A development platform for embedded domain-specific languages

Shigeru Chiba, YungYu Zhuang, Thanh-Chung Dao

Abstract The use of domain-specific languages (DSL) is a promising approach to helping programmers write an efficient program for high-performance computing. The programmers would feel difficulties in writing such a program by hand with only low-level abstractions, such as arrays and loops, provided by a general-purpose language. This chapter presents our new implementation technique for domain-specific languages. Since existing techniques are not satisfactory, we developed our technique called *deep reification*. This chapter also presents *Bytespresso*, which is our prototype system to use deep reification. Several Java-embedded DSLs implemented with Bytespresso are presented to assess the effectiveness of deep reification and Bytespresso. Program fragments written in these DSLs are embedded in Java but they are dynamically offloaded to native hardware to obtain good execution performance. Since they are embedded in Java, the syntax of Java is reused by those DSLs and hence the development costs of these DSLs are reduced.

1 Embedded Domain-Specific Languages

The usefulness and necessity of domain-specific languages (DSLs) is getting widely recognized in the high-performance computing (HPC) area. DSLs provide higher-level abstractions for a specific domain than general-purpose languages. These abstractions consist of domain-specific data types and operators and they hide various techniques for their efficient implementation from the programmers' views. We first

Shigeru Chiba, The University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113-8656, Japan, e-mail: chiba@acm.org. YungYu Zhuang, National Central University, No. 300, Zhongda Rd., Zhongli District, Taoyuan City 32001, Taiwan (R.O.C.), e-mail: yungyu@acm.org. Thanh-Chung Dao, School of Information and Communication Technology, Hanoi University of Science and Technology, No 1 Dai Co Viet, Hanoi, Vietnam, e-mail: chungdt@soict.hust.edu.vn. This is a preprint of the book chapter 8 of "Advanced Software Technologies for Post-Peta Scale Computing: The Japanese Post-Peta CREST Research Project," Mitsuhisa Sato ed., Springer Singapore, pp.139–161, 2019.

briefly overview the motivation of this approach by presenting several well-known implementation techniques for it.

1.1 Domain-specific data types and operators

Providing useful data types and operators for a particular application domain is a promising approach to reduce programming costs in high performance computing. Since the underlying hardware and software stack is getting complicated, writing a raw Fortran or C/C++ program with built-in data types for simple arrays is getting a tedious task. During writing such a program, programmers have to consider various non-functional concerns such as how to exploit parallel computation provided by hardware, how to improve a cache hit ratio, and how to replicate data among distributed nodes. If domain-specific data types and operators are provided to abstract these non-functional concerns away, programmers can more focus on application-specific concerns. They would not be bothered about parallelism or distribution and hence reduce programming costs.

Such data types and operators can be implemented as a library in object-oriented languages. Since objects encapsulate implementation details of data manipulation and they provide methods as operators for processing the data, objects and methods are appropriate vehicles for implementing data types and operators for high-performance computing. We do not have to modify a programming language to provide data types and operators. Suppose that we have a library providing `Matrix` and `Vector` objects. Then we would be able to write the following code:

```
d = (a * p) * q
```

Here, `a` is a `Matrix` object, `p` and `q` are `Vector` objects, and `d` is a variable of `double`. The first `*` is a multiply method in the `Matrix` class and the second is a method in `Vector`. If the language does not support operator overloading, the code above would be like this:

```
d = (a.multiply(p)).multiply(q)
```

In either case, implementation details will be hidden from the programmer. `a` might be a sparse matrix and uses the compressed sparse row format. It might be a large matrix and the data are allocated on multiple distributed nodes where data are exchanged through a MPI library. Since the implementation of the `Matrix` and `Vector` is provided by our library, the programmer only has to select an appropriate implementation, for example, by selecting a subclass `SparseMatrixByMPI` of `Matrix` when creating the object that the variable `a` refers to.

```
Func f = (Vec4Array pos, int i, Vec3 pi, float wi) -> {
  Vec3 a = pos.sum((Vec4Array p, int j, Vec3 pj, float wj)->{
    Vec3 r = pi.sub(pj);
    float ra = reciprocalSqrt(r.mult(r) + soft);
    return r.scale(wj * (ra * ra * ra));
  });
  Vec3 v2 = vel.get(i).add(a.scale(delta)).scale(damping);
  vel.set(i, v2);
  return pi.add(v2.scale(delta));
};
```

Listing 1 N-body simulation in Java

1.2 C++ template libraries

A challenge of the approach to providing domain-specific data types and operators by object orientation is efficient implementation. Despite vigorous research activities for a long time, object-oriented mechanisms such as method calls tend to involve runtime penalties.

Let us see the example taken from our paper [2]. Listing 1 written in Java defines the kernel computation of the all-pairs approach to N-body simulation [9]. `Func`, `Vec4Array`, and `Vec3` are data types provided by a library. `Func` is a type of lambda expression. `Vec4Array` is an array of four-dimensional vectors. `Vec3` is a three-dimensional vector. `reciprocalSqrt` is an operator provided by the library as well as methods in the library classes, such as `add` and `sub` (subtraction). Since the programmer does not have to implement these classes, Listing 1 is a natural program derived from the following formula:

$$a_i = \textstyle\sum_j w_j (r \cdot r + \varepsilon^2)^{-\frac{3}{2}} r \quad \text{where } r = p_j - p_i \tag{1}$$

$$v'_i = \delta(v_i + \Delta t \cdot a_i) \tag{2}$$

$$p'_i = p_i + \Delta t \cdot v'_i \tag{3}$$

Here, a_i corresponds to the variable `a`. v_i corresponds to `vel.get(i)` and v'_i corresponds to the variable `v2`. p_i is the value returned by the lambda expression `f`. Note that $\sum$ is represented by the `sum` method on `pos`, which is an array of positions p. The `sum` method takes a lambda expression to compute $w_j(r \cdot r + \varepsilon^2)^{-\frac{3}{2}} r$ and hides how to iterate over array elements.

Although the program is straightforward translation into Java except that domain-specific operators are not simple symbols but methods, the runtime performance is not satisfactory when compared with the equivalent C code optimized by hand without using higher-level abstraction such as objects and a lambda expression. Figure 1 shows the execution performance of the N-body program written in several languages. We used OpenJDK (version 1.8.0_151), the Intel C compiler (version 17.0.1) with -fast, GCC 5.4.0 with -Ofast, and Clang 5.0.0 with -Ofast. All the

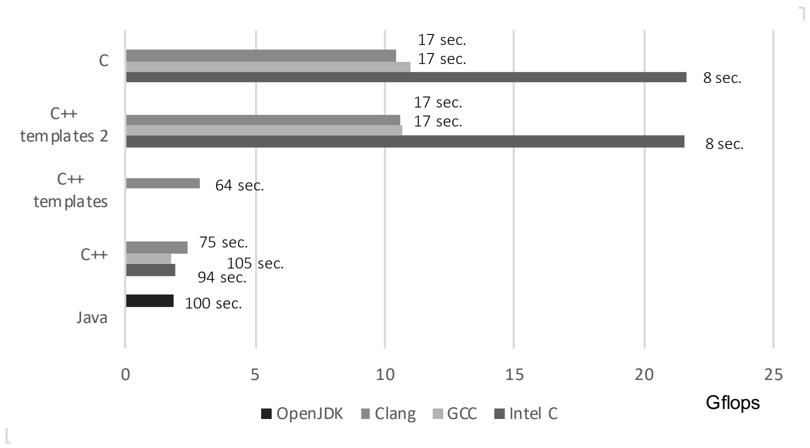

Fig. 1 The execution performance of N-body programs

```
static const auto func
  = [vel](Vec4Array* pos, int i, Vec3 pi, float wi) -> Vec3 {
      Vec3 a = pos->sum([&pi](Vec4Array* pos, int j, Vec3 pj,
                              float wj) -> Vec3 {
          Vec3 r = pi.sub(pj);
          float ra = 1.0f / sqrtf(r.mult(r) + 0.01f);
          return r.scale(wj * (ra * ra * ra));
      });
      Vec3 v2 = (vel->get(i)).add(a.scale(0.016f)).scale(1.0f);
      vel->set(i, v2);
      return pi.add(v2.scale(0.016f));
};
```

Listing 2 N-body simulation in C++

programs were run on a machine with dual Intel Xeon E5-2637v3. The hardware and compilers that we used for this experiment were slightly upgraded from those in our paper [2]. Although the execution performance depends on the optimization by a compiler and even minor differences of library design, the performance of Listing 1 in Java is not comparable with the equivalent C code. Furthermore, a C++ program naively translated from Listing 1 was also slow. As presented in Listing 2, this C++ program also uses lambda expressions and several objects such as `Vec3` and `Vec4Array`.

To avoid runtime penalties due to object orientation, domain-specific data types and operators in C++ tend to be implemented by a template library. In the case of our N-body program, the implementation using templates did not significantly improve the performance. "C++ templates" in Figure 1 indicates the performance of our implementation effort using templates. Since in C++ 17 a lambda expression can be passed as a template argument in only a restricted manner, our template library for the N-body program could be compiled only by Clang and the performance improvement was not satisfactory. “C++ templates 2” in Figure 1 indicates the performance of another implementation, in which the users do not describe kernel computation in the form of lambda expression. They instead write it in the form of a class. Although its performance is comparable to hand-optimized C code as shown in Figure 1, its programming interface is not satisfactory since the users have to write a new template class.

A main trick of performance optimization by templates is to pass compile-time constant values to a template as template arguments so that the resulting code of instantiating the template will be specialized for those template arguments. Hence when a library designer designs her template library, she has to carefully separate constant part of data types and operators from the rest that are dynamically given at runtime. For our example in Listing 1, the `sum` method on `pos` should be specialized for the given lambda expression the value of the variable `vel` might be given at runtime. Furthermore, a library designer has to consider the constraints on template arguments. For example, a string literal cannot be a template argument. We cannot specialize two objects that refer to each other:

```
template <typename T> class A {
  T b;
};
template <typename T> class B {
  T a;
};
```

If the member variable `b` refers to an instance of `B` and `a` refers to an instance of `A`, then we would want to specialize the type of `b` into `B` and the type of `a` into `A`. However, such specialization would be obtained only by `A<B<A<B<...>>>` and `B<A<B<...>>>`, which are not feasible.

Furthermore, we cannot predict with confidence that such specialization is properly performed by a C++ compiler and it actually improves execution performance. This depends on the compiler implementation and a library designer has to struggle it in a try-and-error fashion. Limitations of C++ template libraries would be that this trade-off between a user-friendly programming interface and execution performance is not predictable.

Finally, this approach using a C++ template is only applicable when a program is written in C++. Although most programs for high-performance computing could be written in C++, some programs might be written in other languages such as Go. When we want to use OpenCL, this approach is not effective. Since a kernel function in OpenCL is described in the form of string literals (or character strings), the specialization by C++ templates is not naively applicable to an OpenCL kernel function.

1.3 Pragmas

Since a C++ library with templates has some limitations, there have been other approaches proposed. The most extreme approach is to develop an external domain-specific language (DSL). It is a new language designed for data types and operators for a particular application-domain. Although it can provide the most natural syntax and potentially the best optimizing compiler, a drawback of this approach is its development cost. Not only a compiler but also a development environment such as an editor have to be developed.

Another approach with lower development costs is to add pragmas to an existing language. Pragmas customize a compilation process, in particular, how target native code is generated, although it does not change syntax. OpenMP is one of the most successful pragma-based systems. A `for` statement with the `parallel` pragma of OpenMP can be regarded as a domain-specific operator for parallel looping although the syntactic expression of the `for` statement is still a normal one. In other words, pragmas implement domain-specific data types and operators by changing the semantics of existing language constructs while keeping the original presentation. In practice, the semantic changes are restricted to be compatible to the original semantics. The programs including pragmas should run even when all the pragmas are ignored. For example, a `for` statement with the OpenMP pragma should be a valid `for` statement even when the pragma is removed.

For development of pragmas, we can use various compiler frameworks such as Eclipse [14], ROSE [10], and Roslyn [8]. GCC and LLVM compilers can be also used as a compiler framework. They mitigate relatively high development costs of pragmas. Most pragmas can be implemented by a translator of abstract syntax trees (ASTs) from original trees into transformed trees according to pragmas.

A drawback of this pragma approach is that syntactic presentation is restricted. A pragma can only add extra semantics to an existing language construct such as a `for` statement. The syntax or presentation of that language construct cannot be changed. This restriction on syntax may also complicate the behavior of domain-specific data types and operators by pragmas. For example, since OpenMP pragmas cannot control what programmers write in the body of a `for` statement, the programmers may write an arbitrary body. Hence they may be surprised at unexpected behavior of the statement when the computation in the body has forward or backward dependency. For example,

```
#pragma omp parallel for
for (int i = 0; i < N; i++)
  array[i] += array[i - 1];
```

The result of this `for` statement might be different from the result of the `for` statement without the pragma since OpenMP does not guarantee that `array[i]` is updated after updating `array[i-1]`. If custom syntax can be provided for parallel `for` statements, providing such syntax would be preferable so that programmers could not write a body including forward or backward dependence:

```
parallel_for (array, N) {
  return self + 1.0;
}
```

Here, `parallel_for` is a custom `for` statement with new syntax. It updates every element of `array` with size `N`. The new value is obtained by executing the body `self + 1.0`. The built-in variable `self` refers to the old value of each element. Although it enables only a limited kind of computation, the programmers cannot write an expression with forward or backward dependence. Restricting the expressiveness of data types and operators is sometime useful to avoid unexpected behavior.

1.4 Deep Embedding

Deep embedding is a technique for implementing an embedded domain-specific language (DSL). An embedded DSL is a library providing domain-specific data types and operators through an application programming interface (API) regarded as a DSL. Deep embedding is a variant of the technique called fluent API [3]. The functionality of a library with a fluent API is invoked by a chain of methods.

```
Matrix m;
Vector v, p, q;
   ...
q = m.mult(v).add(p);
```

The variables `m` refers to a `Matrix` object and `v`, `p`, and `q` refer to `Vector` objects. The last line above expresses $q = m * p + v$, matrix-vector multiplication and vector addition. The `mult` method executes multiplication and the `add` method executes addition. These methods return the resulting vector.

A deep-embedding library, however, provides methods returning an abstract syntax tree (AST). For the example above, both `mult` and `add` return an AST corresponding to an expression for multiplication or addition. Hence, the type of `q` has to be changed from `Vector` to `VectorExpr` (vector expression). To obtain the value of vector type, a method for materialization, for example, `eval`, has to be called on the AST.

```
Matrix m;
Vector v, p, r;
VectorExpr q;

   ...
q = m.mult(v).add(p);
r = q.eval();
```

The result of $m * v + p$ is stored in `r`. A unique feature of deep embedding is that programmers explicitly specify when an AST is materialized (by calling `eval`) into a value. Hence, a deep-embedding library can generate an optimized program for executing an AST to be materialized when the materialization method such as `eval` is called. Then, it can execute the program generated on demand so that it can get better execution performance. The generated program does not have to be written in the host language. When a deep-embedding library is for Java, it may generate a C++ or assembly program. It may directly generate native machine code [13].

Since a deep-embedding library generates a program, it is similar to an external-DSL compiler but it is a library embedded in its host programming language. The program generation is performed at runtime as a just-in-time (JIT) compiler does. A deep-embedding library is more portable than external DSLs or the pragma approach; it does not require a custom compiler or language processor.

Although the syntactic presentation of the code above is somewhat ugly, it could be improved, for example, if the host language supports operator overloading:

```
q = (m * v) + p;
r = q.eval();
```

In Scala, dots and parentheses can be omitted when a method takes a single parameter. Hence programmers can write the following code:

```
q = m mult v add p
```

It can be also possible to implement a precedence rule among `mult` and `add` by exploiting types [7]. However, in either cases, the `eval` method is still necessary for explicit materialization.

Explicit materialization is a drawback of deep embedding since programmers are aware of AST construction. They also have to distinguish AST types and value types such as `VectorExpr` and `Vector`. This drawback is mitigated in Lightweight Modular Staging (LMS) [12] by relying on Scala's powerful type system and advanced features such as user-defined implicit conversion (and often Scala-Virtualized compiler extension [11]). Yin-Yang [5] uses Scala's macro system to further mitigate this drawback. These techniques for mitigation, however, are not available in other main-stream programming languages, which do not provide as powerful programming capabilities as Scala's.

2 Deep reification

In our paper [2], we proposed an alternate approach called deep reification. Like deep embedding, deep reification enables a library providing domain-specific data types and operators through a language-like API.

The idea of deep reification is simple. Deep reification is a language mechanism for obtaining the AST of the source code of a given lambda expression (or a function closure) at *runtime*. Deep reification obtains not only the AST of the given lambda expression but also the ASTs of all the methods directly or indirectly invoked from that lambda expression. It can be regarded as a mechanism for obtaining an abstract syntax *forest*. The obtained ASTs also come with runtime values captured by the given lambda expression, and accessed from the directly or indirectly invoked methods. All the types appearing in the obtained ASTs are also collected.

```
dsl.run(() -> {
  pos1.tabulate(i -> new Vec4(i, i, i, 2));
  vel.tabulate(i -> new Vec4(i, i, i, 2));
  dsl.repeat(R, () -> {
    pos2.map(f, pos1);
    pos1.map(f, pos2);
  });
  Vec3 g = pos1.sum((Vec4Array pos, int i, Vec3 v, float w)
                    -> new Vec3(v.x / w, v.y / w, v.z / w));
  Util.print(g.x / N).println();
});
```

Listing 3 Deep reification for N-body simulation

Listing 3 shows an example. It performs the N-body simulation using the lambda expression `f` shown in Listing 1. The `run` method on `dsl` performs deep reification. It takes a lambda expression and obtains the AST of its source code. Since this lambda expression refers to `f`, which is another lambda expression in Listing 1, the AST of `f` is also obtained. Likewise, other methods such as `repeat` and `map` invoked in Listing 1 and 3 are also reified and their ASTs are obtained.

Domain-specific data types and operators are implemented by deep reification as follows. Its approach is similar to deep embedding. A code snippet including domain-specific data types and operators is an embedded-DSL program. It is written in the form of lambda expression such as one passed to `run` in Listing 3. This lambda expression is passed to some method provided by the DSL library and the method extracts the ASTs of that lambda expression and the methods invoked from that lambda expression. The `run` method in Listing 3 is such a a method provided by a DSL library. Then, the extracted ASTs are translated into an optimized program, which will be executed to get the result of the DSL program written by the user. If a host language is Java, the optimized program translated from the ASTs may be a C++ program. It will be compiled dynamically by an external C++ compiler and run while the host program in Java is running. The compiled binary would communicate to the Java virtual machine through Java Native Interface (JNI) or inter-process communication such as a socket and a pipe. The latter means allows us to run the compiled binary on a remote machine with rich computational resources. Otherwise, the compiled binary can be run as a stand-alone program without any communication to the host program while it is running. In this case, the host program can be regarded as a program generator where the specification of the generated program is embedded. The generated and compiled binary will run after the host program completely finishes.

Any practical implementation of deep reification needs a means for delimiting acquisition of ASTs for methods directly/indirectly invoked by the root lambda expression. One simple means is to use naming conventions. In Java, the methods contained in system packages such as `java.lang.*` can be eliminated from the acquisition. Another approach is to let application programmers annotate methods to delimit further acquisition of ASTs. When extracted ASTs as a DSL program are translated into an efficient C++ program, some methods will be *native* methods and their ASTs will not be translated into C++ code *as is*. They will be translated, independently of their ASTs, into predefined code supplied by DSL implementation; further traversal to obtain ASTs will not be necessary. Hence, the annotation to specify such a *native* method can be used to delimit AST acquisition.

A difference from deep embedding is that a DSL program is written by borrowing the syntax of the host language while it is written in the form of method chaining in deep embedding. Deep reification also borrows parts of the semantics of a host language. In the deep embedding approach, borrowing host language features such as function calls is difficult in a DSL program. The syntax for a function call has to be explicitly implemented by the DSL. The DSL cannot reuse a host-language mechanism similar to a function call. Since deep reification traces a method-call chain and variable accesses referring to the outside of a given closure, DSL implementation can exploit that tracing functionality when implementing its equivalent language constructs such as a function call. Deep reification provides for DSL implementation a smoother connection to its host language.

The approach based on deep reification is similar to a JIT compiler but the dynamic compilation in this approach would be heavier than typical JIT compilers. Hence, a DSL program in this approach is appropriate for offloading high-performance computation when executing that computation takes a long time relatively to the compilation time. On the other hand, like the deep-embedding approach, the deep-reification approach is portable since it does not need a custom virtual machine or compiler.

Deep reification is also similar to syntactic macros such as ones found in Lisp. Syntactic macros allow programmers to extract an AST of the expression given as a macro argument. The resulting AST returned by a macro function lexically substitutes the original macro-call expression. Although a syntactic macro can extract only the AST of the expression written as the argument to the macro, deep reificaiton can extract the AST of an expression written in a different place from the place where the execution of deep reification. In Listing 3, the `map` method in line 5 and 6 gets a lambda expression referred to by the variable `f` and extracts its AST. The lambda expression does not have to necessarily be directly written in line 5 and 6. If `map` is a macro function, the lambda expression has to be written in line 5 and 6. Furthermore, deep reification allows programmers to write the following method and use it:

```
void mapmap(Func f, Vec4Array pos1, Vec4Array pos2) {
  pos2.map(f, pos1);
  pos1.map(f, pos2);
}
```

Then line 5 and 6 in Listing 3 can be replaced with the following single call:

```
mapmap(f, pos1, pos2);
```

Defining a convenience method such as `mapmap` is not possible if `map` is a macro function.

3 Bytespresso

To show our idea of deep reification, we implemented a Java library that provides deep reification in Java [2]. Our Java library named *Bytespresso* extracts an AST by bytecode decompilation. It needs to launch the Java virtual machine with the option `jdk.internal.lambda.dumpProxyClasses`. This option generates the bytecode of a dynamically generated lambda expression. For deep reification, Bytespresso reads the bytecode of a given lambda expression and decompiles it to construct an AST. It does not need source code; it only needs Java bytecode.

To support the implementation of an embedded DSL by deep reification, Bytespresso also provides a translator from ASTs to C or CUDA code. The code generated by the translation is normally compiled by an external C (or CUDA) compiler and executed in a separate process from the Java virtual machine. The generated code and the host Java code communicate with each other through a socket for portability.

Delimiting AST acquisition by deep reification, Bytespresso provides the `@Native` annotation. When a method has this annotation, the AST of the method body is not extracted by deep reification. The ASTs for the methods invoked by that method are not extracted either. Although a `@Native` method is translated into a C function by Bytespresso, the body of that C function is the argument to `@Native`. The following method is an example of `@Native` method:

```
@Native("struct timeval time; gettimeofday(&time, NULL); "
        + "return time.tv_sec * 1000000 + time.tv_usec;")
public static long time() {
  return System.nanoTime() / 1000;
}
```

The body of the C function is given as a string literal. Furthermore, Bytespresso also provides the `@Foreign` annotation for delimiting. Unlike `@Native`, the translator provided by Bytespresso does not generate a C function for a `@Foreign` method. A call to this method is translated into a call to the existing C function with the same name. For example,

```
@Foreign public static float sqrtf(float f) {
  return (float)Math.sqrt(f);
}
```

A call to this method is translated into a call to the C function `sqrt` in the standard library. The body of the method is ignored.

The translator provided by Bytespresso does not preserve the original semantics of Java. For example, the generated code does not perform array boundary checking although DSL designers, who implement their DSL compilers, can modify the translator from ASTs to C so that array boundary checking will be done. Garbage collection is optional for the generated code. If it is required, a conservative garbage collector [1] is used. Note that a program processed by Bytespresso is a DSL program. It should be a different language from Java and can provide different semantics although it has to use the same syntax as Java. The choice of which part of Java's semantics is reused by the DSL is the responsibility of the DSL designer. It should be decided to fit the aim of the DSL.

3.1 N-body simulation

We first show our small array library built with Bytespresso. It provides several classes used in Section 1.2 such as `Vec4Array` (an array of vectors in four dimensions) and `Vec3` (a vector in three dimensions). `Vec4Array` provides a method for computing sums:

```
@Inline public Vec3 sum(Func f) {
  Vec3 v = new Vec3(0, 0, 0);
  for (int i = 0; i < size; i++) {
    Vec3 vi = get(i);
    Vec3 v2 = f.apply(this, i, vi, getW(i));
    v = v.add(v2);
  }
  return v;
}
```

This is a natural Java program that can be also executed by the Java virtual machine. `get(i)` obtains the `i`-th element (with only three components) of the vector. `getW(i)` obtains the forth dimension of the `i`-th element. `f.apply` runs the lambda expression `f` with the arguments.

However, this array library is an embedded DSL. Although a program written in this DSL lexically looks like a lambda expression in Java, it is extracted from a host Java program, translated into efficient C++ code, compiled, and executed out of the Java virtual machine. The execution semantics of the DSL code is slightly different from Java's, for example, with respect to array boundary checking.

The vector elements of a `Vec4Array` object are stored in a `FloatArray2D` object. `FloatArray2D` is a class provided by Bytespresso for a two-dimensional array of `float`. The AST-to-C translation of the code related to this object is specially treated and hence every instance of `FloatArray2D` is translated into a `static` global variable in C. An instance of `FloatArray2D` has to be created with its fixed size before deep reification is executed. For example, if the size is 128 by 128, then the instance is translated into:

```
static double gvar[128][128];
```

This is a statically allocated array with a fixed size. It will help the back-end C compiler generate efficient binary.

Note that a DSL program extracted from its host Java program by Bytespresso is compiled into C and executed by a separate process. Therefore, the host Java program and the DSL program run in separate execution environments and they do not share objects or any type of variables. All the Java objects referred to by a DSL program are also extracted by Bytespresso. Then their copies are made and embedded into a generated C program as a statically allocated variable. When data

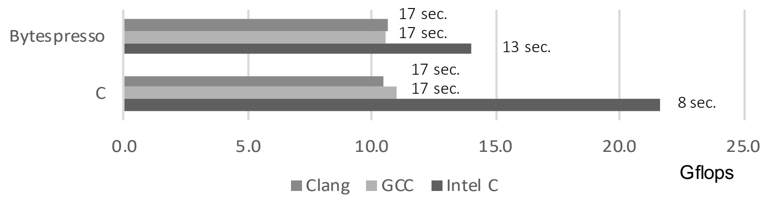

Fig. 2 The execution performance of N-body programs by Bytespresso

have to be exchanged between a host Java program and a DSL program, they have to be explicitly passed by remote method invocation. The AST-to-C translator of Bytespresso supports remote method invocation between host and DSL programs. A method annotated with `@Remote` is treated as a method that can be remotely called. Bytespresso currently supports only primitive data types and arrays as a parameter of remote method invocation.

Figure 2 shows the execution performance of our array library with Bytespresso. We ran the simple N-body simulation shown in Listing 3 on the same hardware and software as in Figure 1 in Section 1.2. The performance of Bytespresso is comparable to the C program written by hand except when the back-end C compiler is the Intel C compiler. It seems that the Intel compiler was able to effectively apply SIMD vectorization when the source code was written by hand and had a simpler structure.

The performance of Bytespresso shown in Figure 2 was achieved by aggressive inlining specified by the annotation `@Inline` for the `sum` method. Our array library supports not only single-thread execution but also OpenMP and CUDA. It provides several subclasses of `Vec4Array` and, by switching them, programmers can change which kind of program is generated from a DSL program written as a lambda expression in Java. When a subclass of `Vec4Array` for CUDA is selected, the AST-to-C translator of Bytespresso generates a CUDA program that computes the `sum` method by using a GPGPU. Therefore, a call to the `sum` method may cause dynamic method dispatch. Bytespresso aggressively inlines a method and attempts to statically resolve such dynamic dispatch. The call to the `sum` method in Listing 1 and all other calls that might have been dynamic calls were statically resolved by Bytespresso and translated into normal calls to C functions specialized for the call sites. Some calls to the specialized functions are further inlined. To help this optimization, the `final` modifier should be added to object fields if possible. Furthermore, Bytespresso provides `@Final` annotation to specify that the value of a field never changes while a DSL program is running. Since it can be updated while a host Java program is running before deep reification is performed, `@Final` is different from the `final` modifier.

```
Boundary b = new FixedEndBoundary();
b.initializer(new Initializer() { ... });
GridFloat2D grid = new CpuGridFloat2D(xsize, ysize, b);
Initializer init = new Initializer() {
  public float value(int i, int j) { return 273.15f; }
};
Kernel k = new Kernel() {
  public float newValue(Float2Array oldValue, Cursor cur,
                        int t, Reduction r) {
    float v = c0 * (cur.north(oldValue) + cur.south(oldValue))
              + c1 * (cur.east(oldValue) + cur.west(oldValue))
              + c2 * cur.self(oldValue);
    return v;
  }
};
grid.initialize(init)
    .each(Reduction.NO, 1, Predicate.FOREVER, k)
    .repeat(N, new MPIDriver(nodes));
```

Listing 4 2-dimensional 5-point stencil

3.2 Auto-parallelization

Another example is a simple framework for stencil computation. It is an embedded DSL and a program written in this DSL is translated into an efficient C program to be run. The abstraction provided by the framework hides all the details of efficient implementation. The framework users do not have to care about the implementation details.

Stencil computation is a well-known programming model useful for, for example, solving a partial differential equation. Listing 4 is a program using our framework for stencil computing. It performs five-point stencil computation in single precision. It first constructs a two-dimensional grid with a concrete boundary condition. In Listing 4, we constructs two-dimensional grid, which is a `CpuGridFloat2D` object with a `FixedEndBoundary` object. Then the `initialize` method on a grid object registers an initializer that sets each grid point to an initial value, and the `each` method registers a kernel function for stencil computation. Finally, the `repeat` method performs deep reification and generates a C program. Since a `MPIDriver` object is given to the `repeat` method in Listing 4, the generated program is an MPI program in C. It is supposed to be compiled and submitted to a job queue of supercomputer after the Java program in Listing 4 finishes.

The kernel function receives the old values of a grid (`oldValue`), and the current position (`cur`). It has to return a new value at the current position of the grid. `cur.north` obtains the old value at the upper position and `cur.self` obtains the old value at the current position. The other parameters `t` and `r` are the current time and an object for computing reduction, which is not specified in Listing 4.

```
grid.initialize(cInitPressure)
    .each(Reduction.SUM, 1, Predicate.FOREVER, new Kernel() {
      public float newValue(FloatArray3D p, Cursor cur, int t,
                             Reduction r) {
        float s0 = cur.self(a0) * cur.east(p)
                   + cur.self(a1) * cur.south(p)
                   + cur.self(a2) * cur.down(p)
                   + cur.self(b0)
                     * (cur.southeast(p) - cur.northeast(p)
                        - cur.southwest(p) + cur.northwest(p))
                   + cur.self(b1)
                     * (cur.downsouth(p) - cur.downnorth(p)
                        - cur.upsouth(p) + cur.upnorth(p))
                   + cur.self(b2)
                     * (cur.downeast(p) - cur.downwest(p)
                        - cur.upeast(p) + cur.upwest(p) )
                   + cur.self(c0) * cur.west(p)
                   + cur.self(c1) * cur.north(p)
                   + cur.self(c2) * cur.up(p)
                   + cur.self(wrk1);
        float ss = (s0 * cur.self(a3) - cur.self(p))
                   * cur.self(bnd);
        r.apply(ss * ss);
        return cur.self(p) + omega * ss;
      }
    }).repeat(N, drv);
```

Listing 5 The Himeno benchmark using our framework

This framework for stencil computation provides several components and the users can write their application program by selecting appropriate components. They can choose boundary conditions and how their program is executed, by CPU, GPGPU, or MPI. Then the framework generates a program for the given configurations.

An interesting research question is whether this framework using Bytespresso can generate a program appropriate for the given configuration, in particular, a back-end compiler. To reduce development costs, existing software tools should be reused if they are appropriate to use, and a generated program should be optimizable easily by such a back-end compiler. For example, there are automatic parallelizing compilers available.

To examine whether our framework can generate a program that such a compiler can parallelize, we rewrote the Himeno benchmark [4] to use our framework and ran it on the Fujitsu FX10 supercomputer. The back-end compiler was the Fujitsu C compiler. To exploit its automatic parallelization, a compiled program has to be a *good* one that the compiler can easily analyze and parallelize. To generate a *good* program, our framework applies function inlining to the whole kernel loop in the framework implementation. It also transforms an object used for reduction into a set of local variables. This technique is often known as object inlining. Without this

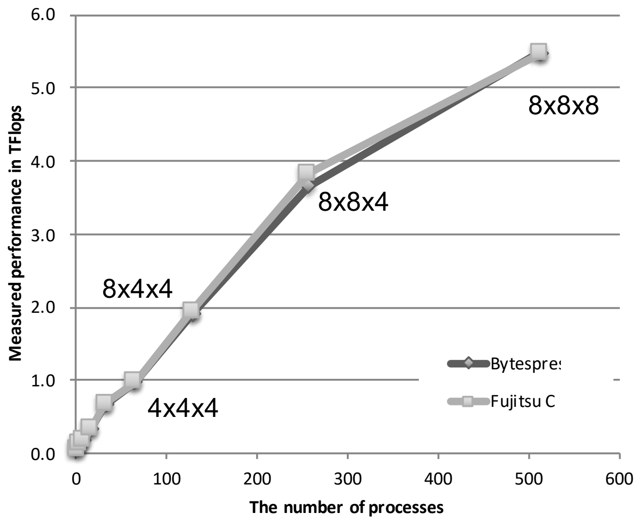

Fig. 3 Strong scale performance of the Jacobi kernel of Himeno on FX10

transformation, the back-end compiler could not parallelize the code. The framework also allocates global variables in a generated C program, which holds grid data, in a specified order on memory. Since all these optimization techniques are specific to the Fujitsu C compiler, we developed framework components for that compiler so that the framework users can choose when their target machine is the Fujitsu FX10 supercomputer.

The Himeno benchmark runs a single three-dimensional Jacobi kernel. The problem size was XL ($512 \times 512 \times 1024$). The original benchmark is written in C with MPI and each MPI process is single-threaded in single precision. The benchmark score mainly reflects the memory bandwidth. Listing 5 shows the kernel of the benchmark program. The variables `a0`, `a1`, and so forth refer to coefficient matrices, which are `FloatArray3D` objects created in the benchmark program.

Figure 3 shows the result. Bytespresso could successfully achieve comparable performance with the hand-optimized C code. Every node of the machine had 32 GByte memory and one SPARC64 IXfx (1.848 GHz) processor with 16 cores. The back-end compiler was Fujitsu C compiler 1.2.1. The compiler option -Kfast,parallel,noprefetch,ocl was given. The page size was set to 256MB.

3.3 NAS Parallel Benchmarks

We also wrote a vector matrix library built with Bytespresso. It runs on the MPI environment and hence it can process large vector and matrices. Like our small array library used for the N-body simulation in Section 3.1, it is an embedded DSL.

```
public double conj_grad(Vector x, Vector z, Matrix.Sparse a,
                  Vector p, Vector q, Vector r, double rnorm) {
  q.set(0.0);
  z.set(0.0);
  r.set(x);
  p.set(r);
  double rho = r.norm();
  for (int cgit = 1; cgit <= cgitmax; cgit++) {
    q.setToMult(a, p); // q = A * p
    double d = inner(p, q); // d = p * q
    double alpha = rho / d;
    z.setToAdd(z, alpha, p);  // z = z + alpha * p
    r.setToAdd(r, -alpha, q); // r = r - alpha * q
    double rho0 = rho;
    rho = r.norm(); // rho = r * r
    double beta = rho / rho0;
    p.setToAdd(r, beta, p); // p = r + beta * p
  }
  r.setToMult(a, z); // r = A * z
  r.setToSub(x, r); // r = x - r
  double sum = r.norm(); // sum = r * r
  return Util.sqrt(sum);
}
```

Listing 6 The CG benchmark using our framework

A program written in this DSL looks like a normal Java program but it is executed after being translated into a C program using MPI. The library users do not have to care about data distribution or exchanges through MPI. Such details are hidden by the library.

To evaluate the performance of this vector matrix library, we rewrote the CG benchmark from the NAS Parallel benchmarks 3.0 so that it will use our library. Since the library provides high-level abstraction, vectors and matrices, the CG benchmark became a largely simplified program than the original one containing a large number of `do` loops. The original program `cg-omp.f` contains 1156 lines while ours corresponding one contains only 751 lines including comments but excluding the library code. Listing 6 shows a main part of the program, the conjugate gradient routine. Since the generated program by our library runs using MPI, for example, the `setToMult` method for computing matrix-vector multiplication synchronously runs with other MPI processes. It first computes the multiplication of a sub-matrix and a sub-vector stored on local memory and then it exchanges the results among the other nodes through `MPI_Irecv`, `MPI_Send`, and `MPI_Wait`. These MPI primitives are implemented by `@Native` methods of Bytespresso.

Besides the CG benchmark, we also wrote a program equivalent to the LU benchmark of NAS parallel benchmark suites. This program does not use high-level abstraction such as a matrix; it directly uses four-dimensional double-precision arrays as the original benchmark program does. We wrote this benchmark program to

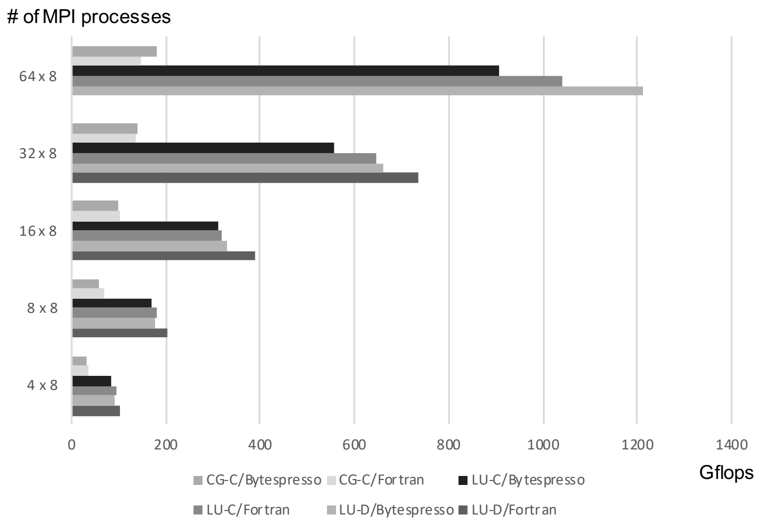

Fig. 4 NAS Parallel Benchmarks on TSUBAME 3.0

examine the basic execution performance of programs generated by the AST-to-C translator that Bytespresso provides.

The execution performance of the CG and LU benchmark programs in Figure 4. We ran them on the TSUBAME 3.0 supercomputer at Tokyo Tech. Each node of the machine has dual Intel Xeon E5-2680v4 processors with 256 Gb memory. For compilation, we used GCC 4.8.5 with the OpenMPI library. The optimization option `-Ofast` was given to the compiler. We created eight MPI processes per node and the maximum number of nodes we used was 64. The CG benchmark using Bytespresso achieved comparable performance to the original Fortran program although the slow down due to Bytespresso was not negligible for the LU benchmark.

3.4 ExaStencil

Our last example is an embedded version of the ExaSlang 4 DSL [6]. ExaSlang is an external DSL for stencil computation. It is being developed by the SPPEXA ExaStencil project. It provides stencil operators and data grids. It also provides a loop-over statement, which abstracts a (sequential or parallel) iteration over a grid.

Since ExaSlang supports multigrid methods, a set of multi-level data grids is called a field. A data grid at a particular level is specified by `@`. For example, `GradientX@finest` represents a data grid at the finest level of the `GradientX` field. A stencil operator consists of stencil coefficients.

```
Stencil SmootherStencil_u@all {
  [ 1, 0] => -1.0
  [-1, 0] => -1.0
  [ 0, 1] => -1.0
  [ 0,-1] => -1.0
  [ 0, 0] => 4.0*alpha + GradientX@current * GradientX@current
}
```

This declares a stencil operator named `SmootherStencil_u` available at all levels. The right operand of => is a coefficient at the position specified by the left operand of =>. The declaration above defines the following stencil:

$$\begin{pmatrix} & -1 & \\ -1 & 4\alpha + I_x^2 & -1 \\ & -1 & \end{pmatrix}$$

Here, I_x is the element of `GradientX` at the current position. The stencil can be used in a function:

```
Function Smoother@all ( ) : Unit {
  // omitted
  loop over Flow_u@current {
    Flow_u[next]@current = Flow_u[active]@current
      + ((1.0 / diag(SmootherStencil_u@current))
         * (RHS_u@current
            - SmootherStencil_u@current * Flow_u[active]@current
            - GradientX@current * GradientY@current * Flow_v[
    active]@current))
  }
  // omitted
}
```

This `Smoother` function is available at all levels and it iterates the loop body over the current level of the `Flow_u` field. In the body, the `SmootherStencil_u` for the current level is applied to the current level of (the `active` slot of) the `Flow_u` field.

We implemented an embedded version of this DSL by using Bytespresso. In the embedded version, stencil operators and fields are defined as Java objects. For example, the `SmootherStencil_u` shown above is written as follows:

```
final Stencil smootherStencil_u
  = new StencilBuilder()
    .add(1, 0, -1.0)
    .add(-1, 0, -1.0)
    .add(0, 1, -1.0)
    .add(0, -1, -1.0)
    .add(0, 0, (current, x, y) ->
      4.0 * alpha + gradientX.at(current).get(x, y)
                  * gradientX.at(current).get(x, y))
    .build();
```

To obtain better performance, an instance of `CustomStencil` should be used:

```
final CustomStencil smootherStencil_u = new CustomStencil() {
  public double calc(final LayeredNodeField layeredField,
```

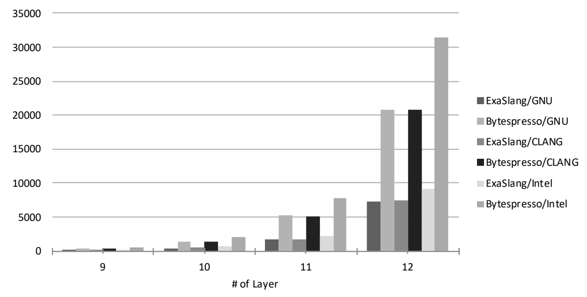

Fig. 5 Execution time of ExaSlang programs (msec.)

```
                    final int current, final NormalIndex x,
                    final NormalIndex y) {
    return -1.0 * (layeredField.get(current, x, 1, y, 0)
                  + layeredField.get(current, x, -1, y, 0)
                  + layeredField.get(current, x, 0, y, 1)
                  + layeredField.get(current, x, 0, y, -1))
                  + (4.0 * alpha + gradientX.get(current, x, y)
                                 * gradientX.get(current, x, y))
                    * layeredField.get(current, x, 0, y, 0);
  }
};
```

This object directly represents the expression computed by a stencil operator. Likewise, the `Smoother` function is written in our DSL as follows:

```
@Inline public void smoother(int current) {
  // omitted
  flow_u.next(current).loopOver(current, (c, x, y) ->
    flow_u.active(c).get(c, x, y)
    + smootherStencilDiagInv_u.calc(
        ((rhs_u.get(c, x, y)
         - smootherStencil_u.calc(flow_u.active(c), c, x, y))
         - gradientX.get(c, x, y) * gradientY.get(c, x, y)
           * flow_v.active(c).get(c, x, y)),
        c, x, y));
  // omitted
}
```

This function takes a parameter specifying a level. Its expression is more verbose than ExaSlang but keeps the same abstractions.

We compared the execution performance of programs that compute two-dimensional optical flow by the multigrid method. One program was written in the original ExaSlang language while the other was written in our embedded DSL. Both are single-threaded. We compiled the two programs by GCC 5.4.0, Intel C compiler 17.0.1, and Clang 5.0.0 with `-O3`, and ran them on the machine with Intel Xeon E5-2637v3. We examined with different numbers of layers for the multigrid method. Figure 5 shows the execution time of the kernel part of the two programs. The program written in

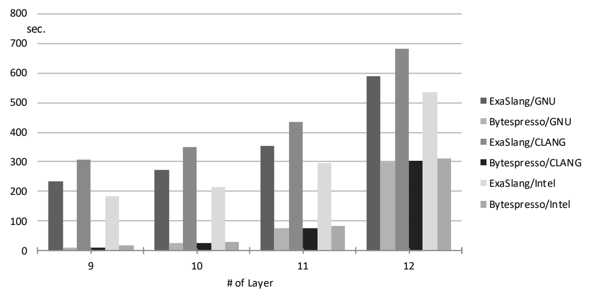

Fig. 6 Compile and execution time of ExaSlang programs (sec.)

our embedded DSL was only three times slower than the program in the original ExaSlang language. Figure 6 shows the sum of the compilation time and the execution time of the two programs. Since the compilation by the current ExaSlang compiler is slow, our embedded DSL was rather faster.

4 Summary

This chapter presented *Bytespresso*, a prototype system to use our deep-reification technique, in Java. Bytespresso allows DSL developers to easily implement efficient DSLs embedded in Java. The embedded DSL code is dynamically extracted and offloaded from the Java virtual machine onto native hardware. The DSL developers can exploit an existing tool chain, including an external optimizing compiler, when offloading DSL code. Since the syntax of the DSLs are borrowed from Java's and a few language mechanisms in Java, such as a method call, are reused, the development costs of the DSLs can be reduced.

Acknowledgements I would like to thank Maximilian Scherr and Toshiyuki Takahashi for their various contributions to this work. This is partly supported by JST/DFG SPPEXA ExaStencil project. We deeply thank Christian Lengauer, Sebastian Kuckuk, Christian Schmitt, Matthias Bolten, Frank Hannig, and Harald Köstler. We also thank Shuichi Chiba at Fujitsu Ltd. for the experiment on FX10.

References

1. Boehm, H.J., Weiser, M.: Garbage collection in an uncooperative environment. Software Practice and Experience **18**(9), 807–820 (1988)
2. Chiba, S., Zhuang, Y., Scherr, M.: Deeply reifying running code for constructing a domain-specific language. In: Proc. of the 13th Int'l Conf. on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, PPPJ '16, pp. 1:1–1:12. ACM (2016)

3. Fowler, M.: Fluentinterface. https://www.martinfowler.com/bliki/FluentInterface.html (2005)
4. Himeno, R.: Himeno benchmark. `http://accc.riken.jp/2444.htm` (2001)
5. Jovanovic, V., Shaikhha, A., Stucki, S., Nikolaev, V., Koch, C., Odersky, M.: Yin-yang: Concealing the deep embedding of dsls. In: Proc. of the 2014 Int'l Conf. on Generative Programming: Concepts and Experiences, GPCE 2014, pp. 73–82. ACM (2014)
6. Kuckuk, S., Haase, G., Vasco, D.A., Köstler, H.: Towards generating efficient flow solvers with the ExaStencils approach. Concurrency and Computation: Practice and Experience **29**(17), 4062:1–4062:17 (2017)
7. Nakamaru, T., Ichikawa, K., Yamazaki, T., Chiba, S.: Silverchain: A fluent api generator. In: Proc. of the 16th ACM SIGPLAN Int'l Conf. on Generative Programming: Concepts and Experiences, GPCE 2017, pp. 199–211. ACM (2017)
8. .NET Foundation: The .net compiler platform "roslyn". https://github.com/dotnet/roslyn (2014)
9. Nyland, L., Harris, M., Prins, J.: Fast n-body simulation with CUDA. In: H. Nguyen (ed.) GPU Gems 3, chap. 31, pp. 677–695. Addison-Wesley (2007)
10. Quinlan, D., Schordan, M., Miller, B., Kowarschik, M.: Parallel object-oriented framework optimization. Concurrency and Computation: Practice and Experience **16**(2-3), 293–302 (2004)
11. Rompf, T., Amin, N., Moors, A., Haller, P., Odersky, M.: Scala-Virtualized: linguistic reuse for deep embeddings. Higher-Order and Symbolic Computation **25**(1), 165–207 (2012)
12. Rompf, T., Odersky, M.: Lightweight Modular Staging: A pragmatic approach to runtime code generation and compiled DSLs. In: Proc. of the Ninth Int'l Conf. on Generative Programming and Component Engineering, GPCE '10, pp. 127–136. ACM (2010)
13. Rompf, T., Sujeeth, A.K., Brown, K.J., Lee, H., Chafi, H., Olukotun, K.: Surgical precision JIT compilers. In: Proc. of the 35th ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI '14, pp. 41–52. ACM (2014)
14. The Eclipse Foundation: Eclipse IDE. http://www.eclipse.org (2001)