How the lambdas might effect Aparapi.

Traditionally for Aparapi we ask the user to inherit from a base class called Kernel and override a Kernel provided public void 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 working using an anonymous innerclass 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 :

**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];**

**}**

**}**

And injects the constructor call to Main$1 into the bytecode passing each ‘captured’ reference from the call site which is used by the run method (actually any method, but we only have one here).

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 ;)

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 Runnable, I really don’t 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 just use

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

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

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

In ‘project lambda’ any interface which has a Single Abstract Method is called a SAM type. Any SAM type can be used in place of a ‘long-hand’ form of declaration.

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.

Another area where lambdas are similar to anonymous inner classes concerns call site captues. Like anonymous inner classes lambdas can also 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();**

The name variable is ‘final’ so 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();**

Note for lambas 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 is not changed.

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

**{**

**String name = “John Lennon”;**

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

**}**

So let’s look at what 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 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**

And notice that the lambda is captured in the same class as the calling method as a private synthetic method called lambda$3. Here is its bytecode .

**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**

Note that the synthetic lambda method takes a single String. This is the captured String from the call site, passed into the method. Even though we are providing an implementation for Runnable.run() this actually turns into a boring plain old method (albeit synthetic with a weird name) .

Going back to the call site we can see that the above we see that invoke dynamic is being used to ‘somehow’ hook the argument to Thread’s constructor (which is a Runnable) to the lambda.

**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 these two are actually connected.

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

So in pseudo code we have

**XX: push “John Lennon”**

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

From the invoke dynamic bytecode we have two operands #4 (reference into the constant pool) and 0. At present this second operand is always 0 for invokedynamic.

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 has two operands, the #0 refers to slot #0 in the bootstrap methods table of this class, whereas the #36 refers to a NameAndType entry in the constant pool.

Lets look at the boostrap method table

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

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

Here is how I think we might introduce Lambdas to Aparapi

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

**);**

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

**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.

Of course for Aparapi I need to grab the bytecode of the run method of the SAM type so that I can convert it to OpenCL. Then I can then do some buffer magic and execute on the GPU.

In the current implementation of project lambda the call site of the SAM type actually created a synthetic anonymous inner class. This was awesome, because it means that the work for us to modify Aparapi would be really simple. Sadly, this is not going to be the \*real\* implementation. The real implementation will use method handles and the new invoke dynamic instruction.

When I compile using the –XdlambdaToMethod (using the latest lambda tree from project lambda website) sure enough, I see an anonymous inner class created which has the bytecode for my lambdafied Aparapi.SAM.run() method.

However when I compile with the –XdlambdaToMethod, I get all sorts of goodies in my class file ;) but cannot for the life of me find the bytecode for my lambdafied Aparapi.SAM.run() method.

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

}