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 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 r = new Runnable(){

@override void run(){

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

}

};

Instead I should be able to just use

Runnable r = ()->{/\* 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 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 (Single Abstract Method!). Any SAM type can be used in place of a ‘long-hand’ form of declaration.

This is useful for allsorts 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 takes a single ActionEvent arg. So it’s lambda form has a named variable inside the parenthesis (ae). You will note that we don’t have to match the arg type. From the ActionListener declaration the compiler \*knows\* the type. We just need to give the compiler an arg name to bind the arg value type to.

One more thing that make lambda’s ‘look like’ anonymous inner classes, they too can capture state from the call site.

So if I wanted to launch a thread which printed a String I could use

final String name = “John Lennon”;

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

Note for lambas that the captured string has to be ‘effectively final’. With anonymous inner classes these had to be declared final.

Effectively final means that the compiler can prove that the variable is not changed.

So the following does not need final declaration

{

String name = “John Lennon”;

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

}

So 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>