http://doi.acm.org/10.1145/358896.358899
9. Joo, B., Smelyanskiy, M., Kalamkar, D.D., Vaidyanathan, K.: Wilson Dslash Kernel From Lattice QCD Optimization (Jul 2015), http://www.osti.gov/scitech/servlets/purl/1223094
10. Kamil, S., Husbands, P., Oliker, L., Shalf, J., Yelick, K.: Impact of Modern Memory Subsystems on Cache Optimizations for Stencil Computations. In: Proceedings of the 2005 Workshop on Memory System Performance. pp. 36–43. MSP '05, ACM, New York, NY, USA (2005), http://doi.acm.org/10.1145/1111583.1111589
11. Kennedy, K., Broom, B., Chauhan, A., Fowler, R.J., Garvin, J., Koelbel, C., Mccosh, C., Mellor-Crummey, J.: Telescoping languages: A system for automatic generation of domain languages. Proceedings of the IEEE 93(2), 387–408 (2005)

12. Majeti, D., Barik, R., Zhao, J., Grossman, M., Sarkar, V.: Compiler-Driven Data Layout Transformation for Heterogeneous Platforms. In: Euro-Par 2013: Parallel Processing Workshops, pp. 188–197. Springer Science + Business Media (2014), http://dx.doi.org/10.1007/978-3-642-54420-0_19
13. Maslov, V.: Delinearization: an Efficient Way to Break Multiloop Dependence Equations. In: In Proc. the SIGPLAN'92 Conference on Programming Language Design and Implementation. pp. 152–161 (1992)
14. More, T.: Axioms and Theorems for a Theory of Arrays. IBM J. Res. Dev. 17(2), 135–175 (Mar 1973), http://dx.doi.org/10.1147/rd.172.0135
15. Mullin, L.: A Mathematics of Arrays. PhD thesis, Syracuse University, December 1988 (1988)
16. Roth, G., Mellor-Crummey, J., Kennedy, K., Brickner, R.G.: Compiling Stencils in High Performance Fortran. In: Proceedings of the 1997 ACM/IEEE Conference on Supercomputing. pp. 1–20. SC '97, ACM, New York, NY, USA (1997), http://doi.acm.org/10.1145/509593.509605
17. S. Haney J. Crotinger, S.K., Smith, S.: Easy expression templates using PETE, the Portable Expression Template Engine. Technical Report LA-UR-99 (1999)
18. Sung, I.J., Stratton, J.A., Hwu, W.M.W.: Data Layout Transformation Exploiting Memory-level Parallelism in Structured Grid Many-core Applications. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques. pp. 513–522. PACT '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1854273.1854336
19. Tang, Y., Chowdhury, R.A., Kuszmaul, B.C., Luk, C.K., Leiserson, C.E.: The Pochoir Stencil Compiler. In: Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures. pp. 117–128. SPAA '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1989493.1989508
20. USQCD: QDP++. http://usqcd-software.github.io/qdpxx/ (2002)
21. Veldhuizen, T.: Expression Templates. C++ Report 7, 26–31 (1995)
22. Verdoolaege, S.: isl: An Integer Set Library for the Polyhedral Model, pp. 299–302. Springer Berlin Heidelberg, Berlin, Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-15582-6_49
23. Winter, F T and Clark, M A and Edwards, R G and Joo, B: A Framework for Lattice QCD Calculations on GPUs. In: 2014 IEEE 28th International Parallel and Distributed Processing Symposium. IEEE (may 2014), http://dx.doi.org/10.1109/IPDPS.2014.112
24. Wolf, M.E., Lam, M.S.: A Data Locality Optimizing Algorithm. In: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation. pp. 30–44. PLDI '91, ACM, New York, NY, USA (1991), http://doi.acm.org/10.1145/113445.113449
25. Xu, S., Gregg, D.: Semi-automatic Composition of Data Layout Transformations for Loop Vectorization. In: Network and Parallel Computing, pp. 485–496. Springer Science + Business Media (2014), http://dx.doi.org/10.1007/978-3-662-44917-2_40
26. Yan, Y., Lin, P.H., Liao, C., de Supinski, B.R., Quinlan, D.J.: Supporting Multiple Accelerators in High-level Programming Models. In: Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores. pp. 170–180. PMAM '15, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2712386.2712405