A Framework for Multiplatform HPC Applications

Masayuki Ioki
Tokyo Institute of Technology
www.csg.ci.i.u-tokyo.ac.jp

Shigeru Chiba
The University of Tokyo
www.csg.ci.i.u-tokyo.ac.jp

Abstract

This paper proposes a framework for building multi-platform applications in Java for High Performance Computing (HPC). It allows HPC developers to write their programs in Java but dynamically translate part of the programs into C programs using MPI or CUDA so that the translated code can be executed on multi-platforms. The source of the translated code is written in Java but with extensions for MPI and CUDA supports. The implementations for different platforms are switched by object-oriented mechanisms such as dynamic method dispatch. However, object oriented mechanisms are major sources of execution overheads. To reduce these overheads, the proposed framework requires that the translated code is subject to our coding rules, in which object-oriented mechanisms are available only in limited contexts. All objects except arrays must be immutable and most class types must be leaf classes. Only the types of method parameters and instance fields can be non-leaf class types. These restrictions allow our framework to statically determine object types during the code translation while they still enable building a practical class library for HPC with respect to customizability. This paper presents examples of the class libraries built on top of our framework. Their performance is sometime better than the performance of the programs written in C++ with equivalent class libraries since C++ is a general-purpose language and thus its expressiveness does not perfectly fit our problem domain, HPC applications.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors

General Terms Languages, Performance

Keywords Java, Software Productivity, MPI, CUDA

1. Introduction

In the High Performance Computing (HPC) domain today, developers need to consider not only execution performance but also software productivity. Modern super computers have heterogeneous architectures and thus, for portability, HPC programs should support various platforms such as GPUs.

To modularly implement HPC programs and improve developers' productivity, a classic approach is to divide the programs into platform-neutral code and the code depending on a particular platform. It allows developers to build a program for a specific platform by only switching the platform-dependent code while reusing the platform-neutral code as it is. However, this approach has been less appealing in the HPC domain since it often heavily uses modularization mechanisms of the programming language, for example, classes and dynamic method dispatch in object-orientation. They often imply non-negligible performance costs, which do not meet the performance requirements in HPC.

This paper presents a framework for developing a class library to build multi-platform HPC applications with acceptable performance overheads. The applications on top of our framework are written in Java with simple extensions but the kernel code can be dynamically translated into optimized C (or CUDA) code for a specific platform such as MPI and GPU. During the translation, the code is highly optimized by using runtime type information and thus reduce performance overheads due to the modularization by object-orientation in Java. The HPC applications on our framework achieve comparable execution performance to the equivalent C++ code using template meta-programming. The optimization performed by our framework consists of devirtualization [4, 8] and object inlining [5, 6, 18]. These are classic optimization techniques, which the just-in-time compiler of most commercial Java virtual machines adopt, but our framework aggressively applies them to all dynamic method dispatches and object references within the translated Java code. This paper presents the effects of this optimization by showing the results of our experiments on the TSUBAME 2.0 super computer [17].

The translated kernel code is written in plain Java although the language provides simple extensions to support CUDA and/or MPI programming. It also provides a foreign function interface to directly invoke library functions implemented in C. It does not have any syntactic extensions; the programs can be compiled by the standard Java compiler. However, to enable the aggressive optimization, the translated code cannot exploit the full set of language features of Java. It must be written to meet the coding rules of our framework. For example, the use of dynamic method dispatch is restricted within specific contexts. These coding rules still enable building a practical class libraries. This paper presents a stencil-computation application built on top of our framework.

Since the translated code is written in plain Java but in a restricted coding style, our approach is similar to Delite [2], the approach using a domain specific language (DSL) for describing the kernel code to obtain optimized performance. Our contribution is that we show a DSL for building a class library where platform-dependent code is clearly separated and switchable with small runtime costs. Our DSL is not for describing application logic with domain-specific abstraction. An advantage of our approach is that the developers have only to learn the coding restrictions for their familiar language. Our approach also has an advantage against the approach based on template meta-programming in C++ since the optimization is automated by our framework. Furthermore, the tem-

PMAM 2014, February 15, 2014, Orlando, FL, U.S.A..

ACM 978-1-4503-2655-1/14/02. . . $15.00.
http://dx.doi.org/10.1145/2560683.2560693

plate meta-programming in C++ has a problem with respect to the expressiveness. Our approach is applicable not only to Java. We chose Java since a Java program is relatively simple and easy to analyze for optimization.

The rest of this paper is organized as follows. Section 2 shows a motivating example of multi-platform application in HPC. Section 3 presents our framework and the restrictions applied to the translated code. In Section 4, we present the results of our experiments. Related work is discussed in Section 5. Finally, in Section 6, we conclude this paper.

2. Stencil computations

Writing class libraries, where a different feature (or concern) is separately implemented in a different class, has been widely accepted as an approach to improve programmers' productivity. Figure 1 shows a feature model [12] that an application for stencil computation should implement. Feature modeling is widely used as a tool for design, especially when designing software product lines [14]. A feature represents a functionality that a user programmer can select. For example, Dimension has three sub-features, each of which specifies the dimension of simulation space. Parallelism is an optional feature for specifying a platform; if a problem size is small, a programmer will not select this feature or its sub-features such as GPU. A user programmer will select those features and construct an application.

Figure 2 shows the class diagram of the class library for the stencil applications implementing the feature model in Figure 1. The main components are StencilRunner, PhysQuantity, and StencilSolver. The StencilRunner class implements how to run a program in parallel. If the StencilGPU4DblB_MPI class is selected, the program is run with double buffering on multiple nodes with GPUs communicating by MPI. If the StencilCPU4DblBuffer class is selected, the program is sequentially run with double buffering; it does not use GPUs or MPI. The PhysQuantity class implements a physical-model feature. The StencilSolver is a class implementing a kernel operation applied to every grid element. A programmer only writes a subclass of this class. An example is the Dif1DSolver class in Listing 1. Its solve method implements a solver of a one-dimensional diffusion equation. Note that it is independent of other features such as Parallelism.

These classes are selected to compose an application. Listing 2 is an example of the main program. The parameter n specifies the grid size. The program instantiates several classes selected from the provided classes and user-written classes and finally creates a runner object of the StencilRunner type. The stencil computation starts when the invoke method is called on runner. This is a typical code structure of the main program with class libraries. It instantiates several component classes and combines the instances to compose a larger object representing the application, which actually starts when the start method such as invoke is called on this composed object. Note that in most cases the composed object never changes during runtime since it mainly represents the application logic but it does not represent the data processed in the application.

Although our class library shown above improves programmers' productivity, it involves two problems. The first one is performance penalties due to object orientation, which is indispensable for building a class library. The abstraction and modularity by object orientation are not free of charge. Figure 3 shows a comparison among three implementations of simulation of three-dimensional diffusion equation. *Java* refers to the simulation program using our class library written in Java. *C++* refers to the program naively written in C++ to be equivalent to our class library in Java. Finally, *C* refers to the program written in C without a class library; it is optimized by hand. All the three programs ran by a single thread without a GPU or MPI. The figure illustrates that *Java* and *C++* are

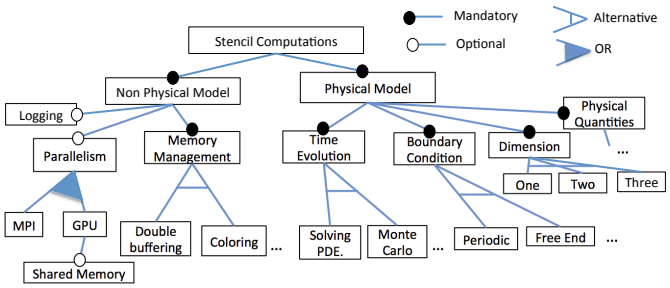

Figure 1. A feature model for stencil computation

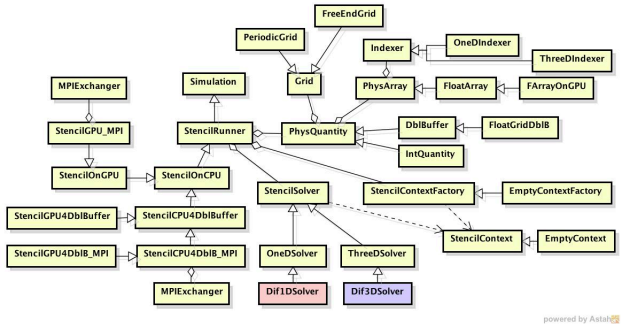

Figure 2. The class diagram of a stencil-computation class library

Listing 1. One-dimensional diffusion solver

```
class Dif1DSolver
  extends OneDSolver<ScalarFloat, FloatGridDblB,
                     EmptyContext> { ...
  @Override public ScalarFloat solve(
      ScalarFloat left, ScalarFloat right,
      ScalarFloat self,
      FloatGridDblB q, EmptyContext context){
    float value = a * (left.val() + right.val())
                                  + b * self.val();
    return new ScalarFloat(value);
  }}
```

more than ten times slower than *C*. It reveals that the main source of the performance overhead is not Java but object orientation.

The other problem is that Java programs run on the Java virtual machine and hence it cannot directly exploit the capability of the underlying hardware. A typical supercomputer today has dedicated hardware for inter-node communication and provides a library, such as MPI, for exploiting it. However, such a library is usually available only in Fortran and C. Although Fujitsu FX10 supercomputer provides a Java binding to MPI (a Java library to call the MPI library in C), its execution performance is not comparable to direct accesses in C since runtime costs of bridging between Java and C are not negligible. This problem is also serious when GPUs are available. Since the current Java virtual machine does not support GPUs, a software solution is needed to enable a Java program to execute some computation on GPUs.

3. WootinJ

To enable multiplatform HPC applications in Java, we have developed a framework named *WootinJ*. This framework allows developers to build a class library that are written in Java but can partly run on GPUs and/or multiple nodes communicating through MPI.

Listing 2. Run the solver with MPI and GPU

```
public static void mpiGpuRun(int n, int mpiSize){
  SimulationConfig conf; ...
  StencilSolver solver = new Dif1DSolver();
  DblBuffer buf = new FloatGridDblB(
    new FArrayOnGPU(n, new OneDIndexer()) ,
    new FArrayOnGPU(n, new OneDIndexer()));
  SizeDTO mpiSizeDto = new SizeDTO(mpiSize);

  StencilRunner runner
    = new StencilGPU4DblB_MPI(conf, solver,
      buf, mpiSizeDto);

  runner.invoke();
}
```

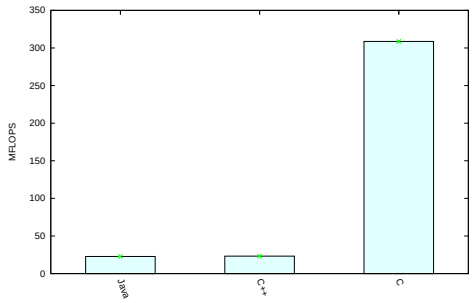

Figure 3. The performance of the 3D diffusion ($128 \times 128 \times 128$) in Java, C++, and C

In WootinJ, a selected Java method can be dynamically — *just in time (JIT)* — translated into C or CUDA code. Since WootinJ processes Java bytecode, the source code is not necessary for this translation. Then the translated code is compiled by a C/CUDA compiler and it is invoked through JNI (Java Native Interface) with the given arguments on the specified hardware such as GPUs. If MPI is used, the translated code is invoked by the mpirun command.

Multiplatform:

A method translated by WootinJ is written in Java but with simple extensions. If a method is annotated with @Global, it is translated into a *global* function in CUDA. A call to a @Global method in Java is considered as a call to a global function in CUDA, that is, starting multi-threaded execution on a GPU. For MPI functions, WootinJ provides the MPI class in Java. Since this class is not a wrapper class that accesses the MPI functions in C through JNI, no runtime penalties are involved in this class. A call in Java to a method in the MPI class is translated by WootinJ into a direct call in C to the corresponding MPI function. Note that making other C functions available in Java as well as MPI functions is easy for WootinJ users; WootinJ provides a mechanism for programmers to define a method call that are translated into a direct call to the corresponding C function.

Optimization:

To overcome overheads due to object orientation, WootinJ aggressively applies devirtualization [4, 8] and object inlining [5, 6, 18] to the translated Java code. WootinJ translates *all* occurrences of dynamic method dispatch into static function calls in C so that a runtime penalty due to method calls will be eliminated (devirtualization). It also transforms an object into a set of primitive-type variables and then translates *all* field accesses into simple accesses to the local variables (object inlining). This requires that the JIT compiler of WootinJ can statically determine the actual type of the target object at every object reference. Since statically determining all the types is not possible in general, the Java code translated by WootinJ must be subject to our coding rules. This allows the JIT compiler to perform devirtualization and object inlining for every object reference by simple program analysis. Note that only the Java code dynamically translated by WootinJ into C/CUDA code is subject to the coding rules. The rest of the program does not have to follow the rules.

In the code subject to the coding rules, objects are mostly immutable; we call this property *semi-immutable* as shown later. Furthermore, all class types appearing in the code must be leaf-class types except method parameters and object fields. We will later mention this property named *strict-final*. These properties obviously make static analysis easier and enable devirtualizing all object references. On the other hand, they considerably restrict the design of class libraries built on WootinJ. According to our observation shown in the previous section, however, this restriction is practical in our problem domain since instances of library classes do not change after they are initialized and the start method such as invoke is called. We will later show that the class library in the previous section can be built on WootinJ.

3.1 Client View

To illustrate how to build a class library on WootinJ, we present a simple example, which is a class library for the *one-point* stencil computation using GPU and MPI. Listing 3 shows a program written by the user of the class library. It includes three classes: PhysDataGen for generating a data grid, PhysSolver defining a kernel operation applied to every grid element, and Main for the main method. The former two classes implement the interfaces provided by the class library, Generator and Solver.

The main method first makes a Stencil object. Stencil is the main class of the class library and represents a stencil application. It has the run method, which first generates a data grid by calling the make method on the Generator object. Then it iterates applying the kernel operation implemented by the Solver object to every grid element. The arguments to the run method specify the grid size and the number of the iteration.

Unlike an application written with a typical class library, the main method does not directly call the run method on the Stencil object. Rather, it calls the jit4mpi method in WootinJ (or the jit method unless the program uses MPI) to translate the bodies of the run method and the methods called from run into the C/CUDA code using MPI. Since the run method indirectly calls methods on the PhysDataGen and PhysSolver objects, these methods are also translated. The translated code is then compiled by the CUDA/MPI compiler. The annotation @WootinJ indicates that the class definition satisfies the coding rules of WootinJ and hence it can be translated. Note that the arguments passed to the run method are also given to jit4mpi. They are recorded and used for optimization during the translation. After calling jit4mpi, the main method calls set4MPI to configure the execution environment for MPI and then it calls invoke to invoke the mpirun command, which executes the translated code with the arguments recorded in jit4mpi.

Listing 4 presents the class library's classes including the StencilOnGpuAndMPI class. Note that they are written by not library users but library developers, who have deeper knowledge about GPU and MPI. Their implementations are hidden from the library users, who write Listing 3. Hence those library classes are low-level code specific to GPU and MPI and they are written in Java with extensions for supporting GPU and MPI. For example, a method with the @Global annotation is translated into a C function with __global__, which is a modifier introduced by CUDA. Since a global

Listing 3. A program written by the class library user

```
@WootinJ class PhysDataGen implements Generator{
  ...
  @Override float[] make(int length, int seed) { ... }}

@WootinJ class PhysSolver implements Solver{ ...
  @Override float solve(float self, int index){ ... }}

class Main{
  public static void main(String... args){
    int length, updateCnt;
    ...
    Generator generator = new PhysDataGen();
    Solver solver = new PhysSolver();
    Stencil stencil =
          new StencilOnGpuAndMPI(generator, solver);

    JitCode code =
      WootinJ.jit4mpi(stencil, "run", length, updateCnt);
    code.set4MPI(128, "./nodeList");
    code.invoke();
    ...  }}
```

Listing 4. The classes in the class library

```
@WootinJ class StencilOnGpuAndMPI extends Stencil {
  Solver solver;
  Generator generator;
  CUDA cuda = new CUDA();
  ...
  public StencilOnGpuAndMPI(Solver _solver,
                            Generator _generator) {
    ... }

  void run(int length, int updateCnt){
    int rank = MPI.rank();
    float[] array = generator.make(length, rank);
    float[] arrayOnGPU = cuda.copyToGPU(array, length);

    dim3 block = new dim3(length);
    CudaConfig conf = new CudaConfig(block);
    for(int i=0; i < updateCnt; i++)
      runGPU(conf, arrayOnGPU);
      ... }

  @Global void runGPU(CudaConfig conf, float[] array){
    int x = cuda.threadIdx.x;
    array[x] = solver.solve(array[x],x); }}

@WootinJ interface Solver {
    float solve(float self, int index); }
@WootinJ interface Generator {
    float[] make(int length, int seed); }
```

function in CUDA takes special arguments surrounded by <<< >>>, the method annotated with @Global instead takes a CudaConfig object as the first argument. It specifies the special arguments. WootinJ also provides the @Shared annotation for specifying that the annotated field represents shared memory. dim3 is another class provided by WootinJ. It represents the dim3 type in CUDA. CUDA and MPI are also classes provided by WootinJ. They have utility methods to access library functions of CUDA or MPI.

The run method in Listing 4 is straightforward Java representation of the one-point stencil program written in CUDA and using MPI. The only object-orientation found is the dynamic method dispatch from the runGPU method to the solve method on solver. This method dispatch is an essence of component composition provided by the class library. Since it is devirtualized during the translation, the code generated by WootinJ is a typical CUDA/MPI program shown in Listing 5. Now the call to runGPU is translated into a special call to a global function with <<< >>>. It starts a number of threads computing the runGPU function on a GPU in parallel.

The translated code is executed in a separate memory space. If a part of the translated code is executed on a GPU, it is executed in another memory space; it cannot transparently access the data used by the rest of the translated code running on a CPU or the data used by the Java program not translated. When the translated code starts running, the arguments are deeply copied from the Java memory space to the memory space used by the translated code. The arguments to the GPU code are also deeply copied when it is invoked. The modified data are not copied back to the original memory space when the translated code terminates. We chose this design since the abstraction of transparent memory access should be provided by not WootinJ but a class library on top of WootinJ.

Listing 5. The CUDA/MPI code generated after the translation

```
float* make(int length, int seed){ ... }

__device__
float solve(float self, int index) { ... }

__global__ void runGPU(int* array){
  int x = threadIdx.x;
  array[x] = solve(array[x], x);
}

void run(int length, int updateCnt){
  int rank;
  MPI_rank(&rank);
  float* array = make(length, rank);
  float* arrayOnGPU;
  cudaMemcpy(arrayOnGPU, array,
    sizeof(float)*length, cudaMemcpyHostToDevice)

  dim3 block(length);
  for(int i=0; i < updateCnt; i++)
      runGPU<<<1, block>>>(arrayOnGPU);
  ...
}

int main(int argc, char* argv[]){
  MPI_Init(&argc, &argv);
  int length, updateCnt;
  ...
  run(length, updateCnt);
  MPI_Finalize(); }
```

3.2 Code Translation

We next show an overview of the code translation by WootinJ. The code translated by WootinJ has to follow the coding rules.

Definitions

Before presenting the coding rules, we introduce two properties: *strict-final* and *semi-immutable*. First, we define strict-final, which means a leaf-class type with fields of leaf-class types.

A type T is *strict-final* if either:

1. T is a primitive type,
2. T is an array type and the element type is strict-final, or
3. T is a class type, it is a final class (*i.e.* no subclasses), and all the fields of the class and its super classes are of strict-final types.

The last condition means that T is a leaf class that has only fields of leaf-class types. Thus, if a variable is of a strict-final type, all the objects reachable from the variable are also of a strict-final type.

Next, a type S is *semi-immutable* if either:

1. S is a primitive type,

2. S is an array type and the element type is semi-immutable and strict-final, or
3. S is a class type satisfying all the following preconditions:
   (a) All the fields of the class are of semi-immutable types.
   (b) All super classes of S are of semi-immutable types. The Object class is a semi-immutable type.
   (c) Unless a field of the class is of an array type, then it is a constant. Once the field is initialized in a constructor, the value of the field is never modified. A subclass constructor can modify the value of the field declared and initialized in the super class (thus, the field is not necessarily a final field).
   (d) In constructors, conditional branches, such as if, for, and the conditional operator (?:), method calls or this variable are not available.
   (e) S is not a recursive type.

An instance of a semi-immutable class is immutable except array-type fields. All the fields reachable from the instance are immutable except arrays. Only the array-type fields can be modified during runtime to change the array object that the field refers to. An array element is also modifiable. Another property of being semi-immutable is that the actual types of any objects reachable from the instance are statically determined if the actual types of the constructor arguments are given since the constructor does not include conditional branches. Here, giving the actual type of the argument includes giving the actual types of all the objects reachable from the argument.

Coding rules

The translated code by WootinJ consists of methods annotated with @WootinJ. It must be subject to the following coding rules, which basically require that all the objects are immutable and most types are leaf-class types:

1. All the types appearing in the code are semi-immutable.
2. All the types appearing in the code are also strict-final except method parameters and field types. Local-variable types, return types, and cast types are strict-final.
3. All method parameters are constant (*i.e.* final).
4. A type parameter T has the upper bound S and all the direct subclasses of S are strict-final and semi-immutable. A type argument given for T must not be S; it must be a subclass of S. A wild card is not used.
5. All static fields are constant (*i.e.* final) and not an array type.
6. Recursive calls are not used.
7. The conditional operator (?:) or the reference equality operators (== and !=) are not used.
8. Exception handling, reflection, multithreading, native access (for example IO), .class, instanceof, or null literals are not used.

These rules allow WootinJ to perform devirtualization and object inlining for every object reference by simple program analysis. In compensation, these rules significantly restrict developers' coding style although they are applied to only the Java code dynamically translated by WootinJ.

Since an array object is major storage of data in HPC, our coding rules do not require that an array object is immutable; the value of an array element can be updated at any time if the element type is a primitive type.

Although most types must be leaf-class types (*i.e.* strict-final), the types of fields and method parameters can be non leaf-class types. This enables a class library user to customize a component by giving an instance of a different class. However, the fields and the method parameters must be constant and the objects that they refer to must be immutable. It is prohibited to assign a new value to a method parameter. Thus, if the initial value of a method parameter of class type X is an instance of a subclass of X, the value of the method parameter is never updated to an instance of a different subclass of X.

3.3 Translation from Java to C

The translation from Java bytecode into C/CUDA code using MPI is fairly straightforward except a few kinds of language constructs. We below present a brief overview of the translation.

Class declaration. A class declaration in Java is simply removed. After the translation, an instance of the class is represented by a set of local variables of primitive types. Each variable corresponds to a field. A static field is translated into a set of global variables. These global variables are initialized by copying the values of the static field when the translated code starts running. Note that the coding rules require that the static field is constant.

Method declaration. When WootinJ translates given Java code, it receives not only the entry method but also the arguments passed to the entry method. Assuming that the actual types of the given arguments are the formal parameter types of the method, WootinJ translates the declaration of the entry method into a function declaration in C. Note that @Global annotating a method declaration is translated into the __global__ modifier in CUDA.

Method calls. When a method call is encountered during the translation of a method body, WootinJ statically determines the actual type of the receiver object. This can be done by a simple program analysis. First, since the types of local variables and the return types of methods are strict-final, determining the actual type of the expression computing the receiver object is straightforward. Note that the use of the conditional operator is prohibited. Hence, the following code never appears:

```
(i > 0 ? new IntGrid() : new FloatGrid()).size()
```

The types of fields and method parameters are exceptions; they can be non strict-final. However, the coding rules require that the method parameters are constants. Moreover, all the objects are of semi-immutable types. Thus, the actual types of the fields reachable from the method parameters can be also statically determined if the actual types of the method arguments (and all the objects reachable from the method parameters) are given. For example,

```
parameter.grid.size()
local.grid.size()
new Stencil(new FloatGrid()).grid.size()
```

Here, parameter is a method parameter and local is a local variable. The actual type of the grid field is statically determined for all the expressions shown above. Since the type of the local variable is strict-final, the actual type of the object that local refers to is equivalent to the variable type. If Stencil is a semi-immutable type, the actual type of grid is statically determined by analyzing the constructor body if the actual type of the constructor argument is given.

After determining the actual type of the receiver object, WootinJ selects the method implementation for that type and translates it into a function declaration in C. The method call is translated into a call to this function. When translating the method implementation, WootinJ specializes the generated function declaration for the actual types of the arguments. Note that the actual types of the arguments can be statically determined in the same algorithm as the receiver type. Thus, WootinJ may generate multiple function dec-

larations from a single method implementation for different types of the arguments.

In CUDA, a global method can call only *device* methods. When translating a method implementation called from a global function, WootinJ adds the __device__ modifier to the function declaration generated for the method implementation.

Local variables and method parameters. In Java, a variable of class type holds a reference to an object in heap memory. Since the actual type of an object is statically determined as shown above, after the translation into C, an object is *inlined*; it is implemented by a set of local variables on stack. The objects reachable from it are also inlined. A variable does not hold a reference but directly represents the object. If the value of a variable is assigned to another variable or if it is passed to a method, (a copy of) it is stored or passed instead of a reference. This does not break the semantics of the code since all the objects are semi-immutable. The this variable is treated in the same manner as a local variable.

Constructors. If a new object is instantiated within the translated code, the constructor call is inlined in the body of the generated function in C.

Array. An array type is also an exception. An array object is allocated in heap memory and an array-type variable holds a reference to it. If an array element is a not-array object, the element directly holds the object as a value. It does not hold a reference to the object. Since the coding rules require that the element type of an array object is strict-final, the size of the array elements can be statically computed.

Other issues. To support the shared memory in CUDA, if a field of an array type has the @Shared annotation, it is translated into a variable with __shared__. Garbage collection or array boundary checks are not provided by the code after the translation. They are developers' responsibility. For garbage collection, the free function provided by WootinJ must be explicitly called.

4. Evaluation

This section presents the results of our performance measurement of applications written with WootinJ. They were executed on the TSUBAME 2.0[17] super computer at Tokyo Institute of Technology. Every node of the computer is equipped with two Intel Xeon 2.9 GHz CPUs (with 6 cores) and three NVIDIA M2050 GPUs with 54GB main memory and 3GB GPU memory. The Java virtual machine is IBM J9 VM (Java version 1.6.0). The C compiler is *icc* (version 13.0.0.079).

We wrote two class libraries on top of WootinJ. The first one is a class library for stencil computation that has been already shown in Section 2. The other is a small class library for computing matrix multiplication. For comparison, we wrote one program in C and three programs in C++ for each class library. We below call them *C*, *C++*, *Template*, and *Template w/o virt.*. *C*, the program written in C, implements the same algorithm as the WootinJ equivalence but without considering code reuse or modularity of components. *C++* is a program in C++ providing the same abstraction as WootinJ. It naively uses virtual functions for dynamic method dispatch. *Template* is another C++ program in that dynamic method dispatch is devirtualized by template meta-programming. All occurrences of the -> operator are replaced by the . (dot) operator. Finally, *Template w/o virt.* is a C++ program in that virtual functions are not used at all. To emulate method overriding, all the methods in the super classes have been manually copied into the subclass body. This makes a significant impact upon the code reuse and modularity.

The C++ programs have other differences from the WootinJ program. First, since virtual function calls by -> operator in CUDA on GPUs were unstable in our environment, we did not use virtual function calls by -> operator in the kernel functions for CUDA.

Listing 6. Mutual type reference

```
class MPIThread implements OuterThread{
  OuterThreadBody body;
  void start(Matrix a, ...){
    ...
    body.run(this, a, ...);
  } ... }

class FoxAlgorithm implements OuterThreadBody{
  void run(OuterThread thread, Matrix a, ...){
    ... }}
```

Program	Compiler options
C++	-ipo -O3 -rcd -i-static -parallel
Template	-ipo -O3 -rcd -i-static -xHost -parallel
Template w/o virt.	-ip -O3 -rcd -i-static -xHost -parallel
C	-ipo -O3 -rcd -xHost -parallel

Table 1. The compiler options for the 3D diffusion equation

Hence, with respect to the GPU code, the C++ programs provide modularity inferior to the WootinJ program.

Furthermore, if the program includes mutually referential classes, we could not rewrite these classes into template classes. In Listing 6, the MPIThread class has a reference to an OuterThreadBody object. To devirtualize a call on the OuterThreadBody object, we must parameterize the actual type of this object and modify the MPIThread class into a template class such as MPIThread<FoxAlgorithm>. However, for devirtualization, the FoxAlgorithm class must be also a template class taking the actual type of the OuterThread object passed to the run method. In our example, this actual type is MPIThread<>, which mutually refers to FoxAlgorithm in the template argument. In our experiment, since we could not naturally rewrite such classes into template classes, these classes are not fully composable. We abandoned code reuse and wrote classes specialized for a specific combination of components, for example, the FoxAlgorithmForMPIThread, in which all the occurrences of the type name OuterThread are replaced with the concrete class MPIThread. Therefore, the C++ programs using templates are inferior to the WootinJ program with respect to code reuse and modularity.

4.1 Three-dimensional diffusion equation

To evaluate our class library for stencil computation, we wrote a solver of three-dimensional diffusion equation. The main component of the solver program is a Dif3DSolver class, which implements the three-dimensional diffusion equation we solve. It is a subclass of ThreeDSolver from the class library. The program also uses the FloatGridDblB (a float array with double buffering) class from the class library. For switching execution hardware, we selected an appropriate subclass of StencilRunner from the class library. All the programs were compiled with the options listed in Table 1.

Figure 4 shows the weak scalability of the solvers running on multiple nodes communicating through MPI. Only one CPU/thread was used per node and no GPUs were used. The problem size was $128 \times 128 \times 128$ per node. We also compared the strong scalability of the solvers between *C* and *WootinJ*. The problem size was $128 \times 128 \times (128 \times 8)$. Figure 5 shows the result.

We also ran the program on GPUs. We used one GPU per node and all the computation was performed on GPUs (CPUs were not used for the computation). Figure 6 shows the weak scalability on GPUs. The problem size was $384 \times 384 \times 384$ per GPU. It requires

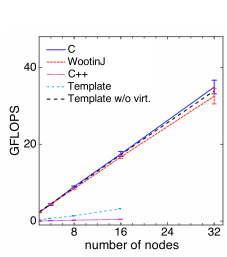

Figure 4. The weak scalability on CPUs

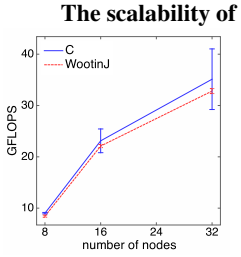

Figure 5. The strong scalability on CPUs

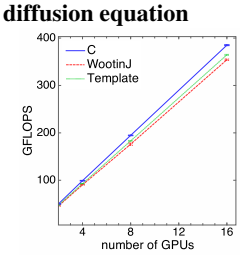

Figure 6. The weak scalability with GPUs

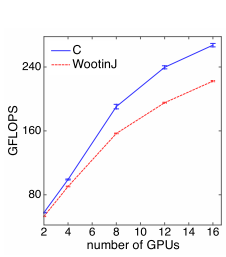

Figure 7. The strong scalability with GPUs

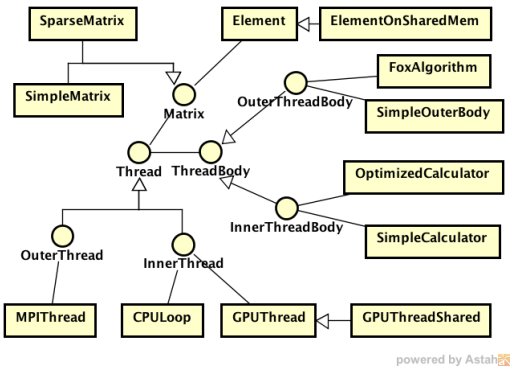

Figure 8. The class diagram of our class library for matrix multiplication

Program	Compiler options
C++	-ipo -O3 -rcd -i-static -xHost
Template	-ipo -O3 -rcd -i-static-xHost
Template w/o virt.	-ipo -O3 -rcd -i-static -xHost -parallel
C	-ip -O3 -rcd -i-static

Table 2. The compiler options for matrix multiplication

the whole memory of GPUs. The strong scalability on GPUs is showed in Figure 7. The problem size was 384 × 384 × (384 × 4).

4.2 Matrix multiplication

We also wrote a class library for computing matrix multiplication in different algorithms. Figure 8 shows the class diagram of this class library. The class library consists of three kinds of components: Thread, ThreadBody, and Matrix, which are implemented by interfaces. Thread represents how to run the kernel computation in parallel. The class library users can choose MPIThread for using MPI, CPULoop for computing only on CPUs, and GPUThread for using GPUs. ThreadBody represents a parallel algorithm of matrix multiplication such as the Fox algorithm. Matrix represents the data structure implementing a matrix. The class library users can choose a normal dense matrix or a sparse matrix.

For evaluation, we wrote a program combining the library classes SimpleMatrix (for a dense matrix), SimpleOuterBody, and OptimizedCalculator. The program was compiled with the options listed in Table 2. We chose these options since they showed the best performance.

Figure 9 shows the weak scalability of the program running on multiple nodes communicating through MPI. Only one CPU/thread was used per node and no GPUs were used. The problem size was

	sec
Matrix Mul. (single CPU)	1.405 ± 0.046
Matrix Mul. (MPI)	3.060 ± 0.696
Matrix Mul. (MPI & GPU)	4.138 ± 1.246
3D diffusion (single CPU)	2.039 ± 0.258
3D diffusion (MPI)	3.273 ± 1.097
3D diffusion (MPI & GPU)	3.781 ± 1.289

Table 3. The compilation time by WootinJ

2048 × 2048 × 2048 per node. Figure 10 shows the strong scalability of the programs *C* and *WootinJ*. The problem size was 2048 × 2048 × (2048 × 8). Figure 11 shows the weak scalability of the programs running on GPUs. All the computation was performed on GPUs and CPUs were used only for inter-node communication. The problem size was 14592 × 14592 × 14592 per GPU. It requires the whole memory of GPUs. The strong scalability with GPUs is showed on Figure 12. The problem size was 14592 × 14592 × (14592 × 8).

The results of our experiments revealed that the WootinJ programs still involves runtime penalties against the C programs. For the comparison to the C++ programs, the WootinJ programs are significantly faster than the plain C++ programs using virtual functions but the performance difference between the WootinJ programs and the C++ programs using templates depends on how well the optimization works. In the case of the diffusion equation on CPUs, *Template w/o virt.* outperformed the WootinJ program whereas it showed unsatisfactory performance in the case of matrix multiplication on CPUs. When the programs ran on GPUs, *Template* always showed similar performance to the WootinJ program. However, as we have already mentioned, the C++ programs using templates have drawbacks with respect to code reuse and modularity. The C programs also share those drawbacks.

4.3 Compilation time

A major factor of the execution overhead of WootinJ is compilation time during runtime. Table 3 shows the compilation time by WootinJ. It is about four to five seconds. The compilation time includes the time for code generation by WootinJ and the compilation of the generated code by an external compiler. Since this overhead is getting relatively smaller as the translated code runs longer and also it is independent of the problem size, we show the strong scalability of the WootinJ program excluding compilation time in Figure 13 to 16. These figures show that the execution performance of the WootinJ programs are comparable to the C programs written by hand without considering code reuse and modularity.

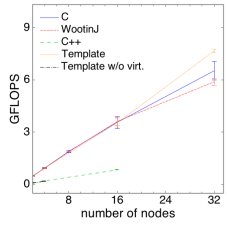

Figure 9. The weak scalability on CPUs

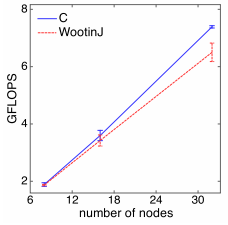

Figure 10. The strong scalability on CPUs

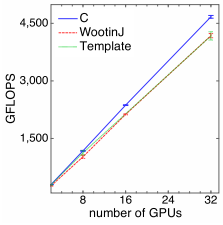

Figure 11. The weak scalability with GPUs

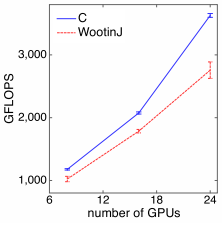

Figure 12. The strong scalability with GPUs

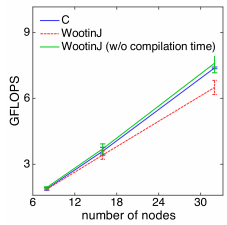

Figure 13. The matrix multiplication on CPUs

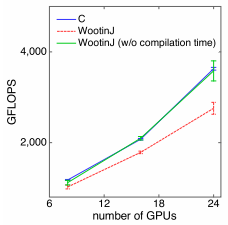

Figure 14. The matrix multiplication with GPUs

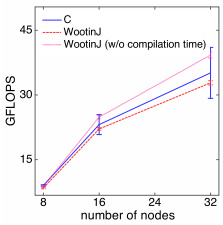

Figure 15. The 3D diffusion equation on CPUs

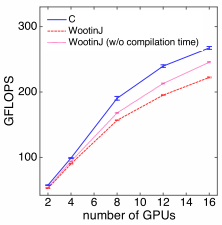

Figure 16. The 3D diffusion equation with GPUs

4.4 Comparison to Java

The basic architecture of WootinJ is similar to the just-in-time compilation of the Java virtual machine (JVM). To investigate this similarity, we ran the programs written for our experiments on a single CPU/thread without a GPU and then compared to the performance of the Java program running without WootinJ. Note that the class libraries written with WootinJ are Java programs and hence the application programs built on these class libraries can run without WootinJ unless they use MPI or GPUs.

Figure 17 shows the performance of the programs for the three-dimensional diffusion equation. It is an extension of Figure 3. The problem size is $128 \times 128 \times 128$. Figure 18 shows the performance of the matrix multiplication. The problem size is $1024 \times 1024 \times 1024$. These figures show that WootinJ achieved considerably better performance than Java running on the JVM. Since WootinJ requires that the translated code is subject to our coding rules, which help the optimization, it can perform aggressive devirtualization and object inlining. On the other hand, the JVM executes the program as a pure Java program and thus the capability for the optimization is restricted. The figures also show that WootinJ achieves relatively better performance for the three-dimensional diffusion solver, which is a more complex class library. Such a class library involves more objects and more dynamic method dispatches and thus it has larger room for optimization.

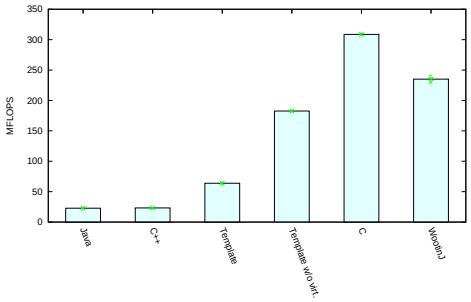

Figure 17. The performance of the 3D diffusion ($128 \times 128 \times 128$)

5. Related Work

5.1 Java (bytecode) on GPUs

There have been several tools for providing foreign function interfaces from Java to CUDA. For example, JCuda [11] allows programmers to call from Java a CUDA function running on a GPU. The programmers do not have to use JNI (Java Native Interface), which is known as a complicated system. JCUDA [19] also provides foreign function interfaces from Java to CUDA. It extends the syntax of Java to enable natural syntax to call a CUDA function. Since these tools provide foreign function interfaces, a function running on a GPU is written in not Java but CUDA by hand although WootinJ allows writing it in Java.

Like WootinJ, Aparapi [1] and Rootbeer[16] allow programmers to write a function in Java and run it on a GPU. Aparapi is a translator from a subset of Java to OpenCL [13] on AMD GPUs. Unlike WootinJ, however, Aparapi does not support objects or dynamic method dispatch. Rootbeer does not support dynamic method dispatch as well.

JCudaMP [7] is an OpenMP [3] framework for Java. This framework translates Java bytecode into CUDA including OpenMP directives. Unlike WootinJ, it does not support dynamic method dispatch. JConqurr [9] is a toolkit for many-core programming in Java. For GPU programming, this tool helps developers convert a for loop in Java code into an equivalent GPU function. Unlike WootinJ, JConqurr is not a tool for translating a Java method into a function running on a GPU.

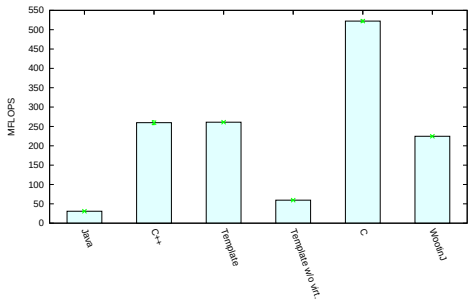

Figure 18. The performance of the matrix multiplication ($1024 \times 1024 \times 1024$)

JaBEE [20] is the execution environment for Java bytecode on a GPU. It fully supports object-oriented features of Java, including dynamic method dispatch. Since the aim of this work is to fully support object orientation and execute normal Java bytecode, JaBEE does not support aggressive devirtualization or object inlining performed by WootinJ. Firepile [15] is a runtime compiler for translating Scala code into OpenCL code. Like JaBEE, it supports dynamic method dispatch appearing in the functions running on a GPU. The aim of Firepile is also the full support of object orientation. The aggressive devirtualization or object inlining performed by WootinJ are not supported.

5.2 Optimization techniques

The class hierarchy analysis has been used as a simple technique for devirtualization [4, 8]. If the target method of devirtualization has only a single implementation, the method call can be replaced with static method dispatch. To minimize the overheads due to guard tests, Ishizaki *et al.* proposed a technique called direct devirtualization with the code patching mechanism [10]. The approach of WootinJ to devirtualization is different since WootinJ assumes that the translated code satisfies the coding rules, which simplify devirtualization.

The object inlining is also a well-known technique. Dolby *et al.* developed a static compiler for object inlining in C++ [5, 6]. This compiler performs whole program analysis and checks whether it can perform object inlining. Wimmer *et el.* proposed automatic feedback-directed object inlining for the JVM [18]. The JVM automatically analyzes field accesses and applies object inlining to frequently accessed fields if it is possible. These approaches are different from WootinJ's since WootinJ requires that all references in the program satisfy the preconditions of object inlining and hence WootinJ does not check object inlining is applicable.

6. Conclusion

We proposed a framework for multiplatform HPC applications in Java. The applications built on this framework achieve execution performance comparable to C++ code heavily using the template meta-programing technique, which often results in complicated, difficult-to-read code. To support MPI and/or GPU programming in Java, our framework adds simple extensions to Java, which are implemented with annotations but without syntax extension. The annotated Java code is dynamically translated into the CUDA or C code using MPI according to the underlying platform. Since our framework aggressively applies devirtualization and object inlining, our framework requires that the translated code is subject to our coding rules. Although the coding rules significantly restrict the coding style, this paper presented that we could develop a class library for stencil computing running on CPUs or GPUs in parallel through MPI. It achieved execution performance comparable to C++ code optimized by template meta-programming. Our future plans include to develop larger class libraries in the HPC domain and evaluate the practicality of our framework.

References

[1] Aparapi. AMD developer central. `http://developer.amd.com/zones/java/aparapi/Pages/default.aspx`.

[2] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. *SIGPLAN Not.*, 46(8):35–46, Feb. 2011. ISSN 0362-1340. . URL `http://doi.acm.org/10.1145/2038037.1941561`.

[3] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[4] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP '95*, pages 77–101. Springer-Verlag, 1995.

[5] J. Dolby. Automatic inline allocation of objects. In *PLDI '97*, pages 7–17. ACM, 1997.

[6] J. Dolby and A. Chien. An automatic object inlining optimization and its evaluation. In *PLDI '00*, pages 345–357. ACM, 2000.

[7] G. Dotzler, R. Veldema, and M. Klemm. JCudaMP: OpenMP/Java on CUDA. In *Proceedings of the 3rd Int'l Workshop on Multicore Software Engineering (IWMSE '10)*, pages 10–17. ACM, 2010.

[8] M. F. Fernández. Simple and effective link-time optimization of Modula-3 programs. In *PLDI '95*, pages 103–115. ACM, 1995.

[9] G. Ganegoda, D. Samaranayake, L. Bandara, and K. Wimalawarne. JConqurr - a multi-core programming toolkit for Java. *Int'l Journal of Computer and Information Engineering*, 3(4), 2009.

[10] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. A study of devirtualization techniques for a Java just-in-time compiler. In *OOPSLA '00*, pages 294–310. ACM, 2000.

[11] jcuda.org. jcuda.org - Java bindings for CUDA. `http://www.jcuda.de`.

[12] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, DTIC Document, 1990.

[13] Khronos OpenCL Working Group. The OpenCL specification, 2008.

[14] C. Mellon. Software product lines — overview. `http://www.sei.cmu.edu/productlines`.

[15] N. Nystrom, D. White, and K. Das. Firepile: run-time compilation for GPUs in Scala. In *Proc. of the 10th ACM int'l conf. on Generative Programming and Component Engineering (GPCE '11)*, pages 107–116. ACM, 2011.

[16] P. C. Pratt-Szeliga, J. W. Fawcett, and R. D. Welch. Rootbeer: Seamlessly using GPUs from Java. In *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 375–380. IEEE, 2012.

[17] Tokyo Institute of Technology. TSUBAME computing services. `http://tsubame.gsic.titech.ac.jp`.

[18] C. Wimmer and H. Mössenböck. Automatic feedback-directed object inlining in the Java hotspot virtual machine. In *Proc. of the 3rd int'l conf. on Virtual Execution Environments (VEE '07)*, pages 12–21. ACM, 2007.

[19] Y. Yan, M. Grossman, and V. Sarkar. JCUDA: A programmer-friendly interface for accelerating Java programs with CUDA. In *Euro-Par*, pages 887–899, 2009.

[20] W. Zaremba, Y. Lin, and V. Grover. Jabee: framework for object-oriented Java bytecode compilation and execution on graphics processor units. In *Proc. of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units (GPGPU-5)*, pages 74–83. ACM, 2012.