Exploiting Implicit Parallelism in Domain-Specific Languages

Vlad Ureche LAMP, I&C, EPFL

Abstract—In the recent years hardware evolution shifted from increasing execution speed to increasing the number of available execution units. As the aggregate computing power increases, the problem is how to use it efficiently: instead of running in a sequential manner, as before, software has to execute in separate tasks, each running in parallel on an execution unit. The dominating paradigm is explicit parallelism: software developers manually plan the execution and enforce the schedule using synchronization primitives. This is a tedious and bug-prone job.

An alternate approach is to shift the burden of parallelization to the compiler. The semantics of some high-level instructions, such as maps and filters, allow them to be executed as separate tasks: they are implicitly data-parallel. As we will see in the rest of this paper, exploiting the implicit parallelism in a language is the first step towards efficiently utilizing the processing power. Still, some considerations must also be taken into account in order to obtain good performance, as we will see in the discussed papers.

This work summarizes three papers in the field of languages and parallel execution: Single Assignment C; a language with built-in support for multi-dimensional arrays, Nested Data Parallelism in Haskell, a compiler technique that flattens nested parallelism to obtain uniform execution and finally Building-Blocks for Performance Oriented DSLs, a framework for multi-stage compilation and parallelization.

Index Terms—implicit parallelism, domain specific languages, high-dimensional arrays, data parallelism, nested parallelism

I. INTRODUCTION

In the recent years hardware evolution shifted from increasing execution speed to increasing the number of available execution units. Modern processors feature multiple execution cores, clusters of computers are commonly used for large data processing and graphical card processors provide hundreds of cores that can be used in parallel. The aggregate processing power is increasing, but using all of it efficiently is hard: instead of running in a sequential manner, as before, software has to execute parallel tasks which communicate and interact as little as possible. There are two paradigms for developing parallel software: on one hand we have explicit parallelism, where the programmer controls each of the parallel tasks and on the other hand we have implicit parallelism, when the compiler automatically generates parallel tasks from seemingly sequential operations.

The dominating model used in software development today is explicit parallelism: software developers manually split the execution into tasks, plan the succession of tasks based on data flow and enforce this succession by architecture-specific means. This places an important burden on programmers who are used to thinking and writing programs in a sequential manner. Furthermore, the need to learn multiple APIs and the peculiarities of each parallel environment greatly increase the effort necessary to create parallel programs [1], [2], [3]. The explicit parallelism model trades ease of use for performance: the tight control offered to the programmer can help in obtaining good performance, but programming becomes a tedious and error-prone job. Programmers must take multiple scenarios into account, such as cluster nodes failing, threads interleaving in a less common pattern or tasks simply encountering unexpected errors.

The opposite paradigm is exploiting implicit parallelism, where the burden of parallelization is shifted to the compiler. The programmer writes the code as a sequence of operations. If the operations are implicitly parallel, the compiler can generate code that splits the input data into chunks, feeds it to separate tasks working on different execution units and later joins back the data for use in the next operation. This approach trades simplicity for performance: it is easier for the programmer to think sequential and delegate parallelization to the compiler, but the compiler’s limited understanding of the program’s semantics translate to suboptimal memory access patterns, redundant synchronization and in general worse performance. Choosing the right abstractions in a language can increase the compiler’s ability to analyze and reason about the code, therefore increasing execution speed. As we will see in the rest of this paper, exploiting the implicit parallelism in a
language is just the first step towards efficiently utilizing the processing power, because other considerations also have to be taken into account in order to compete with explicitly parallel programs.

This work summarizes three papers in the field of array languages and parallel execution. We start with a motivating example, the Single Assignment C [4], a side-effect-free restriction of C with built-in support for multi-dimensional arrays. We then present an approach for transforming programs into uniform data parallel operations discussed in Nested Data Parallelism in Haskell [5] and finally Building-Blocks for Performance Oriented DSLs [6], a multi-stage compilation framework that provides the necessary infrastructure to ease building performance DSL compilers.

II. SINGLE ASSIGNMENT C

The Single Assignment C [4] paper contains both theoretical and practical contributions in array programming. On the theoretical side, the paper presents a model for embedding dense n-dimensional arrays with $O(1)$ access time in a side-effect-free language. Dense arrays are characterized by dimension and shape: the number of elements on each dimension. To increase expressivity, the SAC language features first-class, shape-invariant support for arrays and built-in shape-invariant comprehensions that increase language expressivity. The SAC type system infers array dimensions and shapes where possible. These design choices create an expressive language with high-level optimization opportunities, that the compiler fully exploits to generate fast programs.

On the practical side, the paper presents the Single Assignment C compiler. It parses, type checks and optimizes programs written in the SAC language and outputs C code that is further compiled to binaries. Several implementation decisions have been taken for the compiler:

- call by value semantics
- first-class and shape-invariant n-dimensional array support built into the compiler (as opposed to a library approach)
- garbage collection with reference counting, as done in Sisal [7]

The programs written in SAC are more expressive and run faster than their equivalents written in imperative array languages such as HPF [8].

The following subsections will explain how the decisions above impact the language and the compiler.

A. Core Language

The SAC language was designed as a functional restriction over C². The side-effect-free semantics lead to important properties not found in imperative languages: referential transparency, meaning any expression can be replaced by its value without affecting the overall program behavior and Church

1 the access time is bounded by a constant although on modern architectures individual access times can vary because of caching
2 we use the term functional restriction instead of functional language because SAC does not feature first-class support for functions.

Rosser property which ensures that expressions can be evaluated in any order and still yield the same result. These two properties enable better program analysis, more sophisticated optimization and parallel execution.

Using C as a starting point eased code translation while at the same time provided programmers with a known syntax. Starting from the C syntax, all side-effecting constructs were removed while maintaining similarity in all other aspects. To this end, several side-effecting elements of the syntax were eliminated: global values, references and pointers. Multiple assignments to the same variable were replaced by let bindings where inner values shadow the outer ones. The function body syntax was also restricted to become functional: the function can have a single return statement, as the last instruction in the function block.

B. Multidimensional Array Design

Multidimensional arrays are represented as contiguous memory blocks. To guarantee $O(1)$ access time to any element, the array is stored in memory as a contiguous block of data. Mapping block indices and multi-dimensional index vectors is done based on the shape vector. The shape vector contains the number of elements on each dimension and allows the program to check the correctness of index vectors: The values in the index vector must always be less than their corresponding shape vector, otherwise an out of bounds error is generated. The same block of data can be used with different shape vectors, as long as the product of their elements is the same. Reshaping is one of the basic operations provided by the language.

Data representation in SAC is uniform: all expressions are arrays. Regarding all the expressions in the language as arrays allows the SAC compiler to simplify the analysis and reason about shapes uniformly. This way, scalar expressions become 0-dimensional arrays while lists become 1-dimensional arrays. Array shapes and index vectors are no exception: they are 1-dimensional integer arrays, with their semantic meaning given by the way they are used.

SAC arrays are immutable. All functions and built-in operations are SAC return newly allocated arrays. While using immutable arrays eliminates side-effects, the code must be carefully optimized during compilation: mutable updates should be performed wherever possible. For the intermediary arrays that are not optimized away during compilation, a reference counting method is used for garbage collection. (SAC provides a series of built-in basic operations):

- element-wise arithmetic for integer and floating-point arrays, creating a new array with the result
- element-wise logic for boolean arrays, creating a new array with the result
- \(\text{dim}(array) \Rightarrow\) returns the dimension count of \(array\)
- \(\text{shape}(array) \Rightarrow\) returns the shape of \(array\)
- \(\text{sel}(iv, array) \Rightarrow\) returns an element, vector of block from \(array\) using the vector indexing semantics explained in the previous section, where \(iv\) is the index vector
- \(\text{reshape}(shp, array) \Rightarrow\) returns a new array with the contents in \(array\) and the shape given by \(shp\)
• cat(d, a, b) ⇒ returns the concatenation of the two arrays around axis d. The shapes of a and b must be equal except for the d-th position
• other selection and update procedures such as genarray, modarray, tile, shift, rotate, etc.

C. Shape-Invariant Array Comprehensions

"With loops", SAC’s array comprehensions, are the main method of manipulating arrays. They are first-class citizens in the language for three reasons:

• they are expressive high-level shape-invariant data-parallel operations that reduce the need to rely on nested for loops and complex indices
• most of the SAC basic array operations can be expressed in terms of with loops, making the language uniform
• being first-class citizens in SAC, they can be well optimized in the compiler, translating them to destructive updates and merged with loops

With loops are composed of two parts: The element selection and the processing stage. The element selection loops through the array index vectors and passes them to the processing stage. The element selection is done by specifying loop boundaries, strides and widths on each axis (if they are missing, the compiler can infer them from the arrays involved):

\[
\text{with}[\{\text{low} \lesssim | \leq \text{upp}\} \{\text{step \ step \ \ \text{width}}\}]
\]

Once the element selection is done, the with loop action follows: genarray, modarray and fold. The genarray operation creates an empty array and fills in the selected positions with the results of a given expression. Modarray takes an array and changes its values according to the with index vectors in the selection and the given expression. Since arrays are immutable, the result will be a new array while the base array remains unchanged. The fold operation is much like a reduce operation on blocks of a matrix. The signatures are as follows:

\[
\begin{align*}
\text{genarray} & \ (\text{shape}, \ iv \Rightarrow \text{expr}(iv)) \\
\text{modarray} & \ (\text{array}, \ iv \Rightarrow \text{expr}(iv)) \\
\text{fold} & \ (\text{oper, neutral}, \ iv \Rightarrow \text{expr}(iv))
\end{align*}
\]

The genarray action will start from an array of shape filled with null elements. For each index vector in the element selection, it will fill the corresponding cell with the value given by the expression expr(iv). The modarray action starts from the given array and changes elements in the comprehension with the values given by expr(iv). The returned array has the same shape as array. Finally, the fold operation takes blocks of the matrix and applies oper between each other. If \( IV = \{iv_0, iv_1, \ldots, iv_n\} \) is the set of index vectors generated by the element selection, the result of the fold operation is\(^1\):

\[
\begin{cases}
\text{neutral} & \text{if } IV = \emptyset \\
\text{expr}(iv_0) \odot \text{expr}(iv_1) \odot \ldots \odot \text{expr}(iv_n) & \text{if } IV \neq \emptyset
\end{cases}
\]

Basic operations in SAC are expressed in terms of with loops. Addition, for example, can be expressed as a genarray with loop. The index vector sweeps all positions of the array while the expression adds the elements together:

\[
\text{arr1} + \text{arr2} = \text{assert}(\text{shape}(a) = \text{shape}(b)) \\
\text{with}(iv)\text{genarray}(\text{shape}(a), \\
iv \Rightarrow \text{sel}(iv, \text{arr1}) + \text{sel}(iv, \text{arr2}))
\]

D. Type System

In SAC, shape information can be attached to an expression’s type, as the two go hand in hand. Since all expressions in SAC are arrays, restricting shapes can be done as part of the type system. The type system itself serves two important functions: detecting possible runtime errors at compile time and easing analysis in the optimization phases. Adding shape information in the type system serves both detecting static shape incompatibility and the later optimization phases.

The shape inference system must restrict the domain of shapes as much as possible while still accepting valid programs. A hierarchical shape information scheme was developed for SAC: the top case is knowing the exact shape of an expression. The middle cases in the hierarchy are knowing the dimension of the expression, but not the exact shape. Finally, the bottom value in the hierarchy is not knowing anything. The hierarchical shape system will accept any valid program and will not over constrain its expressions. It will be able to prove some of the domain restrictions statically while leaving others to be checked at runtime. In the above element-wise plus example, if the the dimensions of the arrays are known to be different, a compiler error will be issued letting the programmer know two arrays of different dimensions may not be added. On the other hand, if one shape is known (top) and the other is not known at all (bottom), the type system must accept the program, as at runtime the two shapes may actually be the same. In this case the runtime test is kept, as the incoming arguments may or may not be compatible.

Inferring shape information for functions requires inlining. The function signatures provide the types and shapes required for the arguments and the type and shape of the result. If we think of the identity function on integer arrays, it will take any shape and return any shape. The problem is correlating the fact that the output shape is always equal to the input shape. To obtain this result, SAC specializes the function for the incoming shape: it propagates the shape information of the arguments inside the function and analyzes the return shape. For example, if the argument to the identity function is known to have 2 dimensions, propagating this information in the function will output a 2-dimensional array, so the type system inferred more than just the formal return type specified in the function, any shape.

E. Code Optimization and Compilation

The SAC compiler performs a comprehensive set of optimizations, both high-level and low-level. The high-level
optimizations include inlining, dead code removal, copy propagation, common subexpression elimination, constant propagation, algebraic simplification, constant folding, loop optimizations and the most important, with loop folding. On the low-level C code generation part, the most important optimization is cache-aware with loop rewriting.

The most important high-level optimization is “with loop” folding. The key to this optimization is the equality relation on arrays: $\text{map } f \circ \text{map } g \equiv \text{map}(f \circ g)$. This shows the equivalence between successive with loops operating one after the other and single with loops, as shown in the example in Figure 1. In terms of speed with loop folding eliminates both temporary arrays and redundant iterations. One limitation of with loop folding is that it requires all bounds, steps and widths to be known statically. To run, the with loop folding phase relies on other high-level optimization phases to provide this information. Therefore, while the rest of the optimization phases do not bring significant performance gains by themselves, they are important for their role in enabling with loop folding and further low-level optimizations.

**Fig. 1: With loop folding example**

From the low-level optimizations, the most important is cache-aware “with loop” generation. With loops resulting from multiple folds can contain many sets of bounds and strides, each with a different expression, as can be seen in Figure 1. If each set of bounds is transformed into nested for loops, memory locality is not taken into account and this results in cache misses and speed loss. The SAC compiler rearranges operations by means of merging and splitting. Nested for loops are optimized one level at a time and the result is a cache-aware linear access set of nested for loops. The generated code can also benefit from optimizations in the C compiler, namely inlining for loops. Another two low-level optimizations are reference counting inference and destructive updates. If an array is not used anywhere else, modarray with loops can perform updates in-place, destroying the previous array but saving memory and initialization time.

**F. Benchmarks**

The paper presents the PDE1 benchmark taken from High Performance Fortran compiler comparisons. The benchmark consists of an algorithm for approximating three-dimensional poisson equations, implemented both in Fortran90/HPF and SAC. The benchmark implementation in SAC has 6 different versions, with an increasing use of shape-invariant operations. The results are shown in Table I. The two conclusions that can be derived for the PDE1 benchmark are that:

- the SAC compiler infers shapes and can perform aggressive optimizations even for of shape-invariant code
- the SAC compiler obtains performance on par with Fortran90/HPF

**G. Conclusions**

The Single Assignment C - efficient support for high-level array operations in a functional setting paper presents a functional restriction of the C language with first-class support for high-dimensional arrays and their comprehensions. The choice of a functionally-restricted dialect of C, call by value semantics, uniform representation of all expressions as shape-invariant arrays and high-level array comprehension format lead to the creation of a language that can be aggressively optimized to yield results comparable to imperative high performance array languages while proving better expressivity. The type system with its shape inference and function specialization (inlining) play an important role in statically inferring shapes. The effectiveness of the shape inference algorithm can be seen in the PDE1 benchmark version 5, where the entire program is written in a shape-invariant format with no performance penalty. Furthermore, the benchmarks in the paper show the important role played by “with loop” folding in cutting the execution time.

The SAC language is well suited for exploiting the implicit parallelism. Thanks to the high-level array comprehensions that ultimately translate to flat data-parallel operations, the SAC language is a very good candidate for parallelization.

**III. Harnessing the Multicores: Nested Data Parallelism in Haskell**

Harnessing the Multicores: Nested Data Parallelism in Haskell [5] presents a solution to the problem of obtaining stable and predictable speedups from parallel execution. Data

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Time taken</th>
<th>Memory used</th>
</tr>
</thead>
<tbody>
<tr>
<td>HPF</td>
<td>7.16s</td>
<td>470 MB</td>
</tr>
<tr>
<td>SAC - Low Level</td>
<td>2.80s</td>
<td>267 MB</td>
</tr>
<tr>
<td>SAC - Relax 1</td>
<td>2.80s</td>
<td>267 MB</td>
</tr>
<tr>
<td>SAC - Relax 2</td>
<td>2.73s</td>
<td>267 MB</td>
</tr>
<tr>
<td>SAC - Relax 3</td>
<td>2.84s</td>
<td>267 MB</td>
</tr>
<tr>
<td>SAC - Relax 4</td>
<td>2.92s</td>
<td>267 MB</td>
</tr>
<tr>
<td>SAC - Relax 5</td>
<td>2.88s</td>
<td>267 MB</td>
</tr>
</tbody>
</table>

TABLE I: SAC and Fortran90/HPF results for the PDE1 benchmark
parallel algorithms, where the same operation is performed in parallel over different chunks of data, are well supported by current platforms and proved their efficiency over time. Data parallelism requires that processing any element takes roughly the same time. But not all algorithms fit this description: Common cases such as recursive tree traversal or sparse matrix operations use recursive calls or unbalanced processing time per element, leading to unbalanced jobs and ultimately bad parallel performance. While work stealing can alleviate the problem by re-balancing the jobs, this paper presents a bolder approach: vectorization, a program-wide transformation that adapts nested parallelism to tried-and-tested flat data parallelism.

The paper builds upon the experience of the NESL [9] nested parallel language, adapting and developing it to work with the GHC Haskell compiler. The prototype presented in the paper, Data Parallel Haskell, features flattening of higher-order functions and solves the problem of mixing parametric and non-parametric arrays in the context of polymorphism. This report will only focus on the contributions of the paper, skipping two chapters due to space constraints: the vectorization algorithm will only be briefly mentioned while the motivating example in the paper, Barnes-Hut n-body interaction will be completely skipped.

A. Intuition

To explain why jobs become unbalanced and how vectorization eliminates the problem, we can look at the task of recursively traversing a tree data structure. The left side of Figure 2 shows a tree where only white nodes will be traversed in depth. Trying to traverse the tree in parallel will result in significant imbalances between jobs, even with work stealing. On the other hand, if we grouped the nodes by level, each group could be traversed in a data parallel fashion. Apart from the result, the level traversal returns the leaves, which are used as input for the next traversal. This is shown on the right side of Figure 2. In the above example, all parallel jobs, assigned to workers, are equal and use all the processing power from the second traversal onwards. The most important point in flattening is vectorization: modifying the function to operate on a parallel array at a time instead of a single element.

```
<table>
<thead>
<tr>
<th>W1</th>
<th>W2</th>
<th>W3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Fig. 2: Tree (left) and traversal with vectorization (right). The vertical lines separate flat data parallel calls while horizontal lines separate jobs that can be assigned to worker nodes.

B. Non-Parametric Representation

In data parallelism, the way objects are placed in the memory can make an important speed difference. The default Haskell representation of arrays consists of objects allocated on the heap and arrays containing pointers to them. This needs to be changed for two reasons: pointers create indirection which slows down access and objects randomly allocated in the heap are not contiguous, so data locality and caching are affected. Fortunately both targets can be met at once by switching to non-parametric representations: the array is stored as a contiguous block of memory with objects stored one after the other. For objects of constant size, this eliminates the pointer indirection and enables data locality. There are three challenges to this: one is transforming the existing data structures to arrays with good access patterns, the second is transforming the functions so they operate on parallel arrays and the third is maintaining compatibility with libraries and other parts of the program where arrays remain parametric.

For base types arrays are transformed into simple blocks of memory. A trickier transformation occurs for higher-order arrays, structures and recursive types. For higher-order arrays, such as \( [: [: a : ] : ] \) the transformation is based on linearization: we store a data structure with both the lengths of the inner arrays and a linearization of all the elements in the higher-order array. This ensures we have fast access time to array elements while at the same time the array can be processed as a linear structure. Exactly the same reasoning can be applied to array nestings of any depth, by storing an array of lengths for each level of nesting.

Transforming tuples and structures rearranges the internal representation. Tuple and structure transformation becomes more complex: an array of tuples \( [: (a, b) :] \) is transformed into a tuple that stores arrays of the elements: \( (Int, PA a, PA b) \). The \( PA \) is the rewritten version of the array in its non-parametric flavor. The integer at the beginning is the length of the original array. It is important to notice the data structures commute: before, we had an array of tuples, now we have a tuple of arrays. Commutativity enables further rewriting of \( [: a :] \) and \( [: b :] \) to \( PA a \) and \( PA b \), so the recursive rewriting can reach all the way to the innermost layers of the data structures. Structures with named elements can also be seen as tuples, therefore the same rule applies to them. Last but not least, recursive data structures can be rewritten using exactly the same pattern, just that some part will reference the structure once again: \( data Tree = Node Data [: Tree :] \) becomes \( data PA Tree = ATree Int (PA Data) (PA (PA Tree)) \) where the \( Data \) is the node data and the array of trees are the children. The integer's purpose is to store the length of the array. Other data structures can be transformed following the same algorithm.

A major challenge was keeping the data representation compatible with the rest of the Haskell library and programs. Parameterized arrays contained polymorphic functions such as map, filter, length, etc. These functions cannot work for non-parametric arrays, as the storage method is different. Therefore the functions had to be re-implemented separately. But the functions on parametric arrays were polymorphic while for the
non-parametric arrays they were mono-morphic. Instantiating all types in the polymorphic class was impossible as GHC, the Haskell compiler, allows separate compilation, so the entire program is not available at once to know all specializations for the polymorphic functions. The solution came as type classes [10], which enabled preparing a type class object for each type of non-parametric array while offering polymorphic functions on the outside. Then each transformed data structure had to create its own type class instance, PAElem a, containing implementations of the common functions.

C. Higher-order Function Representation and Vectorization

Vectorization requires all functions operating on scalar elements be translated to their array equivalents. This transformation is the core of eliminating nested parallelism: it defers nested calls made while processing individual elements until the array has been processed. The uniform array representation makes it possible to have constant-time processing for each element. Coupled with deferring recursive calls, this leads to flat data parallelism that offers stable and predictable performance. We will not present this phase in detail here, as it’s not a major contribution of the current paper. One thing to notice, though, is that simple vectorization must be adapted to the functional nature of Haskell: arrays can contain not only data but functions and also closures. We will look at several problems that appear in the following paragraphs.

The result of a function vectorization must carry both the scalar function and the vectorized, or lifted, equivalent. We can think of the following example:

\[ g :: (Int -> Int) \rightarrow (Int, [: Int :]) \]

where \( g f = (f 2, mapP f [: 1 2 3:]) \). Creating the lifted version of \( f \) here is not enough: \( g \) needs to apply both the scalar function in \( f 2 \) and the lifted one in \( mapP f [: 1 2 3:] \). Going even deeper, we see that just carrying the scalar and lifted functions is not enough: functions that return closures may also need to be stored in arrays. Storing closures in arrays must be done in the same optimized manner as for data: there are two pointers to code, for the scalar and lifted functions and a list of closure environments. This layout enables efficient mapping over closure arrays.

D. Parallel Optimization

After vectorization, the code is transformed to the Single Program Multiple Data model. This model is the generalization of data parallelism on all architectures: it features operations such as \( \text{splitD}, \text{map}, \text{filter} \) and \( \text{joinD} \). The normal workflow is splitting the data among the available Processing Elements, performing an operation and later joining the data back into a single array.

The SPMD model allows optimizing the code by inlining, eliminating synchronization and loop fusion. Each map in the original program ultimately becomes a \( \text{joinD}(\text{map f(splitD(data))}) \). By inlining the operations that generate data, the program can eliminate sequences of synchronization such as \( \text{splitD}(\text{joinD}(\ldots)) \). This must be done in a careful way, as filter operations normally unbalance the chunks on each processor. Furthermore, once synchronization is eliminated, successive maps can be eliminated by fusion: \( \text{map f map g(data)} \) is the same as \( \text{map f g(data)} \). Fusion creates code that traverses the data only once, therefore runs faster.

IV. BUILDING-BLOCKS FOR PERFORMANCE ORIENTED DSLS

The Building-Blocks for Performance Oriented DSLs [6] paper presents a novel approach to creating domain specific language compilers that target heterogeneous parallel architectures. Performance-conscious code usually loses touch with the specification as data structures and control flows are reorganized. The problem is even more visible when targeting parallel architectures: the code becomes tailored for the parallel execution model and sprinkled with calls to architecture-specific API. Ideally compilers should take care of bridging the gap between specifications and low-level code, but this requires intimate understanding of the data structures and the operations performed. This is very hard to achieve in generic imperative languages that offer many ways to interact with data, especially as side effects. Domain specific languages, on the other hand, restrict the scope of the language and have a higher-level understanding of the program, offering them a better shot at optimizing the code.

Domain specific language development requires significant effort, as there are two complex tasks involved: on one hand, creating a language that is expressive, flexible and can be thoroughly optimized and on the other hand developing a compiler that uses all the optimization opportunities, at different levels of abstraction, to obtain good performance. A commonly used approach is to embed the DSL in a generic language as a library. While it side-steps the need to build a compiler from scratch, it results in limited ability to optimize the programs, especially since a whole-program view is not available. Lightweight modular staging [11] is a multi-stage compilation approach to optimizing embedded DSL code. Delite, the framework presented in this paper builds upon LMS while offering important building blocks for targeting heterogeneous parallel architectures.

A. Embedding a Domain Specific Language

Lightweight modular staging is the starting point for embedding a DSL. Although parsing, semantic checking and code generation are common parts of any compiler, they are not trivial and require time to develop. Fortunately, this compiler infrastructure can be reused by embedding the DSL a generic language, as if it was a library. Embedding comes at the price of limiting the DSL syntax to the constructs that conform to the generic language. This can be solved by embedding the DSL in a flexible language like Scala [12] which allows user-friendly DSL syntax [13]. The second problem is the lack of a whole-program or even whole-function view. It is solved by the multi-staged approach in LMS: the program is compiled and ran, at staging time, and behaves like a code generator. Instead of generating the result of the operations, it generates optimized code to perform them. In the second
stage, the newly generated code is compiled and run, this time generating the desired result. A closer look reveals the reason behind the need to compile and run twice: in the first run, called staging, LMS records the program’s operations into an intermediary representation similar to a tree. The intermediary representation is then optimized with domain-specific knowledge and output as new code. The new code is equivalent to the original program, modulo the flattening and optimizations, which we will look at in the next section.

LMS relies types to mark expressions that will be available in the second stage. Not all of a program written in a DSL embedded in LMS is automatically transformed into an intermediary representation: Marking expressions that take part in the IR, called staged expressions, is done by adding a parametric wrapper around a base type. For example, expressions of type `Int` will be computed in the first run, while expressions of type `Rep[Int]` will appear in the intermediary representation. Expressions of any type can be converted to their staged type, as the staged type intuitively tells the compiler to defer the computation to the next stage and optimize it. Functions that take staged types as arguments also return staged types: intuitively, the arguments’ values will only be known in the second execution, so the result cannot be available immediately, it has to be deferred. All staged expressions become nodes in the intermediary representation and are be output as code.

LMS stages control structures and tracks effects. Control structures such as if statements and loops can also be staged: an if-then-else branch with a `Rep[Boolean]` condition returning `Rep[T]` on both branches will become a node in the intermediary representation. On the contrary, an if-then-else with a `Boolean` condition will be executed at runtime, even if it returns staged types. Side effects such as printing a value and writing to a variable are also tracked by the framework: the order of the effects is remembered and generated code always executes the effects in the same order (or an equivalent one). To accommodate effects, data and control flow dependencies, the IR consists of a sea of nodes rather than a list or a tree. Dependencies are introduced to account for effects and data flow dependencies. This representation encourages code motion, as we will see later.

The DSL composition and code generation are based on the principle of mixing in the necessary blocks. LMS contains generic language components such as arithmetic support, loop support, if-then-else constructs, support for mutable variables etc. Generic components can be mixed with custom components designed by the DSL author to produce the final domain specific language. The same approach is taken in all stages: LMS provides specific optimizations and code generation for the generic language components while the custom components need to provide their own.

The Delite framework enables DSL developers to target heterogeneous architectures. Delite is where the semantics of the DSL come into play: the implicitly data-parallel operations can be marked as such, providing Delite with semantic understanding of the data-parallelism in the operations. With this high-level information available at staging time, operations can be optimized in several ways, such as merging them together, eliminating intermediary storages, etc.

Even with data-parallelism information, Delite can not schedule the program until the exact architecture is known. The scheduling in Delite is done at runtime, depending on the available CPU cores and whether or not a GPU core is present. For this reason, instead of a program, Delite generates kernels, separate pieces of code that can run in parallel. It also generates a graph that shows data dependencies between kernels. Since the architecture each kernel will run on is not known at runtime, Delite generates kernels for all available architectures.

Once the parallel kernels are constructed, the Delite runtime decides on the scheduling and puts the entire program together. The generated set of kernels does not tie the program to a certain architecture: once the delite framework is invoked, it decides on a static schedule that fulfills kernel dependencies. Kernels may be run on any number of processors or on the GPU. The runtime will find the fastest schedule and take care of the data communication and synchronization between kernels. In order to avoid redundant communication, kernels that depend on each other are clustered together and run on the same type of processor.

### 2. Building-Blocks for Performance DSLs

Delite and LMS perform a number of common compiler optimizations at no cost to the DSL author. Common subexpression elimination, dead code elimination, constant folding and code motion are automatically performed by the compiler at staging time. Code motion can lift constant expressions outside loops, as long as they do not depend on any iteration-time values. The opposite of lifting expressions is nesting deeper: for example, expressions can be moved inside then/else branches if they are used in one branch but not the other. Code motion can also be used on custom language components, as the semantics of each operation is defined by the DSL authors. On the other hand, the generic language components already define their code motion semantics so the DSL author gets the optimization at no cost. The other three optimizations are language-wide and apply on any combination of language components.

By specifying the semantics of loops and data structures, Delite can merge loops and eliminate intermediary data structures. Most compilers have a good understanding of loops, but are not able to optimize them either because of the side-effects they produce or because of the lack of data structure information. In the case of Delite, loop operations can be fused together and access to internal data structures can be optimized. These operations create similar transformations to the Single Assignment C folding example in Figure 1.

Higher-order abstractions for control (functions, closures) and data (object, boxing) are necessary for large scale programs to evolve, but hinder optimizations. For example the analysis required for higher-order function aliasing is complex and must always be conservative. The staging approach extracts the useful computations while eliminating the abstractions, enabling better reasoning and code optimization. One example is flattening higher-order functions: by changing the type of the function in the signature of map, one
can force flattening or allow higher-order functions in the language: $\text{f: Rep}[A] \Rightarrow \text{Rep}[B]$ forces the flattening in the intermediary representation. While this is beneficial for optimization, higher-order functions could also be allowed in the language: with $\text{f: Rep}[A \Rightarrow B]$ we signal entire function is a dynamic value, not just its input and output.

C. Benchmarks

The paper presents a case study of the OptiML machine learning DSL. The test program used for the benchmarks implements an algorithm used in bioinformatics. Two implementations are presented: a hand-optimized example in C++ and its equivalent in OptiML. The difference in abstraction level between the C++ program and the OptiML is significant. Still, the Delite framework proves its usefulness by transforming the high-level OptiML code such that it obtains the same performance as the hand-optimized C++ code running on up to 8 cores. GPU vs 8 CPU cores benchmarks are also presented and prove that performance is greatly dependant on application at hand. The conclusion is that heterogeneous models are best, as they can take advantage of the architecture difference.

D. Conclusions

The paper presents the compiler building blocks offered by the Delite framework and shows how these can be used to bridge the gap between high-level programs and optimized low-level code. The paper highlights the advantages of using the Delite framework to build DSLs at different layers. Using the multi-staging approach to embedding domain specific languages in Scala eliminates the need for common but complex compiler infrastructure, all without losing the ability to optimize code. The Lightweight Modular Staging framework then provides the common language building blocks, such as arithmetic, mutable structures and control structures. The author needs to prepare the DSL-specific primitives, optimization and semantics. The next layer is internal optimization: just by giving Delite semantic information about the DSL primitives such as parallel loop type, data structures involved and effects, it is able to optimize the code to the same performance as an experienced programmer. At the last stage, LMS and Delite provide code generation for the common language components so the DSL authors only need to provide code generation for their own components.

V. Conclusions

The three summarized papers draw the direction of implicit data parallelism in domain specific languages. The first paper presents Single Assignment C, an array language well suited to optimization and a compiler that obtains very good performance. The paper also describes the compilation phases a program has to go through to obtain the same performance, prompting us to try and reproduce the same results. The Harnessing the Multicores: Nested Data Parallelism in Haskell paper presents an whole program transformation approach to obtaining stable and robust performance in the case of nested parallel jobs. Finally, the Building Blocks for Performance-Oriented DSLs presents a novel method of creating domain specific languages with limited effort and Delite a framework for parallel execution.

The three papers sketch the directions of current and future research. The Delite framework evolution is driven by the embedding of new domain specific languages, each with its own set of requirements. Therefore, with each language added, the framework’s support evolves to become more comprehensive and easier to use. So far, semester project work made progress in two directions: embedding the SAC domain specific language in LMS and later Delite and improving the SAC shape inference algorithm. We are now at the point where many of the optimizations in SAC must be implemented in the Delite DSL in order to obtain running times similar to the ones published in the paper. This proved to be a tough challenge, as the structure of Delite does not naturally accommodate the patterns required for optimizing SAC. Therefore the short-term goals are to implementing the multi-stage rewriting necessary to optimize SAC code in Delite, producing new optimization patterns that will also benefit other domain specific languages embedded using Delite. The long-term goals include finding new targets and with complex requirements that enable Delite to evolve and become a general platform for embedding a large array of domain specific languages.

REFERENCES