Writing a modular GPGPU program in Java

Masayuki Ioki Shumpei Hozumi Shigeru Chiba
Tokyo Institute of Technology
www.csg.is.titech.ac.jp

Abstract

This paper proposes a Java to CUDA runtime program translator for scientific-computing applications. Traditionally, these applications have been written in Fortran or C without using a rich modularization mechanism. Our translator enables those applications to be written in Java and run on GPGPUs while exploiting a rich modularization mechanism in Java. This translator dynamically generates optimized CUDA code from a Java program given at bytecode level when the program is running. By exploiting dynamic type information given at translation, the translator devirtualizes dynamic method dispatches and flattens objects into simple data representation in CUDA. To do this, a Java program must be written to satisfy certain constraints. This paper also shows that the performance overheads due to Java and WootinJ are not significantly high.

Categories and Subject Descriptors D.3.3 [*PROGRAMMING LANGUAGES*]: Language Constructs and Features

General Terms Languages

Keywords HPC, CUDA, Java

1. Introduction

A program for scientific computing has been written in Fortran — by real programmers. This is mainly for achieving the highest performance of underlying hardware called a supercomputer but, as modern machine architecture such as distributed memory, multicores, and GPGPU is getting adopted, the programming costs have been significantly increased. Writing a classic Fortran program for modern supercomputers is never a simple task today if achieving the highest performance is a primary goal.

Modern supercomputers require programmers to consider not only scientific computation but also performance-aware coding. How to parallelize computation, how to minimize data transfers, and how to utilize maximum memory bandwidth are examples of performance-aware coding. Programmers have to deal with those two concerns at the same time and classic languages including Fortran and C do not provide sufficient modularization capability for separately implementing the two concerns in a clear and easy manner. Thus the resulting code is tangling and increases development and maintenance costs.

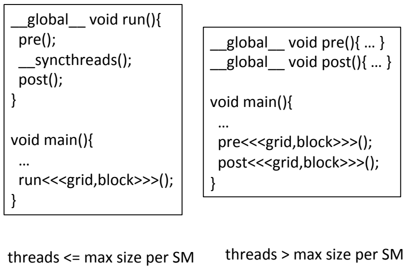
This tangling problem is not only on the programming in Fortran or the like. Popular tool sets for high-performance computing (HPC) such as MPI [7] and PGAS[10] provides relatively lower-level abstraction, which does not hide underlying hardware architecture so that programmers can be aware of it for performance tuning. The tangling problem could be addressed by introducing a modern language with a rich modularity mechanism but the performance penalties due to modularization has been a major obstacle to the adoption of such a language in HPC.

This paper presents our approach to make rich modularization available in HPC. As the first step, we are developing a Java-based programming system for CUDA [8], named *WootinJ*. CUDA is Nvidia's programming environment for GPGPU (General-Purpose Graphics Processing Units, or GPU in short) computing. Since GPU is getting widely adopted by top supercomputers, for example, the world's 5th fastest supercomputer TSUBAME 2.0 [9] of our university, supercomputer programmers have to utilize GPUs and hence write a CUDA program to obtain the best performance. The language used in CUDA is an extension of the C language. WootinJ provides a program translator from Java to CUDA, which performs the translation at runtime for advanced optimization. Despite its runtime code translation, the performance overhead is not significantly high. WootinJ exploits dynamic type information and performs devirtualization and flattens object structures so that they will be simple data representations in C. On the other hand, WootinJ accepts only a Java program written in the restricted form.

2. A CUDA program

In CUDA, programmers have to carefully consider underlying hardware architecture. Figure 1 shows simplified examples of CUDA programs. In both programs, pre and post

MISS'12, March 27, 2012, Potsdam, Germany.
Copyright © 2012 ACM 978-1-4503-1217-2/12/03. . . $10.00



```
__global__ void run(){
  pre();
  __syncthreads();
  post();
}

void main(){
  ...
  run<<<grid,block>>>();
}
```

threads <= max size per SM

```
__global__ void pre(){ ... }
__global__ void post(){ ... }

void main(){
  ...
  pre<<<grid,block>>>();
  post<<<grid,block>>>();
}
```

threads > max size per SM

Figure 1. CUDA code samples

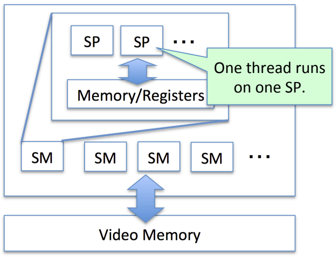
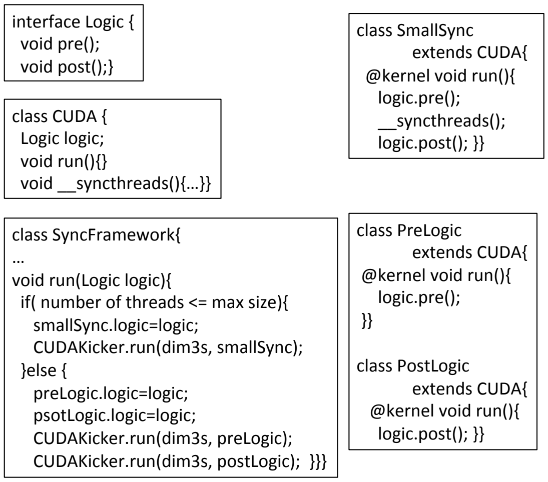


Figure 2. The architecture of Nvidia GPU

functions implement the core logic of scientific computing and they are run on GPUs. Since CUDA programs are written in the SPMD (Single Program Multiple Data) style, these functions are responsible for the computation on only part of array elements. At runtime, a large number of threads are created to execute those functions in parallel. In CUDA, <<<grid,block>>> following a function name specifies that the function is invoked by a number of threads.

An interesting problem is synchronization. In CUDA, programmers can be aware of the underlying GPU architecture to obtain the best performance. The flip side is that they must write a program with lower-level abstraction, for example, with respect to synchronization. Suppose that the post function must start after all the invocations of the pre function. In the left program in Figure 1, the main function is run on a host processor (CPU) creates threads on a GPU and each thread executes the run function in parallel. The run function calls the pre function and then __syncthreads function for synchronization. The post function is called after __syncthreads.

This program, however, works only when the total number of threads is small since the physical synchronization is available only for the threads running on the same Streaming Multiprocessor (SM). Figure 2 illustrates the basic architecture of Nvidia GPUs. A GPU has several SMs (Streaming Multiprocessors), each of which consists of 32 CUDA cores (or Streaming Processors, SPs) in the Fermi generation and performs SIMD (Single Instruction Multiple Data) computing with them. Hence, physical synchronization is available only per 32 threads. CUDA also provides a logical thread group called *a block*. The __syncthreads function synchronizes all the threads in the same logical group although these threads must run on the same SM and thus the maximum parallelism is limited.



```
interface Logic {
  void pre();
  void post();}
```

```
class CUDA {
  Logic logic;
  void run(){}
  void __syncthreads(){...}}
```

```
class SyncFramework{
...
void run(Logic logic){
  if( number of threads <= max size){
     smallSync.logic=logic;
     CUDAKicker.run(dim3s, smallSync);
  }else {
     preLogic.logic=logic;
     psotLogic.logic=logic;
     CUDAKicker.run(dim3s, preLogic);
     CUDAKicker.run(dim3s, postLogic);  }}}
```

```
class SmallSync
          extends CUDA{
  @kernel void run(){
    logic.pre();
     __syncthreads();
    logic.post(); }}
```

```
class PreLogic
          extends CUDA{
 @kernel void run(){
    logic.pre();
 }}

class PostLogic
          extends CUDA{
  @kernel void run(){
    logic.post(); }}
```

Figure 3. A synchronization framework in Java

If the number of threads is large, the right program in Figure 1 should be used to increase parallelism. Note that this program does not include a call to __syncthreads. The main function directly invokes the pre and post functions with <<<grid,block>>>. Each function is executed in parallel in the SPMD style. Since CUDA implicitly performs global synchronization to wait until all the threads started by <<<grid,block>>> complete, the program does not have to explicitly perform synchronization. However, this program is slow if the number of threads is small. Since the main function runs on a CPU, the overheads due to the invocation by <<<grid,block>>> is not negligible.

From the viewpoint of software maintenance, the pre and post functions should be modularized separately from the main (and run) function. For the latter, two sets of reusable modules should be prepared for large and small problem sizes and programmers should be able to choose between them. Implementing the latter module to be generic and reusable is significantly difficult since language constructs used for implementing reusable modules are not available in CUDA. For example, function pointers are not available. The reason would be making compiler optimization for SIMD programs easy.

If a Java-like language were available, the module separation above would be straightforward although programmers must accept serious performance penalties. For example, Figure 3 shows a pseudo framework for synchronization in Java. The computation concern corresponding to the pre and post functions is represented by the Logic interface. The performance concern corresponding to the main (and run) function is represented by the CUDA class. Programmers write concrete classes implementing or inheriting from Logic and CUDA. SyncFramework is a framework class that combines instances of Logic and CUDA according to the problem size. The run method in another framework class CUDAKicker corresponds to <<<grid,block>>> in CUDA, which starts threads on a GPU in the SPMD style.

3. WootinJ

To allow writing a program in a Java-like language for GPGPU with acceptable overheads, we are developing a programming system named *WootinJ*. The main component of WootinJ is a Java-to-CUDA runtime translator. It translates a Java method implementing computation that will run on a GPU. We call this method *a kernel method* to follow the naming convention of CUDA. WootinJ runs on the standard JVM on a host processor (CPU). During runtime, WootinJ receives a kernel method, the target object that the method is called on, and the arguments passed to the method. Then it translates the method and the related methods called within that method. The generated code is a CUDA program. WootinJ then compiles it, automatically transfers it to a GPU, and executes it there with the arguments.

Since WootinJ receives a target object and actual arguments when translating a Java method, it generates optimized code by using the actual types of the target and the actual arguments. All dynamic method dispatches are devirtualized and Java objects are flattened into plain data representation available in C, such as an array and a set of primitive values. Indirect field references are statically resolved. To help these optimization techniques, the types of all object references must be statically resolvable; programmers must guarantee that no polymorphism is used in the Java methods translated by WootinJ. Only the field values of the target object and the parameters to the kernel method are exceptions.

This optimization allows the generated CUDA code to show comparable performance against the equivalent program written by hand. Although the overheads due to runtime code generation is not small and the constraints on polymorphism is not negligible, these drawbacks are acceptable for our primary goal, building a framework in Java for separating computation-concern code and performance-concern code. An important usage of polymorphism in framework code is to combine objects implementing different concerns. In a typical framework in scientific computing, the parameters to a kernel method are such objects implementing a concern and other variables used in the kernel method do not need polymorphism. Thus, we expect that the constraints on polymorphism are not a serious drawback. Furthermore, the values of those parameters do not change during a large number of iterations of executing the kernel method within a kernel loop. This lets WootinJ reuse the generated CUDA code and reduces the overhead due to runtime code generation.

3.1 The execution model

WootinJ converts a Java method to an optimized CUDA code at runtime. This translater constructs a Java AST(Abstract Syntax Tree) from the Java bytecode of the method at runtime. From that AST, this translater generates a CUDA code. That code is compiled by nvcc, CUDA compiler, and WootinJ runs it on GPU. Other methods called from that method are also converted into CUDA code.

Listing 1. Java's dynamic method dispatch sample

```
class A{ @kernel int getI(){return 10;}}
class B extends A{
  @kernel int getI(){ return -1;}}

@kernel void run(A a,A b){
  int x=a.getI();
  int y=b.getI(); }
```

WootinJ's main method is CUDAKicker.run:

```
CUDAKicker.run(Dim3s dim3s, Object target,
               String method, Object[] args)
```

The first argument dim3s is the informations of number of threads. The second argument target is the object that the kernel method is called on. The third argument method is the name of that kernel method. The last argument args is the arguments passed to that kernel method. With this run method, the users can specify which method is run on a GPU with what arguments.[1]

The data that are used in a kernel method are automatically send to GPU memory from CPU memory (Java memory). But the data that are rewritten in GPU are not automatically got back from GPU memory. This design is for speed, because the translation data between CPU and GPU is very heavy. The users want to use that data in CPU, they must explicitly use a WootinJ's method CUDAData.get. For example, in the case of using an array of int, named arr, WootinJ automatically sends that data and manages that the pointer in a GPU memory. To get back that data to CPU memory, the user write CUDAData.get(arr). CUDAData.get method's argument is not a pointer but a plain Java object. The pointers are managed by WootinJ. The users can forget the pointers.

Kernel methods can be run as pure Java methods. The user can choose either GPU mode or pure Java mode. Since the user can debug in the pure Java environment.

3.2 Devirtualization

For optimizing kernel methods, WootinJ changes dynamic method dispatches to static method dispatches in those at a point of running those. Actually each method dispatch in kernel methods is changed to a unique function call in the generated CUDA code. In the Listing 1, the appearance types of arguments of run method are the same. @kernel is an annotation of kernel method. If the actual type of object b is B and the actual type of a is A, this code will be converted to the Listing 2. Each method call is converted to CUDA function call(__device__ means that is a function called from global functions). Each function body is got from the Java method body determined by the actual type.

This optimization requires following conditions.

[1] A call to CUDAKicker.run(Dim3s dim3s,Object target) is equivalent to a call to CUDAKicker.run(dim3s, target, "run", null).

Listing 2. converted CUDA code

```
__device__ int A_getI(){ return 10;}
__device__ int B_getI(){ return -1;}
__global__ void run(){
  int x=A_getI();
  int y=B_getI(); }
```

Listing 3. a sample of using Java objects

```
final class A{ B b; int x;}
final class B{ int y;}

@kernel public void run(A a){
  int sum = a.x+a.b.y; }
```

1. In the kernel method, both types of sides in the assignment expression must be strcitfinal.
2. The return type of a kernel method must be strictfinal.
3. The array in the kernel method must be strictfinal.
4. The kernel method must not call non kernel methods(or Constructors).

Strictfinal is an attribute of type (We will explain about strictfinal later). This attribute warrants to be same an appearance type and an actual type. From second and forth condition, any objects generated in kernel method are strictfinal. On the other hand, objects generated at outside kernel methods are not required strictfinal. These objects are given to a kernel method as the parameters or fields of it. These conditions are able to be checked at compile time.

The definition of strictfinal are shown below.

1. Primitive types are strictfinal.
2. An array that its element type is strictfinal, is strictfinal.
3. The class that is final class and all fields (include super classes' fields) are strictfinal, is strictfinal.

For example, there is an object t that its appearance type is strictfinal type T, the actual type of t must be T. Because the strictfinal type has no subtypes. Any method calls from a strictfinal object can be determined a unique method body from its appearance type.

3.3 Flattening the structure of an object

WootinJ supports the optimization for field accesses in Java. Java's field access may be not small overhead as a CUDA code. Since WootinJ flattens structures of objects in kernel method. For example, we consider the situation in Listing 3. There are some ways to change this Java code to CUDA code. Maybe a simple way is to use the C's struct and arrow operator(like a.b-¿y), but the arrow operator costs some time to access to the structs. In this case, if there is no side-effect for the object a in the kernel method, we can consider the object a as a group of values (x and y). Since we can get the result in Listing 4. In this way, WootinJ can change objects to groups of primitive values under certain conditions.

This optimization requires the following conditions.

Listing 4. flattern CUDA code

```
__global__ void run(int x,int y){
  int sum = x+y; }
```

1. The target object's type is not the recursive type.
2. After creating an alias of the target object, there is no side-effect for those.
3. If the tager object is a paramater of the kernel method, there is no side-effect for it.

The side-effect means an assignment to the variable or the field. WootinJ considers Java class as a group of primitive types, since cannot flatten the recursive type. And current WootinJ does not have the alias analysis feature, the second condition is required.

If a return object is the flatten target object, WootinJ will do in-line expansion for that method. Since WootinJ does not do the interprocedural analysis.

3.4 Other Issues

The semantics of Java is not completely preserved. For example, an array in Java is converted to an array in C. In the generated CUDA code, even if there is an access to the array with negative index, any exceptions will be not thrown. The semantics of multidimensional array is not preserved, either. The multidimensional array in Java is expressed as an array of arrays, but in the generated CUDA code, it is converted a multidimensional array in C.

4. Micro benchmark

We did the micro benchmark about our tool's performance. We used the synchronization sample, to add to each elements of the array with that next elements. The pre method adds the next element on the right hand, and post method adds the left element. That calculation is run ten million times.

We developed three versions programs for that. The first is WootinJ version. We developed a small threads sample(1024 threads: in our micro benchmark's environment, the max size of threads per SM is 1024) using WootinJ and a big threads(2048 threads) sample using WootinJ. The Second version is the writing CUDA codes in hand that are same applications with WootinJ version's. The last is the pure C version that do not use the GPU.

We measured three points of the performance. The first point is the overhead of generated CUDA code. For this, we measured the GPU running time for the WootinJ version and the hand version. The second point is a time of generating and compiling CUDA code. The last point is a comparison of CPU and GPU. For this point, we measured the running time of pure C code without GPU. The environment of this micro benchmark is TSUBAME 2.0 [9] that CPUs are two Intel Xeon 2.93 GHz (6 cores) , GPUs are three NVIDIA Tesla M2050 and capacity of memory is 54GB.

The result is shown in Table 1. The time of generating and compiling CUDA code are about 5 sec. This overhead is large, but this is done only once on the application runtime. If the whole calculation time is large, this overhead is not serious. About the GPU running time, WootinJ version is bit slower than Hand version. The 1024 version is not so different. The 2048 version's overhead is about 13% of the Hand version. Both versions of WootinJ and Hand are faster than the C version.

Table 1. micro benchmark result

	GPU time[sec]	CUDA code generation and compile[sec]
WootinJ 1024	15.61 ± 0.03	5.01 ± 0.01
WootinJ 2048	79.52 ± 1.69	5.06 ± 0.01
Hand 1024	15.56 ± 0.01	-
Hand 2048	68.93 ± 0.79	-
C 1024	82.46 ± 0.94	-
C 2048	166.03 ± 0.62	-

5. Related Works

5.1 Java to CUDA

JCuda[6] is a bridge tool between Java and CUDA. This tool has APIs for connecting to CUDA from Java. But this tool cannot automatically convert Java to CUDA. The users must write a CUDA code by themselves.

JCUDA[11] is an expanded Java language for accessing to CUDA(This is not same as above one). This brings in new syntaxes to Java, for example, this has a notation of ⟨⟨⟨...⟩⟩⟩ to identify a kernel method. On the other hand, users of WootinJ can write their code in pure Java.

JConqurr[4] is a toolkit for multi-core programming in Java. For GPU, this tool provides the feature for loop handling. This converts the for-loop in user's Java code to the GPU function. But this tool has no optimization for Java's rich modularities. WootinJ optimises the Java's rich modularities.

5.2 Optimization Techniques

Class hierarchy analysis can be used for devirtualization[2][3]. If the devirtualization target method has a single implementation, its method call can be replaced by the static method dispatch. Kszuaki Ishizaki et el proposed the technique called *direct devirtualization with the code patching mechanism*[5]. In this technique, the compiler generates the inline code of the method, together with the backup code of making the dynamic method dispatch. At runtime, if the inlined code is invalid, this system uses the backup code instead of that. WootinJ can deal with the overrided methods under WootinJ's constrain, and convert the dynamic method dispatches to the static method dispatches at runtime.

The object flattening is known. Kaiyu Chen et el use the object flattening for generating the optimized C++ code[1]. On the other hand WootinJ generates the CUDA code.

6. Conclusions

We proposed a Java to CUDA translater, WootinJ, for scientific computing applications. This tool generates the optimized CUDA code from a Java method under certain constrains at runtime. That constrains are about the dynamic method dispatch, however this tool saves the ability of constructing a Java framework for separation of concerns. Since the users can write a application for scientific computing using Java's rich modularities with small overheads.

In the future we plan to reduce WootinJ's constrains. For example, if WootinJ has the alias analysis, several restrictions of the object flattening will be removed. The cost of generating and compiling CUDA code is big. We should reduce it.

References

[1] K. Chen, S. Chan, R.-C. Ju, and P. Tu. Optimizing structures in object oriented programs. In *Interaction between Compilers and Computer Architectures, 2005. INTERACT-9. 9th Annual Workshop on*, pages 94 – 103, feb. 2005.

[2] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, pages 77–101, London, UK, UK, 1995. Springer-Verlag.

[3] M. F. Fernández. Simple and effective link-time optimization of modula-3 programs. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 103–115, New York, NY, USA, 1995. ACM.

[4] G. Ganegoda, D. Samaranayake, L. Bandara, and K. Wimalawarne. Jconqurr - a multi-core programming toolkit for java. *International Journal of Computer and Information Engineering*, 3(4), 2009.

[5] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. A study of devirtualization techniques for a java just-in-time compiler. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '00, pages 294–310, New York, NY, USA, 2000. ACM.

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

[7] MPI. Message passing interface. http://www.mcs.anl.gov/research/projects/mpi/.

[8] Nvidia. Parallel Programming and Computing Platform. http://www.nvidia.com/object/cuda_home_new.html.

[9] T. I. of Technology. Tsubame computing services. http://tsubame.gsic.titech.ac.jp.

[10] UPC. Berkeley upc - unified parallel c. http://upc.lbl.gov.

[11] 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.