# **How the lambdas are represented in Java bytecode.**

Traditionally for Aparapi we ask the user to inherit from a base class **com.amd.aparapi.Kernel** and override the **public void Kernel.run()** method.

**class Squarer extends Kernel{**

**float in[], squares[];**

**@Override void run(){**

**Int gid = getGlobalId();**

**Squares[gid] = in[gid]\*in[gid];**

**}**

**float[] square(float\_in[]){**

**in = \_in;**

**squares = new float[in.length];**

**execute(in.length);**

**return(squares);**

**}**

**}**

This allows the user to use code such as:-

**Squarer squarer= new Squarer();**

**float in[] = // allocation and initialization of in[] ommitted**

**float squares = squarer.square(in);**

We often demonstrate (due to brevity) Aparapi using the anonymous inner class style of use.

**Squarer squarer= new Squarer();**

**final float in[] = // allocation and initialization of in[] omitted**

**final float squares[] = new float[in.length];**

**Kernel squarer = new Kernel(){**

**@Override void run(){**

**Int gid = getGlobalId();**

**squares[gid] = in[gid]\*in[gid];**

**}**

**};**

**squarer.execute(in.length);**

This works because Java creates a synthetic anonymous inner class which actually would look something like this (if we had source) :

**class Main$1 extends Kernel{**

**final float in[], squares[];**

**void Main$1 (float \_in[], float \_squares[]){**

**in = \_in;**

**squares = \_squares;**

**}**

**@Override void run(){**

**Int gid = getGlobalId();**

**squares[gid] = in[gid]\*in[gid];**

**}**

**}**

The compiler basically injects the constructor call to **Main$1(float[], float[])** into the bytecode, as if the user had declared and initiated this synthetic constructor. Note that these **float[]** args are ‘captured’ references from the call site that are needed by the run method (actually if there were more methods requiring other captured data, those references would also be passed to the constructor – the constructor args list is the superset of all captures from all declared methods of the interface ).

It is as if the code above :-

**Kernel squarer = new Kernel(){**

**@Override void run(){**

**Int gid = getGlobalId();**

**squares[gid] = in[gid]\*in[gid];**

**}**

**};**

**squarer.execute(in.length);**

Were replaced by :-

**Kernel squarer = new Main$1(squares, in);**

**squarer.execute(in.length);**

So anonymous inner classes just work with Aparapi, mainly because the actual observed bytecode looks just like the code we would write if we used a \*real\* declared class ;)

In Java 8 we will see lambda functions introduced as first class citizens. See <http://openjdk.java.net/projects/lambda/>

In many ways lambda functions behave like anonymous inner classes, except the user does not have to define the class at the call site.

The idea is that if I want to assign to something defined as type **java.lang.Runnable**, I really should not need to name the interface type.

**Runnable runnable = new Runnable(){**

**@Override void run(){**

**/\* code for run method goes here \*/**

**}**

**};**

Instead I should be able to provide the code for the **Runnable.run()** method (and as we will see later we may also need to supply arg bindings).

**Runnable runnable= {/\* code for run method goes here \*/};**

The compiler \*knows\* that I need to create something of type **java.lang.Runnable** and \*knows\* that this interface has a single method called **Runnable.run()**. So why do I need to add all the extraneous text?

The actual syntax for expressing java.lang.Runnable as a lambda is actually this:-

**Runnable runnable= ()->{/\* code for run method goes here \*/};**

The empty parenthesis means we need to match a method which takes zero args (**Runtime.run()** does indeed take takes zero args) and the ‘->’ denotes that the code enclosed in the next {…} is the body of the method.

Any interface which has a Single Abstract Method is referred to as a SAM type. Any SAM type can be used in place of a ‘long-hand’ form of declaration.

So if we write an interface called

**interface FooAble{**

**public void foo();**

**}**

Then we can indeed use the lambda form of assignment

**FooAble fooAble = ()->{System.out.println(“foo!”);}**

And can call the foo method using:

**fooAble.foo();**

Note also that if the interface defines any args

**interface FooAble{**

**public void foos(int count);**

**}**

Then these args (but not the type, the compiler can infer the type from the interface declaration) need to be present inside the ‘()’ and can be used inside the lambda body, just like a normal method arg.

**FooAble fooAble = (count)->{System.out.println(count+“ foos!”);}**

And then the call to foo needs to take an int value for count.

**fooAble.foo(12);**

This is useful for all sorts of things. Specifically in Swing where we have lots of SAM types for listeners.

**JButton button = new JButton(“press me”);**

**Button.addActionListener( new ActionListener(){**

**@Override void actionPerformed(ActionEvent ae){**

**System.out.println(“Yeah!”);**

**}**

**});**

Now we can use :-

**JButton button = new JButton(“press me”);**

**Button.addActionListener((ae)->{System.out.println(“Yeah!”});**

Note here that the interface ActionListener’s actionPerformed() method takes a single ActionEvent arg. So its lambda form has a named variable inside the parenthesis (ae). You will note that we don’t have to define the arg type, from the ActionListener declaration the compiler \*knows\* that the type is ActionEvent. We just need to give the compiler a name to bind the arg value type to.

Lamba’s are also very similar anonymous inner classes, in that they can capture state from the call site.

So if I wanted to launch a thread which printed a String using anonymous inner classes I could use

**final String name = “John Lennon”;**

**new Thread(new Runnable(){**

**@Override public void run(){**

**System.out.println(name);**

**}).start();**

For anonymous inner classes the variable **name** is ‘final’ so it can be captured by the run method of the anonymous inner class (we saw earlier that a synthetic constructor is used to copy the value into the newly created class). If we neglected to declare ‘name’ as final the compiler would fail to compile this code.

Here is the same code using a lambda implementation of the Runnable SAM type.

**final String name = “John Lennon”;**

**new Thread(()->{System.out.println(name);}).start();**

So the name variable is available for use inside the method body.

Note for lambdas the rules are relaxed a little, the captured string only has to be ‘effectively final’. Effectively final means that the compiler can prove that the variable cannot be changed.

For example in the following code we can forgo the final declaration, because the scope of name dictates that name will never be changed.

**{**

**String name = “John Lennon”;**

**new Thread(()->{System.out.println(name);}).start();**

**}**

Lambdas and anonymous inner classes are so similar in fact that Project Lambda first implemented lambdas by creating synthetic anonymous inner classes. This allows the project to quickly get an implementation in place, with minimal changes to the JVM.

The \*production\* version of lambdas will not use this approach, instead (as we see below) they will use the new invokedynamic instruction and some boostrap factory classes to implement the lambda dispatch.

So let’s look at the bytecode looks like for the code above.

First here is the complete code

**package com.amd.aparapi.samples;**

**public class RunTest{**

**public static void main(String[] args){**

**String name = "John Lennon";**

**new Thread(()->{System.out.println(name);}).start();**

**}**

**}**

Here is the bytecode from javap for the static **RunTest.main()** method .

**public static void main(java.lang.String[]);**

**0: ldc #2 // String John Lennon**

**2: astore\_1**

**3: new #3 // class java/lang/Thread**

**6: dup**

**7: aload\_1**

**8: invokedynamic #4, 0 // InvokeDynamic #0:lambda:(String)Runnable**

**13: invokespecial #5 // Method Thread.<init>:(Runnable)V**

**16: invokevirtual #6 // Method Thread.start:()V**

**19: return**

In the same classfile we will also see a new synthetic method which contains the bytecode of the lambda body.

**private static void lambda$3(java.lang.String);**

**0: getstatic #7 // Field System.out:PrintStream**

**3: aload\_0**

**4: invokevirtual #8 // Method PrintStream.println:(String)V**

**7: return**

Even though the method that the lambda is implementing takes zero arguments, the synthetic lambda method takes a single String. This is how the lambda deals with the captured String from the call site. Basically the callsite is passing the ‘effectively final’ String to the synthetic lambda method. Even though we are providing an implementation for **Runnable.run()** the method actually turns into a boring plain old method (albeit synthetic with a weird name) .

This implies that for each callsite, a new method will be created, each with arg lists (signatures) dependant on the captures. So if we had

**package com.amd.aparapi.samples;**

**public class RunTest{**

**public static void main(String[] args){**

**String name = "John Lennon";**

**new Thread(()->{System.out.println(name);}).start();**

**int couint = 5;**

**new Thread(()->{System.out.println(count);}).start();**

**}**

**}**

We would get two distinct synthetic lambda implementation. Presumably :-

**private static void lambda$3(java.lang.String);**

**0: getstatic #7 // Field System.out:PrintStream**

**3: aload\_0**

**4: invokevirtual #8 // Method PrintStream.println:(String)V**

**7: return**

**private static void lambda$4(int);**

**0: getstatic #7 // Field System.out:PrintStream**

**3: aload\_0**

**4: invokevirtual #8 // Method PrintStream.println:(int)V**

**7: return**

Going back to the call site we see that **invokedynamic** bytecode somehow hooks together the argument to **java.lang.Thread**’s constructor (expecting a **java.lang.Runnable**) to the **lambda$3(String)** implementation.

**0: ldc #2 // String John Lennon**

**2: astore\_1**

**3: new #3 // class java/lang/Thread**

**6: dup**

**7: aload\_1**

**8: invokedynamic #4, 0 // InvokeDynamic #0:lambda:(String)Runnable**

However it is not obvious how all this gets connected.

From the above bytecode we can ignore the greyed out code (just prepping the java.lang.Thread constructor using the result of invoke dynamic), the red bytecode is arranging for the String “John Lennon” to be pushed on the stack to be consumed by invokedynamic.

So a simpler form (pseudo code) would be:-

**XX: push “John Lennon”**

**8: invokedynamic #4, 0 // InvokeDynamic #0:lambda:(String)Runnable**

So the invoke dynamic call somehow is causing “John Lennon” to be passed to **lambda$3(String)**

It does this not by calling the lambda$3(String) method directly but by calling a boobstrap method which has also been added to the classfile for RunTest.

The invokedynamic bytecode takes two operands (#4 and 0). The first is a reference into the constant pool and the second (curiously is always 0).

In the constant pool slot #4 (see appendix for complete constant pool listing)

We see the following constant pool entry

#4 = InvokeDynamic #0:#36 // #0:lambda:(Ljava/lang/String;)Ljava/lang/Runnable;

<http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.4.10>

The InvokeDynamic constant pool entry is a new constant pool entry (new in Java 7) and it itself has two operands (#0 and #36). The #0 refers to slot #0 in the bootstrap methods table of this class (the bootstrap methods table is a new Attribute added to the class in Java 7), whereas the #36 refers to a NameAndType entry in the constant pool.

Let’s look at the boostrap method table first

<http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.7.21>

The method at slot 0 in the bootstrap method table is defined to be:-

#32 invokestatic java/lang/invoke/LambdaMetafactory.metaFactory:(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodHandle;Ljava/lang/invoke/MethodHandle;Ljava/lang/invoke/MethodType;)Ljava/lang/invoke/CallSite;

Method arguments:

#33 invokeinterface java/lang/Runnable.run:()V

#34 invokestatic com/amd/aparapi/samples/RunTest.lambda$3:(Ljava/lang/String;)V

#35 ()V

So to decode this, it is a call to

**class java.lang.invoke.LambdaMetafactory{**

**static java.lang.invoke.CallSite metaFactory(**

**java.lang.invoke.MethodHandles.Lookup arg1,**

**String arg2,**

**java.lang.invoke.MethodType arg4,**

**java.lang.invoke.MethodHandle arg5,**

**java.lang.invoke.MethodHandle arg6,**

**java.lang.invoke.MethodType arg7){}**

**}**

**From** <http://www.dalorzo.com/jdk8/javadocs/java/lang/invoke/LambdaMetafactory.html> we get this API description

public static [CallSite](http://www.dalorzo.com/jdk8/javadocs/java/lang/invoke/CallSite.html" \o "class in java.lang.invoke) [metaFactory](http://www.dalorzo.com/jdk8/javadocs/src-html/java/lang/invoke/LambdaMetafactory.html" \l "line.176)([MethodHandles.Lookup](http://www.dalorzo.com/jdk8/javadocs/java/lang/invoke/MethodHandles.Lookup.html" \o "class in java.lang.invoke) caller,

[String](http://www.dalorzo.com/jdk8/javadocs/java/lang/String.html) invokedName,

[MethodType](http://www.dalorzo.com/jdk8/javadocs/java/lang/invoke/MethodType.html) invokedType,

[MethodHandle](http://www.dalorzo.com/jdk8/javadocs/java/lang/invoke/MethodHandle.html) samMethod,

[MethodHandle](http://www.dalorzo.com/jdk8/javadocs/java/lang/invoke/MethodHandle.html) implMethod,

[MethodType](http://www.dalorzo.com/jdk8/javadocs/java/lang/invoke/MethodType.html) instantiatedMethodType)

Standard meta-factory for conversion of lambda expressions or method references to functional interfaces.

****Parameters:****

caller - Stacked automatically by VM; represents a lookup context with the accessibility privileges of the caller.

invokedName - Stacked automatically by VM; the name of the invoked method as it appears at the call site. Currently unused.

invokedType - Stacked automatically by VM; the signature of the invoked method, which includes the expected static type of the returned lambda object, and the static types of the captured arguments for the lambda. In the event that the implementation method is an instance method, the first argument in the invocation signature will correspond to the receiver.

samMethod - The primary method in the functional interface to which the lambda or method reference is being converted, represented as a method handle.

implMethod - The implementation method which should be called (with suitable adaptation of argument types, return types, and adjustment for captured arguments) when methods of the resulting functional interface instance are invoked.

instantiatedMethodType - The signature of the SAM method from the functional interface's perspective

****Returns:****

a CallSite, which, when invoked, will return an instance of the functional interface

Essentially invokedyamic calls a defined bootstrap method which will create a method handle which links this call-site to the synthetic lambda$3 method.

Still trying to work out how to capture all this. TODO:!!!!!

# Here is how I think we might introduce Lambdas to Aparapi

During our initial experiments we will introduce the following capability to Aparapi.

**Aparapi.forEach(<range>, (gid)->{ squares[gid]=in[gid]\*in[gid];}**

**);**

To make this work I creates an **Aparapi** helper with inner **Aparapi.SAM(int)** type and a single threaded implementation of **Aparapi.forEach()** which takes a range and an implementation **Aparapi.SAM(int)**.

**public class Aparapi{**

**public interface SAM{**

**void run(int gid);**

**}**

**public void forEach(int range, SAM sam){**

**for (int i=0; i<range; i++){**

**sam.run(i);**

**}**

**}**

**}**

And everything works magically!

Here is my test code

**public class Main{**

**public static void main(String[] args){**

**final int in[] = new int[100];**

**final int squares[] = new int[100];**

**// fill in[]**

**Aparapi.forEach(in.length,**

**(gid)->{ squares[gid]=in[gid]\*in[gid];}**

**);**

**// use squares[]**

**}**

**}**

I pass a range (say 100) and a lambda implementation of my Aparapi.SAM type and the Aparapi.SAM.run(int gid) method is indeed executed 100 times , each with a unique value of gid.

For execution on the GPU we will need to grab the bytecode of the run method of the SAM type (which will be a synthetic method on the Main class so Aparapi can convert it to OpenCL. We also need to examine the captures (embedded in the lambda signature!) so we can do some buffer magic and execute on the GPU.

Links/References

<http://openjdk.java.net/projects/lambda/>

<http://cr.openjdk.java.net/~briangoetz/lambda/lambda-translation.html>

public class com.amd.aparapi.samples.RunTest

SourceFile: "RunTest.java"

InnerClasses:

public static final #65= #64 of #70; //Lookup=class java/lang/invoke/MethodHandles$Lookup of class java/lang/invoke/MethodHandles

BootstrapMethods:

0: #32 invokestatic java/lang/invoke/LambdaMetafactory.metaFactory:(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodHandle;Ljava/lang/invoke/MethodHandle;Ljava/lang/invoke/MethodType;)Ljava/lang/invoke/CallSite;

Method arguments:

#33 invokeinterface java/lang/Runnable.run:()V

#34 invokestatic com/amd/aparapi/samples/RunTest.lambda$3:(Ljava/lang/String;)V

#35 ()V

minor version: 0

major version: 51

flags: ACC\_PUBLIC, ACC\_SUPER

Constant pool:

#1 = Methodref #10.#28 // java/lang/Object."<init>":()V

#2 = String #29 // John Lennon

#3 = Class #30 // java/lang/Thread

#4 = InvokeDynamic #0:#36 // #0:lambda:(Ljava/lang/String;)Ljava/lang/Runnable;

#5 = Methodref #3.#37 // java/lang/Thread."<init>":(Ljava/lang/Runnable;)V

#6 = Methodref #3.#38 // java/lang/Thread.start:()V

#7 = Fieldref #39.#40 // java/lang/System.out:Ljava/io/PrintStream;

#8 = Methodref #41.#42 // java/io/PrintStream.println:(Ljava/lang/String;)V

#9 = Class #43 // com/amd/aparapi/samples/RunTest

#10 = Class #44 // java/lang/Object

#11 = Utf8 <init>

#12 = Utf8 ()V

#13 = Utf8 Code

#14 = Utf8 LineNumberTable

#15 = Utf8 LocalVariableTable

#16 = Utf8 this

#17 = Utf8 Lcom/amd/aparapi/samples/RunTest;

#18 = Utf8 main

#19 = Utf8 ([Ljava/lang/String;)V

#20 = Utf8 args

#21 = Utf8 [Ljava/lang/String;

#22 = Utf8 name

#23 = Utf8 Ljava/lang/String;

#24 = Utf8 lambda$3

#25 = Utf8 (Ljava/lang/String;)V

#26 = Utf8 SourceFile

#27 = Utf8 RunTest.java

#28 = NameAndType #11:#12 // "<init>":()V

#29 = Utf8 John Lennon

#30 = Utf8 java/lang/Thread

#31 = Utf8 BootstrapMethods

#32 = MethodHandle #6:#45 // invokestatic java/lang/invoke/LambdaMetafactory.metaFactory:(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodHandle;Ljava/lang/invoke/MethodHandle;Ljava/lang/invoke/MethodType;)Ljava/lang/invoke/CallSite;

#33 = MethodHandle #9:#46 // invokeinterface java/lang/Runnable.run:()V

#34 = MethodHandle #6:#47 // invokestatic com/amd/aparapi/samples/RunTest.lambda$3:(Ljava/lang/String;)V

#35 = MethodType #12 // ()V

#36 = NameAndType #48:#49 // lambda:(Ljava/lang/String;)Ljava/lang/Runnable;

#37 = NameAndType #11:#50 // "<init>":(Ljava/lang/Runnable;)V

#38 = NameAndType #51:#12 // start:()V

#39 = Class #52 // java/lang/System

#40 = NameAndType #53:#54 // out:Ljava/io/PrintStream;

#41 = Class #55 // java/io/PrintStream

#42 = NameAndType #56:#25 // println:(Ljava/lang/String;)V

#43 = Utf8 com/amd/aparapi/samples/RunTest

#44 = Utf8 java/lang/Object

#45 = Methodref #57.#58 // java/lang/invoke/LambdaMetafactory.metaFactory:(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodHandle;Ljava/lang/invoke/MethodHandle;Ljava/lang/invoke/MethodType;)Ljava/lang/invoke/CallSite;

#46 = InterfaceMethodref #59.#60 // java/lang/Runnable.run:()V

#47 = Methodref #9.#61 // com/amd/aparapi/samples/RunTest.lambda$3:(Ljava/lang/String;)V

#48 = Utf8 lambda

#49 = Utf8 (Ljava/lang/String;)Ljava/lang/Runnable;

#50 = Utf8 (Ljava/lang/Runnable;)V

#51 = Utf8 start

#52 = Utf8 java/lang/System

#53 = Utf8 out

#54 = Utf8 Ljava/io/PrintStream;

#55 = Utf8 java/io/PrintStream

#56 = Utf8 println

#57 = Class #62 // java/lang/invoke/LambdaMetafactory

#58 = NameAndType #63:#67 // metaFactory:(Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodHandle;Ljava/lang/invoke/MethodHandle;Ljava/lang/invoke/MethodType;)Ljava/lang/invoke/CallSite;

#59 = Class #68 // java/lang/Runnable

#60 = NameAndType #69:#12 // run:()V

#61 = NameAndType #24:#25 // lambda$3:(Ljava/lang/String;)V

#62 = Utf8 java/lang/invoke/LambdaMetafactory

#63 = Utf8 metaFactory

#64 = Class #71 // java/lang/invoke/MethodHandles$Lookup

#65 = Utf8 Lookup

#66 = Utf8 InnerClasses

#67 = Utf8 (Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodHandle;Ljava/lang/invoke/MethodHandle;Ljava/lang/invoke/MethodType;)Ljava/lang/invoke/CallSite;

#68 = Utf8 java/lang/Runnable

#69 = Utf8 run

#70 = Class #72 // java/lang/invoke/MethodHandles

#71 = Utf8 java/lang/invoke/MethodHandles$Lookup

#72 = Utf8 java/lang/invoke/MethodHandles

{

public com.amd.aparapi.samples.RunTest();

flags: ACC\_PUBLIC

Code:

stack=1, locals=1, args\_size=1

0: aload\_0

1: invokespecial #1 // Method java/lang/Object."<init>":()V

4: return

LineNumberTable:

line 3: 0

LocalVariableTable:

Start Length Slot Name Signature

0 5 0 this Lcom/amd/aparapi/samples/RunTest;

public static void main(java.lang.String[]);

flags: ACC\_PUBLIC, ACC\_STATIC

Code:

stack=3, locals=2, args\_size=1

0: ldc #2 // String John Lennon

2: astore\_1

3: new #3 // class java/lang/Thread

6: dup

7: aload\_1

8: invokedynamic #4, 0 // InvokeDynamic #0:lambda:(Ljava/lang/String;)Ljava/lang/Runnable;

13: invokespecial #5 // Method java/lang/Thread."<init>":(Ljava/lang/Runnable;)V

16: invokevirtual #6 // Method java/lang/Thread.start:()V

19: return

LineNumberTable:

line 5: 0

line 6: 3

line 1: 8

line 6: 16

line 7: 19

LocalVariableTable:

Start Length Slot Name Signature

0 20 0 args [Ljava/lang/String;

3 17 1 name Ljava/lang/String;

private static void lambda$3(java.lang.String);

flags: ACC\_PRIVATE, ACC\_STATIC, ACC\_SYNTHETIC

Code:

stack=2, locals=1, args\_size=1

0: getstatic #7 // Field java/lang/System.out:Ljava/io/PrintStream;

3: aload\_0

4: invokevirtual #8 // Method java/io/PrintStream.println:(Ljava/lang/String;)V

7: return

LineNumberTable:

line 6: 0

}