JaBEE - Framework for Object-oriented Java Bytecode Compilation and Execution on Graphics Processor Units

Wojciech Zaremba
Jacques-Louis Lions Laboratory
woj.zaremba [at] gmail.com

Yuan Lin
NVIDIA Corporation
yulin [at] nvidia.com

Vinod Grover
NVIDIA Corporation
vgrover [at] nvidia.com

Abstract

There is an increasing interest from software developers in executing Java and .NET bytecode programs on General Purpose Graphics Processor Units (GPGPUs). Existing solutions have limited support for operations on objects and often require explicit handling of memory transfers between CPU and GPU. In this paper, we describe a Java Bytecode Execution Environment (JaBEE) which supports common object-oriented constructs such as dynamic dispatch, encapsulation and object creation on GPUs. This experimental environment facilitates GPU code compilation, execution and transparent memory management. We compare the performance of our approach with CPU-based and CUDA-C-based code executions of the same programs. We discuss challenges, limitations and opportunities of bytecode execution on GPGPUs.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Object-oriented languages, Concurrent, distributed, and parallel languages; D.3.4 [*Processors*]: Compilers, Run-time environments

General Terms Languages, Performance

Keywords Java, bytecode, PTX, compilation, translation, virtual table, embedded language, CUDA, VMKit, SIMD, dynamic compilation

1. Introduction

Graphics Processing Unit (GPU) based parallel computing offers a significant performance improvement for many of the computational tasks in comparison to the traditional CPU-based computing. The performance increases from five up to even fifteen times are common, when comparing with the modern CPUs [1, 15]. Major GPU vendors such as NVIDIA have designed their own programming environments for GPU-based computing such as CUDA ([18]) and OpenCL([10]). A number of approaches already exist enabling GPU-based code execution from the high-level programming languages (e.g., [4, 5, 13, 14, 19]).

This paper presents JaBEE (Java bytecode execution environment) which is an experimental environment supporting GPU-based execution of Java programs by facilitating their GPU-centric compilation and transparent memory management. Our aim is to understand challenges, limitations and opportunities associated with running bytecode languages on the General-Purpose computation on Graphics Processing Units (GPGPU). The main difference and contribution of the presented research, which distinguishes JaBEE from existing similar solutions, is the support for object oriented constructs. Projects OpenCL [10], JCuda [21], JavaCL[1] do not allow for any operations on the objects processed on the device side. Aparapi[2] provides a mechanism for access to object fields on the device, but not allow method calls on an object. CUDA [18] does not allow passing objects with virtual functions between the host's and the device's environment. JaBEE is able to perform the following operations on a GPU:

- Object creation
- Dynamic dispatch
- Passing objects between a host and a device

We describe main components of our system in section 2, followed by section 3 where we show an example program that computes Julia set. We then describe the compilation process in section 4 and the data management in section 5. In section 6, we compare performance results among different frameworks: (1) CPU-based Java virtual machine code execution, (2) GPU-based Java byte code execution and (3) GPU-based CUDA code execution. We discuss related work in the section 7. Finally, we discuss future work and highlight our contribution with respect to other approaches.

GPGPU-5 March 03 2012, London, United Kingdom

[1] http://code.google.com/p/javacl

[2] http://code.google.com/p/aparapi/

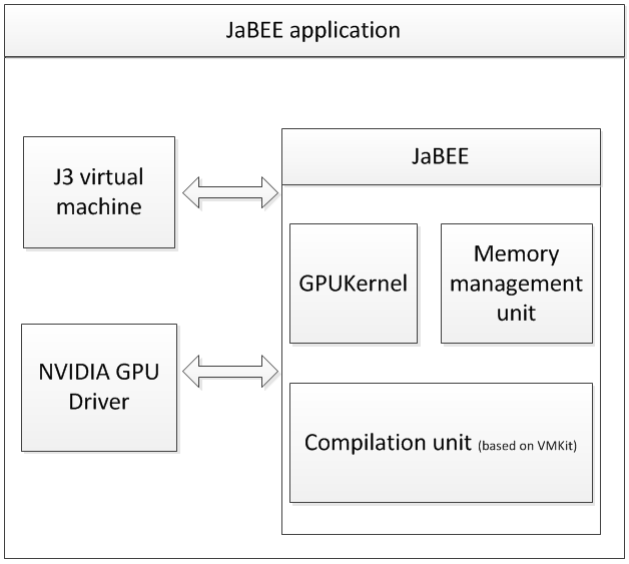

Figure 1: System architecture

The appendix presents a sample code fragment showing the transformation between various phases.

2. JaBEE Architecture

JaBEE architecture is based on a hybrid model allowing selective bytecode execution on both CPU and GPU. Building a complete JVM that executes all bytecode on GPU is currently out of the scope of this research. JaBEE system consists of three main components:

1. A base Java class called GPUKernel, which provides Java interface for GPU code execution (see section 3)
2. An online compiler that selectively compiles Java bytecode to the GPU code (see section 4)
3. A memory management system, which transfers data between CPU and GPU (see section 5)

Figure 1 presents an overview of the system architecture.

A JaBEE execution is initiated by an incoming call from a Java code requesting to perform GPU kernel execution. The control is passed first to the *GPUKernel* object, which is a Java class instance providing an interface for GPU code execution and validation of correctness of a kernel launch request. The GPUKernel passes control further to the native code. The system collects necessary information from the native code, by creating a collection of dependent classes required for compilation, it validates variables names and creates maps of pointers, all of which are needed during further phases of the execution process. Then the control is further passed to the *Compilation Unit*, which compiles a bytecode to a GPU assembly code (this step is optional as a code might be already compiled, when passed to this stage). The control goes next to *Memory management unit*, which copies all the necessary data to GPU and adjusts pointers through changing CPU RAM pointers to GPU RAM pointers. As one of the last steps of the execution process, GPU kernel is getting executed. Finally, after GPU kernel execution, memory is copied back by *Memory management unit* and control returns back to Java code.

2.1 VMKit - Framework for building virtual machines

JaBEE is built on top of VMKit[3] ([8, 9]) which is a substrate that supports building custom virtual machines. VMKit currently has a decent implementation of a JVM called J3. It uses LLVM for compiling and optimizing high-level languages to machine code. VMKit supports three modes of code execution: (1) Ahead-of-time compilation, (2) Code interpretation and (3) Just-in-time compilation. In the Ahead-of-time compilation mode, it can generate LLVM IR files which after linking against Java core libraries and native code generation become executable.

We select VMKit as the backbone for our JaBEE project for practical reasons. VMKit is open source. It supplies a Java bytecode-to-LLVM IR compiler which JaBEE uses as the front-end for its online GPU compiler. JaBEE use J3 as the JVM for bytecode execution on CPU.

3. Core Execution - GPUKernel

GPUKernel is a Java class, which acts as an interface for GPU code execution. Here we uses a simple example to show how *GPUKernel* serves as a 'bridge' between the CPU code execution and the GPU code execution. The program in Listing 1 calculates Julia fractal set, which is a two-dimensional picture where each pixel is defined based on convergence or divergence of a corresponding sequence of complex numbers. The main part of the program requires considerable computational power and is executed on a GPU. All that a programmer has to do is extend the class *GPUKernel* (see line 1) to enable GPU-based code execution. *GPUKernel* is an abstract Java class, which contains two methods:

- *run* - an abstract method whose implementation must be provided in a subclass
- *start* - a final method which is implemented in *GPUKernel*

Method *run* is abstract and must be implemented by a subclass of *GPUKernel*. The code in the *run* method corresponds to the code executed on a GPU (a GPU kernel in CUDA terminology). The code contained in *run* method is executed in parallel by multiple GPU threads. Any call to other methods, called from *run* method, is also executed on a GPU. Another way of thinking about *run* method is to treat it by analogy to *run* method in classes extending *Thread*.

[3] http://vmkit.llvm.org/

```
1  public class Julia extends GPUKernel {
2
3    static Complex c=new Complex(-0.8, 0.156);
4    static int DIM=1000;
5    byte tab[]=new byte[DIM*DIM];
6
7    byte julia(int x, int y) {
8      double jx=(double)(DIM/2-x)/(DIM/2);
9      double jy=(double)(DIM/2-y)/(DIM/2);
10     Complex a=new Complex(jx, jy);
11     for (int i=0;i<255;i++) {
12       if (a.mul(a).add(c).magnitude2() >
             1000)
13         return i;
14     }
15     return 255;
16   }
17
18   public void run() {
19     int i=BlockId.x+BlockId.y*GridSize.x;
20     tab[i]=julia(BlockId.x, BlockId.y);
21    }
22
23   public static void main(String[] args) {
24     Julia m=new Julia();
25     m.start(new Dim(DIM, DIM), new Dim());
26   }
27 }
```

Listing 1. Program *O* - Julia set computation

Method *start* launches kernel execution on GPU. Arguments of the *start* method specifies the number of threads used to execute the kernel. These threads work in groups, which are called blocks. A grid element is comprised of multiple blocks. Size of a block can be defined not only as a single number, but also as a size of a rectangle or as a size of cuboid of threads. In such cases, threads will carry IDs which corresponds to their position within a workload. Method *start* in Julia example is shown at line 25 of listing 1. It launches kernel with a grid of size *DIM x DIM* blocks, and each block contains just one thread.

A major difference, which distinguishes JaBEE from Aparapi and GPU.NET[4], is its support for object operations. JaBEE can create objects inside of a GPU kernel. There is the object of a class *Complex* created in line 10, which is fully functional on GPU. Object *c* in line 3 is allocated on CPU and is used in the kernel in line 12. A JaBEE's GPU kernel can an object's virtual methods and can pass objects as arguments - regardless of whether the object is allocated on GPU or on CPU. For example, object *c* is allocated on CPU and in line 12 and is passed as an argument to method *add*. Dynamic dispatching is also supported within GPU kernel. Method *mul* is called in line 12 for object *a* which is allocated on GPU.

[4] http://www.tidepowerd.com/

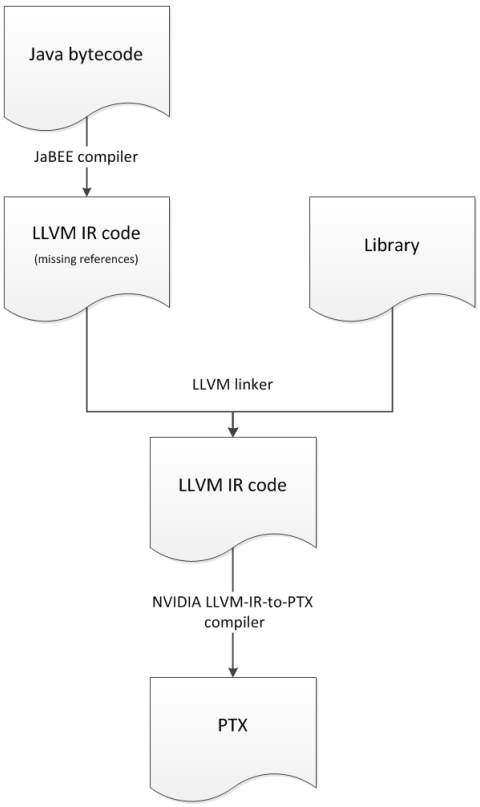

Figure 2: The flow of a compilation

4. Compilation

JaBEE performs on-line code compilation during runtime of an application. The compilation is done in two stages. First, the system transforms bytecode to LLVM IR using a modified Java bytecode compiler provided by VMKit. Then the LLVM IR is compiled into PTX ([17]), which is a virtual compute ISA for NVIDIA GPUs.

4.1 The Flow of a Compilation Process

The compilation flow is presented in Figure 2. During the first phase JaBEE system compiles Java bytecode into LLVM IR code. The generated LLVM IR code contains proper GPU specific address spaces and references GPU specific objects and methods.

JaBEE system links the generated LLVM IR code with a library that implements basic math functions and GPU specific methods using the LLVM linker *llvm-ld*. The linked code now is self-sufficient and does not have external references. JaBEE then passes the linked code to a NVIDIA implemented LLVM-IR-to-PTX compiler, which is responsible for generating the PTX code. The PTX code is ready to execute on a GPU.

VMKit J3 allows JNI methods to access internal data structures and call methods of a Virtual Machine. JaBEE compiles only a subset of classes that are loaded at the time

when the JNI call is made. JaBEE exploits access to J3's internal structures and can find loaded classes that need to be compiled in this step. Currently the JaBEE compiler and the virtual machines are two independent processes which do not share memory. All bytecode scheduled for compilation must be read directly from a file system. For the sake of implementation simplicity, JaBEE reads only these classes that are stored relatively to one root directory. For example, if JaBEE is requested to compile class *a.b.Foo* it will only attempt to read bytecode for this class from *<root_dir>/a/b/Foo.class* location. Compiler does not search entire application's classpath (JAR files). Classes from a standard library, JAR files and other classes which are not stored under the root directory cannot be used on GPU. This technical limitation is an implementation detail that can be easily mitigated in the future, e.g., by having the same process shared by *Compiler Unit* and other parts of the system.

4.2 J3 Specific Data Layout

JaBEE uses J3 as the JVM for bytecode execution on CPU. We initially wanted to make JaBEE VM-agnostic, meaning that Java code could be executed on any Java VM (e.g., Oracle's JVM or Kaffe). However, as our work of adding support for object oriented features continued it became clear that achieving this goal was more difficult than initially anticipated.

Compiler Unit generates code which internally represents a Java object as a data structure with references to other objects as pointers, and fields as values stored in a structure. JaBEE's compiler is based on VMKit, therefore code generated by JaBEE's compiler has the same representation of objects as the one used by VMKit J3 virtual machine when running Java programs. VMKit J3 virtual machine's in-memory objects are compatible with the definition of objects provided by the code generated by JaBEE's compiler. JaBEE transfers the original objects directly from virtual machine's memory to GPU, without altering their structure. However, some of values' pointers have to be transformed. Section 5 gives more details about this aspect. The way how an object is stored in a memory is not part of Java specification ([16]), therefore different virtual machines differ in terms of in-memory data layout. JaBEE can only support VMKit virtual machine as an execution environment since in-memory data layout is known for VMKit what is not the case for other Java virtual machines (e.g., Oracle VM).

```
1 %JavaObject = type {%VT addrspace(1)*, i8*}
2 %VT = type [0 x i32 (...)*]
```

Listing 2. Representation of Java data types in LLVM code

Listing 2 presents a fragment of a LLVM IR code, including the representation of Java Objects (line 1). Internal representation of JaBEE's and VMKit's Java object is a structure that consists of:

- Pointer to the virtual table followed by object-specific data
- Java Object's virtual table being an array of pointers to 32-bit integer data

This definition is specific to VMKit and other virtual machines might have different data layout for object representation.

4.3 Code Generation for GPGPU Address Spaces

NVIDIA's GPUs have their own hierarchical memory model, which is designed for accelerating hyper parallel operations rather than serial computation. PTX as well as enriched LLVM IR has mechanism called *address spaces* for informing the GPU where to allocate memory. Address spaces correspond to a storage area with varying characteristic such as: size, addressability, access speed, access rights, and level of sharing between threads. JaBEE compiler generates code with variables in the following address spaces:

- local
- global
- generic

Each variable has to define allocation address space. Generated LLVM IR code without address space definitions is invalid.

Local memory is private to a given thread. JaBEE generates LLVM code which allocates all local variables and temporary variables in the local address space. Compiler allocates local memory always statically.

All threads have access to the same global memory. Global PTX variables have to be stored in global memory space. Following information is stored in global variables:

- Virtual tables
- Static fields

The global memory space is persistent, thus virtual tables can be assigned once and their values do not have to be reassigned across kernel launches. Static fields are global to the entire application, therefore JaBEE stores static fields as PTX global variables.

Every address space has its own specific set of PTX instructions. For instance, load instruction for the global address space pointer *a* is:

```
%ld.global.u32   r1, a;
```

However, load instruction for the local address space pointer *b* is different:

```
%ld.local.u32   r2, b;
```

Attempt to execute instruction *ld.local.u32* performed on global variable would crash the application. PTX 2.0 offers a feature called generic address space, which unifies access to all address spaces including local and global. Any pointer

can be converted to generic address space pointer. Instruction

```
%ld.u32   r3, c;
```

performed on the generic pointer *c* is valid regardless of whether *c* points to local memory or global memory. JaBEE converts every pointer to generic pointer. References to objects are stored as pointers to generic address space - in LLVM code it is is marked by *addrspace(10)*. Method arguments are always generic address space pointers. For instance, LLVM code of *run* from Figure 5 shows that it takes as an argument generic pointer to Java Object. Converting all pointers to generic ones simplifies JaBEE code as compiler does not need to keep track of variables' memory location.

4.4 Support for GPU Specific Objects and Methods

Java bytecode is enriched with GPU specific methods and calls. Such enrichments allow Java bytecode instructions to become SIMD instructions (single instruction multiple data). References to fields and calls to methods listed below have special meaning in Java bytecode which corresponds to the GPU kernel:

- *ThreadId.x*, *ThreadId.y*, *ThreadId.z* return x-id, y-id, and z-id of a thread (id is coordinate in 3D space)
- *BlockId.x*, BlockId.y, BlockId.z return x-id, y-id, and z-id of the thread's block
- *BlockSize.x*, *BlockSize.y*, *BlockSize.z* return size of x, y, and z side of the block cuboid
- *GridSize.x*, *GridSize.y*, *GridSize.z* return size of x, y, and z side of the grid cuboid
- Call to function *GPUThread.syncThread()* synchronizes all threads in the block
- mathematical functions like *log*, *exp*

Fields *ThreadId.x*, *BlockId.x*, *BlockSize.x*, *GridSize.x* (similarly corresponding *.y* and *.z* fields) are transformed during compilation to corresponding calls to intrinsics, which are used further in PTX code generation. NVIDIA LLVM-to-PTX compiler replaces a call to intrinsics with references to special purpose registers *%tid.x*, *%ctaid.x*, *%ntid.x*, and *%nctaid.x* which correspond to thread id, block id, block size and grid size respectively. Figure 5 shows how code

```
int x=ThreadId.x
```

is first transformed to:

```
%1 = call i32 @llvm.nvvm.read.ptx.sreg.tid.x()
```

and further to:

```
mov.u32 %r2, %tid.x;
```

Method call *GPUThread.syncThread()* is transformed in the same way, however final result in PTX is a call to *bar.sync*, instead of an access to a register. All classes *ThreadId*, *BlockId*, *BlockSize*, and *GridSize* have static fields *x*, *y*, and *z*. *GPUThread* class has static method *syncThread*. Thus references to them can be placed anywhere in Java code keeping the code valid.

Some basic mathematical functions are implemented as a library. The implementation is highly optimized for GPU code execution and is very different from the one shipped with Java core libraries. Functions *log* and *exp* from *java.math* package have implementation which take a *double* as an argument type, but not *float*. Value of *float* passed as argument to *log* function is automatically converted to *double* and operation is performed on *double*. Operations on *double* are less efficient than operations performed on *float*. At this stage, JaBEE lacks the logic that would detect such automatic conversions and undo them in case of calls to mathematical functions leading to code operating on more specific types, e.g., using *float* instead of more general (and less efficient) *double*.

4.5 Dynamic Object Creation

NVIDIA architectures prior to Fermi (e.g., Tesla) do not support dynamic memory allocation. Dynamic memory allocation on Fermi graphics cards can be performed with CUDA *malloc* function call. Probably due to the lack of dynamic allocation in existing architectures (except the Fermi) there is no bytecode execution environment which would offer dynamic memory allocation. All high-level languages preallocate memory before calling a kernel.

For Java developers, dynamic allocation is very common technique. In Java, memory is dynamically allocated for every newly created object. However, as an optimization strategy Java virtual machines try to avoid dynamic allocation and use escape analysis to keep track of life time of an object ([2, 6, 20]) and in reality memory is often allocated statically. Both CPU and GPU allocate memory faster statically than dynamically. Support for objects is essential for JaBEE, so user can create object inside of GPU kernel. Internally there are two ways of handling object creation implemented by JaBEE:

- dynamic allocation (same mechanism as CUDA's *malloc*)
- static allocation on a stack

Control over which allocation is performed is done manually by environment variable manipulation. JaBEE does not implement escape analysis, which would choose appropriate allocation strategy depending on a variable life time. Memory allocation is called from the VMKit garbage collection allocation function *gcmalloc*, which is overridden by JaBEE implementation taken from library. However, current dynamic allocation with *malloc* is very experimental and was not well tested. We executed tests only on small data sets, which do not determine real performance penalty seen in real applications. Evaluation section presents results only with static memory allocation.

4.6 Virtual Table

Virtual table is a table of pointers to methods defined in a class. Every object contains a pointer to the virtual table. The following steps have to be performed in order to call object's method:

1. Read a pointer to the virtual table
2. Read a method pointer from the virtual table
3. Call a method returned from the virtual table

These steps are equivalent to a method call in a Java code. Bytecode transformed to LLVM code or to PTX code has to follow the steps mentioned above when calling a method. Figure 5 shows how code is transformed from bytecode via LLVM to PTX. Call to function *comput* is transformed to corresponding LLVM code, more specifically:

```
%9=getelementptr %VT addrspace(1)* %0,i32 0,i32 32
%10=load i32 (...)* addrspace(1)* %9, align 4
%11=bitcast i32 (...)* %10 to i32 (%JavaObject
    addrspace(1)*, i32)*
%12=call i32 %11(%JavaObject addrspace(1)* %0,i32 %6)
```

Register *%0* stores a reference to *this* Java object for all non-static methods. *JavaObject* structure carries all information about an object. Address to a method pointer is loaded to the register *%9*. Pointer *%9* points to an element of virtual memory (it is a pointer to a method pointer). Register *%10* is a pointer to a method to be called in the last line. LLVM code is then converted to PTX:

```
ldu.global.u32  %r5, [%r4+128];
{
call (retval0), %r5, (param0, param1), prototype_1;
ld.param.u32  %r6, [retval0+0];
...
}
```

Dynamic dispatch is less efficient than a function call, because of overhead of additional operations of look-up to virtual table. Java virtual machine developers already solved this issue, by so called de-virtualization ([3, 7, 11, 12]), however JaBEE does not implement any such mechanism. Code with in-line short non-recursive functions is more effective than with function calls. LLVM-to-PTX compiler is capable replace function call with in-lined code, but it is not able to in-line dynamic dispatch call. LLVM linker, which performs some simple optimizations, is also not able to optimize dynamic dispatch call. That might turn out to be serious performance penalty especially for calls to simple methods like getters or setters.

5. Data management

One of the important goals of JaBEE is to avoid user handling explicit data transfers, including transfer of Java objects, between CPU and GPU. Every time JaBEE's application launches GPU kernel data is copied between host and device. Our current solution takes a simplistic approach by bi-directionally transferring the entire object graph that is reachable from the kernel code. JaBEE copies:

1. *this* object which calls method *run*
2. All objects passed as parameters to kernel method
3. All static fields reachable from *run* method
4. Recursively all objects which are pointed at by already copied objects

The entire object graph is copied back from GPU memory to CPU memory once GPU code execution is completed. JaBEE allows for object graph modification on the GPU. That means that elements of an object graph and relations between them can be modified by code executed on the GPU and after copying back the relation will be preserved. It is achieved by traversing object graph modified on GPU while copying it back to the CPU memory.

Operating system and GPU driver are responsible for memory allocation on CPU and GPU respectively. There are no mechanisms to control address of memory returned from allocation function from user space (JaBEE is user space application) both for CPU and GPU memory. JaBEE stores the same objects both in RAM and in the GPU memory, however their memory addresses do not coincide. Every time when JaBEE copies data between host and device it slightly modifies the data by changing :

1. CPU pointers to Java object - to corresponding GPU pointers and vice-versa
2. CPU virtual table pointer - to corresponding GPU pointers and vice-versa

Pointers to Java objects represent references in Java code. Every object contains virtual table pointer. JaBEE changes all pointers in the application pointing to CPU locations corresponding to GPU locations. The same reverse operation of changing pointers is performed when objects are copied back.

6. Evaluation

It has been the primary goal of the first release of JaBEE platform to deliver a prototype and provide a proof-of-concept solution. This research has not focused so far on delivering a high-performance application, which would be already a fully optimized solution. There are number of optimizations techniques presented in the future work in section 8, which are going to be investigated along as this project develops. The following section estimates constraints for objects support and discusses the importance of optimization, when executing object oriented applications.

A set of tests were performed on two equivalent 'Julia' applications called *P* and *O* (listing 1). While both are almost equivalents, the only difference between them is the fact that in the first case *P*rimitive types are used as a primary type in GPU the kernel, while in the second example - *O*bjects. To conduct this evaluation, programs *P* and *O* were rewritten to pure Java and CUDA, and executed on (1) VMKit virtual

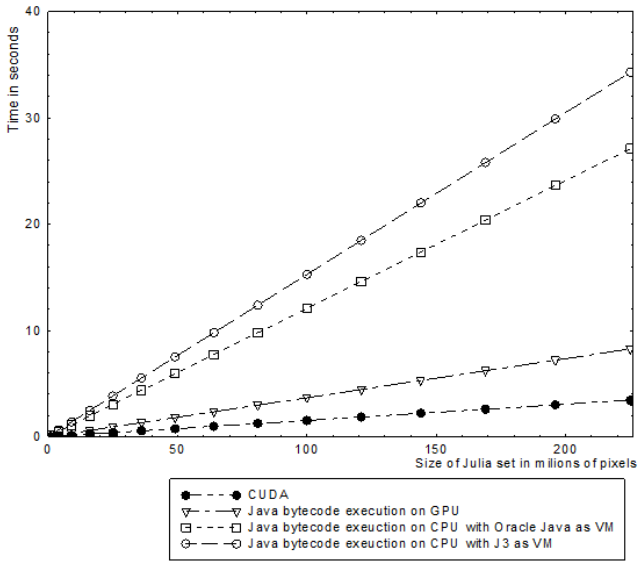

Figure 3: Program *P* execution time evaluation of Julia program, which uses only primitive types

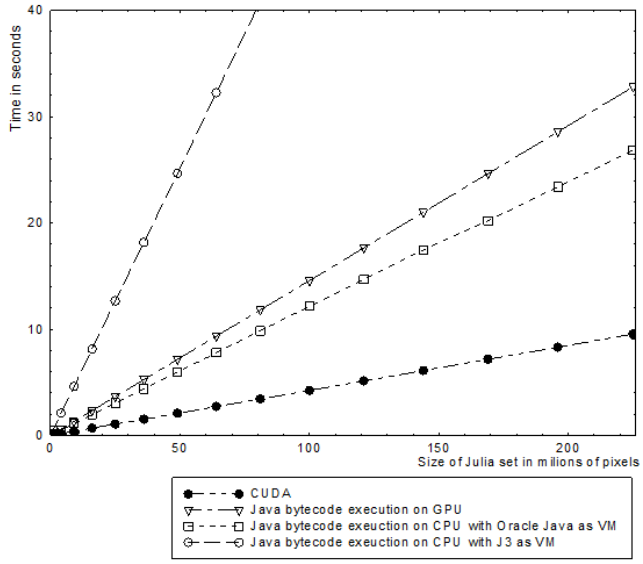

Figure 4: Program *O* execution time evaluation of Julia program, which uses operations on objects

machine, (2) Oracle virtual machine and (3) CUDA system. All tests were executed on a 32-bit Fedora system with 8 core Intel(R) Core(TM) i7 CPU machine with 6 GB of RAM and with GTX 480 NVIDIA graphics card.

The evaluation of *P* program 3 shows that the shortest execution time is achieved with CUDA platform, followed by JaBEE system, then Oracle Java virtual machine and finally the VMKit. The VMKit and the JaBEE internally execute a very similar code, because both of them use the same compiler engine to generate LLVM code out of a bytecode. The speed-up ratio between JaBEE and VMKit shows the performance boost achieved by changing the execution processor from a CPU to a GPU. The timing for the Oracle virtual machine (the fastest VM on the market) indicates the excellent optimization for a Java bytecode. The performance of the CUDA code execution shows the lower bound for the JaBEE possible optimizations.

The second experiment was performed on Julia code called *O*, which uses objects to represent complex numbers (complex numbers correspond to *Complex* class object in listing 1). The Figure 4 shows results of this evaluation, on which the best result has been achieved in CUDA environment, then is the Oracle virtual machine, followed by the JaBEE and finally the VMKit. Usage of objects instead of primitives results in increased execution time for all platforms except Oracle virtual machine. The time ratio between different configurations for programs *P* and *O* is constant for sufficiently big data sets (over 10 million pixels). Table 1 shows the performance ratio for big data sets for all applications, when comparing to the execution time of the program *P* on the VMKit virtual machine:

	J3	**Oracle Java**	**JaBEE**	**CUDA**
Program *P*	1	1.26	4.15	9.97
Program *O*	0.30	1.27	1.04	3.59

Table 1. Speed up ratios in compare to program *P* execution time on VMKit virtual machine

JaBEE keeps the speed-up of about four times in comparison to VMKit virtual machine just by code execution on a GPU instead of a CPU. Oracle Java (table 1) shows that it is possible to optimize operations on objects so good that performance results can be the same for object data types as for primitives. Java Oracle achieves good performance by both static and dynamic optimizations (virtual machines use JIT). JaBEE contains online compiler and it is out of scope of this project to build-in JIT compiler (because of the complexity), which means that only static optimizations are available for the system. The overview of generated PTX and LLVM IR for programs *P* and *O* shows that de-virtualization would speed up program *O*. Methods like *add* and *mul* for *Complex* object are very simple and de-virtualization of them would allow LLVM-IR-to-PTX compiler to in-line such methods. Assembly code generated with de-virtualization for program *O* would be very similar to current final code of the program *P*, so performance would be also alike.

7. Related Work

There are a number of related approaches that attempt to extend Java programming model with GPU-specific constructs. JCuda [21] is a Java binding for CUDA, which translate CUDA methods into Java methods and introduces Java

objects that are CUDA specific e.g. it introduces device *Pointer* object, *cudaMalloc* method for memory allocation or *cudaFree* for memory freeing. Call to such Java method invokes corresponding CUDA method, e.g., call to JCuda method *cudaMalloc* invokes JNI native code, which executes CUDA function *cudaMalloc*. JCuda approach does not raise the level of abstraction of CUDA. JavaCL offers similar, low-level solution for OpenCL.

GPU.NET for .NET and Aparapi for Java are closely related to the JaBEE approach. GPU.NET performs a runtime compilation of .NET bytecode (CLI) into PTX. Similarly to JaBEE, GPU.NET uses an online compiler for code generation. Then, it copies all data to GPU, launch kernel and copies data back, after the computation is completed. For GPU code execution GPU.NET supports operations on primitives and arrays of primitives, but not on objects. Microsoft .NET platform is not an open source and there is no information about in-memory data representation. GPU.NET can get an access to objects only through P/Invoke (.NET equivalent of JNI), however it requires multiple P/Invoke for objects with multiple fields, which is inefficient.

The Aparapi, as well as the JaBEE systems are built as JNI binding to Java. The Aparapi gets an access to an object field address in the memory by a low level routines provided by the class *sun.misc.Unsafe*. The Aparapi infrastructure allows transferring arrays of objects between a host and a device, however objects on GPU do not keep all their properties. Objects on GPU are only data containers in this approach and only methods which can be called on them are their respective getters and setters. There is no access through references to other objects on GPU from such a data container, which gives access only to the primitive data. Aparapi does not allow creating objects within the kernel, which is possible for JaBEE. Therefore, it is currently out of scope for Aparapi to support Java core library classes such as *ArrayList*, *HashSet*, *SortedMap* etc. JaBEE does not compile system libraries, however this limitation is not hard to bypass (see section A). Aparapi bounds compilation scope to class, which contains an entry to a GPU kernel. In the contrast, JaBEE allows for kernel execution from a bytecode collected from multiple classes. Aparapi requires compiling Java classes with debug flag, because it makes use of debug information, while JaBEE works with pure Java bytecode without any debug code.

Aparapi targets OpenCL instead of PTX and therefore the generated code is more restrictive, than in the case of JaBEE. More specifically Aparapi **does not** support :

1. dynamic dispatch,
2. recursion,
3. primitive type double,
4. object allocation on GPU.

These are probably due limitations in OpenCL and/or AMD's GPU hardware.

8. Future Work

The research, which has been conducted so far on JaBEE showed a set of considerable benefits, when using object execution on GPUs. Still, we touched in the context of this work just an "ice berg", as many topics were not addressed or investigated at all. The following set of activities is foreseen to be a part of the future research work:

1. Compiler integration with the remaining part of the system and direct access to bytecode through memory
2. Support for shared memory (there is a fast memory region available on GPU that can be shared among threads running in a block)
3. Compilation optimizations (known optimizations for the Java static compiler)
   - De-virtualization
   - Stack allocation instead of heap allocation
4. Garbage collection on GPU
5. More efficient data management
6. Implementation of VM on GPU

The current approach of separating the *Compilation unit* from the main system has several negative implications. For example it is hard to access bytecode of Java core libraries. Close integration of the compiler would resolve this issue. Partial support for shared memory was implemented, however it has not yet been well tested. Shared memory should be used to achieve performance boost in many GPU specific algorithms (e.g., dot product). Implementation of the compilation optimization for GPU execution would result in a considerable improvement in the performance. Finally, other enhancements such as garbage collection, more efficient data management and VM implementation on GPU are unexplored, and still promising research areas to be conducted in the context of JaBEE.

9. Conclusions

JaBEE research undertook a challenging topic on how to provide a support for objects operations in a GPU kernel and laid down a foundation for object oriented programming on GPU for Java. This initial approach proposes a hybrid model allowing a selective bytecode execution on both CPUs and GPUs.

This paper and the presented research addresses in the new way a set of following aspects:

1. the creation of objects on a GPU,
2. the transparent objects sharing between GPU and CPU code,
3. the dynamic dispatch on GPU.

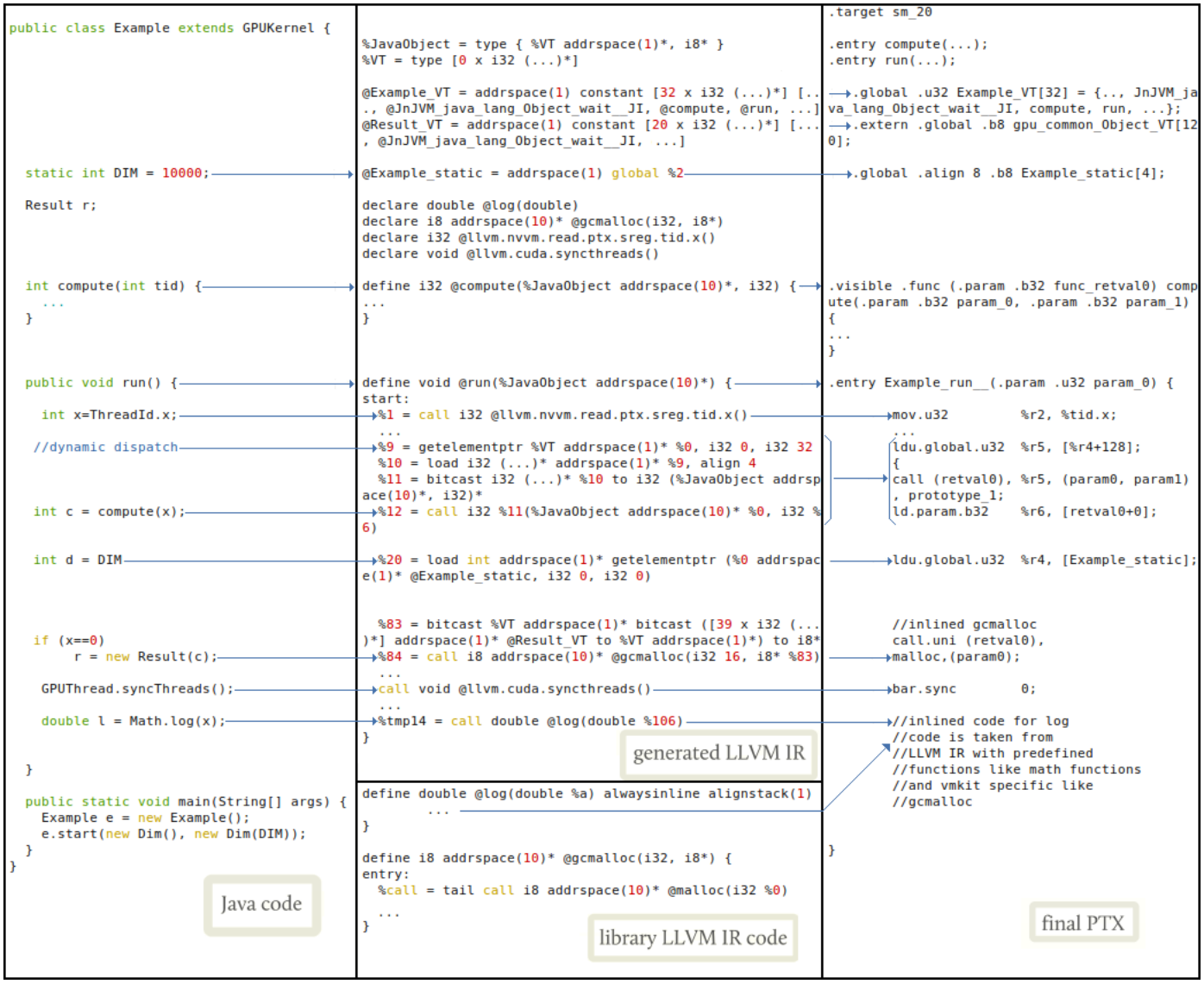

Figure 5: The flow of code transformations

Two approaches have been investigated: (1) a GPU-based computing using objects, and (2) a classical GPU-based computing approach using primitive data types. Although in this initial work, we have experienced some issues related to performance degradation, we have also proposed several optimizations techniques, where through the devirtualization and the local scope of a memory allocation, the performance problems could be addressed. Our research is a proof-of-concept, which proposes a new approach for the GPU computing for Java. Our approach allows Java developers to benefit from GPU-based code execution without dealing with low-level details of GPU-specific programming environments, such as CUDA or OpenCL.

A. Appendix The Flow of the Code Transformation

The Figure 5 presents stages of the code transformation, but it omits Java bytecode to improve its readability (there is a corresponding Java code included). First, JaBEE system compiles a Java bytecode to LLVM IR. The generated LLVM IR code has to fulfill several constraints (details have been presented in the section 4). The generated file is not yet ready for the execution since the following references are missing:

1. VMKit specific methods - gcmalloc (a call to garbage collector to allocate memory), getVT (an operation obtaining virtual table of object) and several others,

2. Mathematical operations - optimized GPU execution implementation instead of core Java libraries implementation

These missing methods are implemented in the LLVM IR library file, which JaBEE links with the generated file through the LLVM linker (*llvm-ld*). Finally, the generated file is passed to the NVIDIA LLVM-IR-to-PTX compiler, which creates PTX file to be executed on the GPU.

References

[1] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC'09*, pages –1–1, 2009.

[2] B. Blanchet. Escape analysis for javatm: Theory and practice. *ACM Trans. Program. Lang. Syst.*, 25(6):713–775, 2003. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/945885.945886.

[3] B. Calder and D. Grunwald. Reducing indirect function call overhead in c++ programs. In *POPL*, pages 397–408, 1994. URL `http://dblp.uni-trier.de/db/conf/popl/popl94.html#CalderG94`.

[4] O. Chafik. *ScalaCL: Faster Scala: optimizing compiler plugin + GPU-based collections*, Sept. 2011. URL `http://code.google.com/p/scalacl/`.

[5] M. M. T. Chakravarty, G. Keller, T. L. M. S. Lee, and V. Grover. Accelerating Haskell array codes with multicore GPUs. In *Declarative Aspects of Multicore Programming 2011*, 2011.

[6] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Escape analysis for java. In *OOPSLA'99*, pages 1–19, 1999.

[7] D. Detlefs and O. Agesen. Inlining of virtual methods. In *ECOOP'99*, pages 258–278, 1999.

[8] N. Geoffray. *Fostering Systems Research with Managed Runtimes*. PhD thesis, PhD thesis, Université Pierre et Marie Curie, Paris, France, September 2009.

[9] N. Geoffray, G. Thomas, J.Lawall, G. Muller, and B. Folliot. VMKit: a Substrate for Managed Runtime Environments. In *Virtual Execution Environment Conference (VEE 2010)*, Pittsburgh, USA, March 2010. ACM Press.

[10] K. Group. *The OpenCL Specification*, Oct. 2009. URL `http://www.khronos.org/registry/cl/specs/opencl-1.0.pdf`.

[11] D. Grove, J. Dean, C. Garrett, and C. Chambers. Profile-guided receiver class prediction. In *OOPSLA'95*, pages 108–123, 1995.

[12] 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, 2000.

[13] A. Klckner. *PyCUDA: CUDA parallel computation API from Python*, Sept. 2010. URL `http://mathema.tician.de/software/pycuda`.

[14] A. Klckner. *PyOpenCL - Python programming environment for OpenCL*, Sept. 2011. URL `http://mathema.tician.de/software/pyopencl`.

[15] J. Kong, M. Dimitrov, Y. Yang, J. Liyanage, L. Cao, J. Staples, M. Mantor, and H. Zhou. Accelerating matlab image processing toolbox functions on gpus. In *GPGPU'10*, pages 75–85, 2010.

[16] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999. ISBN 0201432943.

[17] NVIDIA. *PTX: Parallel Thread Execution ISA Version 2.0*, Jan. 2010. URL `http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/ptx_isa_2.0.pdf`.

[18] NVIDIA. *CUDA - Compute Unified Device Architecture*, Sept. 2011. URL `http://developer.nvidia.com/category/zone/cuda-zone`.

[19] N. Nystrom, D. White, and K. Das. Firepile: run-time compilation for gpus in scala. In *GPCE'11*, 2011.

[20] J. Whaley and M. Rinard. Compositional pointer and escape analysis for java programs. In *In Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 187–206. ACM Press, 1999.

[21] Y. Yan, M. Grossman, and V. Sarkar. Jcuda: A programmer-friendly interface for accelerating java programs with cuda. In *Euro-Par'09*, pages 887–899, 2009.