Converting Java bytecode to OpenCL

I have been asked a number of times to explain how Aparapi converts bytecode to OpenCL.   I will try to describe the basic concept here.

First we will provide a Java file format primer, then we will show how we decoded bytecodes into instructions, then a trick to 'self-assemble' expression trees and finally how we write OpenCL

A Java file-format primer

From http://en.wikipedia.org/wiki/Java_class_file or http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html we get the following high level view of the format of a Java class file.

Where u2 refers to unsigned 16 bit values (two bytes) and u4 refers to unsigned 32 entities (four bytes).

```
ClassFile {
        u4 magic; // CAFEBABE
        u2 minor_version;
        u2 major_version;
        u2 constant_pool_count;
        cp_info constant_pool[constant_pool_count-1];
        u2 access_flags;
        u2 this_class;
        u2 super_class;
        u2 interfaces_count;
        u2 interfaces[interfaces_count];
        u2 fields_count;
        field_info fields[fields_count];
        u2 methods_count;
        method_info methods[methods_count];
        u2 attributes_count;
        attribute_info attributes[attributes_count];
}
```

To access the bytecodes of the methods of a class we need to read through the constant pool (see below) the list of interfaces, the list of fields and finally we get to the list of methods.

Although we really only care about the ConstantPool and the MethodInfo's we will also need some knowledge of how to parse attributes, so this will need some patience.

We will start with the ConstantPool

The constant pool is a list of entries of the following form

```
cp_info {
   u1 tag;
   u1 info[length_of_entry];
}
```

The first byte defines the type of the entry. Most entries are of consistent length, the one exception being a UTF8 entry which depends on the number of characters/bytes in the sequence of characters.

An example entry might be

```
CONSTANT_Integer_info {
  u1 tag;     // 3
  u4 value;  // bytes representing the constant value
}
```

For a **CONSTANT_Integer_info** entry the tag will always be 3 and the tag is followed by a u4 value containing the integer value that we are representing.

For a **CONSTANT_UTF8_info** (a unicode sequence of characters – lets not use the word String here or we will get confused) entry the tag is always 1 and this tag is followed by a u2 value (the length of the following byte array) and then the bytes that make up the UTF8 value.

```
CONSTANT_Utf8_info {
  u1 tag;    // 1
  u2 length;
  u1 bytes[length];
}
```

Some constant pool entries refer to others (I have never seen a forward reference but I don't think it is excluded by the spec).

For example a String constant is represented by

```
CONSTANT_String_info {
  u1 tag;   // 8
  u2 utf8_index;
}
```

So a String constant slot in the constant pool merely contains a reference to the constant pool slot that contains a UTF8 value that contains the length and UTF8 chars that comprise the String.

So why do we have String and UTF8 entries, if all the String does is delegate to the UTF8? The reason is that not all UTF8 entries are code artifacts. For example the name of the class itself is stored as a UTF8 entry in the constant pool, but this is not an entry that is referenced from the code.

When a bytecode instructions needs to reference a String literal/constant it must do so through a slot containing a **CONSTANT_String_info** entry. To reference directly to the underlying UTF8 would be invalid (the verifier would trip up), so by making this separation we can ensure that bytecode only references real String literal references.

One more example

Let's say some bytecode is making a method call to a method

int com.amd.javalabs.MyClass.myMethod(int[] list)

The bytecode representing the call will have an immediate index into the constant pool to indicate which method it is calling. At **constantpool[method_index]** we will have a method_ref entry.

```
CONSTANT_Methodref_info {
  u1 tag; // 10
  u2 class_index;
  u2 name_and_type_index;
}
```

This slot references two other slots, referenced by class_index and name_and_type_index. At **constantpool[class_index]** we will find a class_info entry

```
CONSTANT_Class_info {
  u1 tag;
  u2 name_index;
}
```

Which contains another slot reference (name_index), at **constantpool[name_index]** we will find a UTF8 entry

```
CONSTANT_Utf8_info {
  u1 tag;    // 1
  u2 length;
  u1 bytes[length];
}
```

Which gives us the name of the class containing the method (in our case "com/amd/javalabs/MyClass"), we now have the class name containing the declared method.

Going back to our Methodref_info we find that at **constantpool[name_and_type_index]** we reference a name and type info entry

```
CONSTANT_NameAndType_info {
  u1 tag; // 11
  u2 name_index;
  u2 descriptor_index;
}
```

Which in turn references two other slots (another name_index and a descriptor index), first at **constantpool[name_index]** we will find another UTF8 entry

```
CONSTANT_Utf8_info {
  u1 tag;     // 1
  u2 length;
  u1 bytes[length];
}
```

Which gives us the name of the method, so now we know that the class "com/amd/javalabs/MyClass" contains a method called "myMethod".

Whereas at **constantpool[descriptor_index]** we will find yet another UTF8 entry

```
CONSTANT_Utf8_info {
  u1 tag;     // 1
  u2 length;
  u1 bytes[length];
}
```

Which yields the signature of the method. In this case "([I)I", which is Java crypto speak for 'A method which takes an array of int's and returns an int'.

Although this multiple linking of slots to slots is tedious to decode and track, it does allow the constant pool to be very compact. We can reuse many slots for other purposes. For example if I added

```
        int com.amd.javalabs.MyClass.myOtherMethod(int[] list)
```

To my class this would result in one new method ref (5 bytes)

```
CONSTANT_Methodref_info {
  u1 tag; // 10
  u2 class_index;
  u2 name_and_type_index;
}
```

One new name and type info (5 bytes)

```
CONSTANT_NameAndType_info {
  u1 tag; // 11
  u2 name_index;
  u2 descriptor_index;
}
```

And one new UTF8 3 + "myOtherMethod".length = 12 = 15.

```
CONSTANT_Utf8_info {
  u1 tag;    // 1
  u2 length;
  u1 bytes[length];
}
```

So we added 25 bytes to allow this new method to be added to the constant pool.

The other two new entries can reuse existing entries, We can reuse the same UTF8 containing the descriptor (because our method signature is also (I[)I) and we can reuse the same Class_info and associated UTF8's because the method is indeed in the same class.

One weird thing. Double Constants and Long Constant's each take two slots. So if at **constantpool[4]** we had

```
CONSTANT_Long_info {
 u1 tag;
 u4 high_bytes;
 u4 low_bytes;
}
```

Referencing **constantpool[5]** would be illegal. Essentially it does not exist. I am sure there was a great reason for this at one time ;) it does make parsing the file a little weird.

Attributes

As we continue to parse through the class you will note that there is an attribute list in the class file. This contains a list of attribute records that apply to the class itself.

We will also find that attribute lists occur again later when we parse the FieldInfo and MethodInfo lists, and (just to blow our minds) some Attributes themselves contain other lists of Attributes.

For our purposes we don't really *want* to parse the FieldInfo list, unfortunately these FieldInfo's are not all constant sizes, so we need to parse them, in order to step over them on our way to the MethodInfo list.

Anyway back to Attributes.

An attribute list is a list of 0 or more **attribute_info** structures each looks similar to this.

```
attribute_info {
  u2 attribute_name_index;
  u4 attribute_length;
  u1 info[attribute_length];
}
```

The first u2 value in each attribute is an **attribute_name_index**. This is actually an index into the ConstantPool. At **constantpool[attribute_name_index]** we will find UTF8Info which names this Attribute type. We will see later that for a 'SourceFile' attribute **constantpool[attribute_name_index]** will contain the **CONSTANT_Utf8_info** entry containing the chars 'SourceFile'.

The **attribute_length** defines the number of bytes following the **attribute_length** field. This could of course be 0 if the attribute was just some kind of marker (whereby its existence indicated state) in all other cases it would be >0 and the actual content would immediately follow the **attributes_length** field.

For example, the name of the SourceFile (compilation unit) is a class level Attribute. It will be in the attribute_list held at the class file level.

In this case we will have

```
SourceFile_attribute {
   u2 attribute_name_index;
   u4 attribute_length;            // 2
   u2 sourcefile_index;
}
```

So at **constant_pool[attribute_name_index]** will be a UTF8 slot containing the string "SourceFile"

In this case **attribute_length** is always 2 because the SourceFile attribute itself contains 2 more bytes.

At **constant_pool[sourcefile_index]** will be a UTF8 slot containing the name of the actual Java sourcefile, for example "MyClass.java"

The Java Virtual Machine specification defines a set of attribute names that a virtual machine must interpret and decode at various part of a classfile. It also defines some optional ones (LocalVariableLineNumberTable for example may not exist if javac –O is used), the spec also says that if a JVM comes across an attribute (other than those that it must recognize) that it does not recognize, it can just step over it and continue.

So if you had a special compiler which added a new UTF8Info slot to the ConstantPool with "MyAttribute" you would be free to add any data that you can fit in 2^16 bytes as an attribute in any attribute list in the classfile itself that was tagged with "MyAttribute".

We have an IDF which suggests adding native code (dlls) to classfiles using this mechanism, and having a JVM hack that can load the native code at runtime rather than searching the system path at runtime.

MethodInfo

So we have parsed the constant pool and we know a little bit about how to parse attributes.

Next we need to parse the list of method_info's

```
method_info {
  u2 access_flags;
  u2 name_index;
  u2 descriptor_index;
  u2 attributes_count;
  attribute_info attributes[attributes_count];
}
```

For each method_info we have the following.

access flags contains bit masks for the method. Here specific bits will indicate whether the method is abstract, public, static, native etc.

At **constantpool[name_index]** will be a UTF8 slot defining the name of this method

At **constantpool[descriptor_index]** will be a UTF8 slot defining the signature. Again using the mildly cryptic internal form where "int xxx(int [] list)" would be "([I)I"

Then we have attribute_count which tells us how many attributes we have, followed by the attributes themselves.

One of the attributes in a non abstract non native method will be a Code attribute. That is an attribute that looks like this

```
Code_attribute {
  u2 attribute_name_index;
  u4 attribute_length;
  u2 max_stack;
  u2 max_locals;
  u4 code_length;
  u1 code[code_length];
  u2 exception_table_length;
  exception_table_entry[exception_table_length];
  u2 attributes_count;
  attribute_info attributes[attributes_count];
}
```

Again at **constantpool[attribute_name_index]** will be a UTF8 "Code" because this is a Code attribute and of course attribute_length will tell us how many bytes are in the rest of this code attribute.

max_stack and max_locals define verifiable contracts with the class verifier which limit how much space is required for local variables and the maximum stack size we need for the enclosing code (ignoring it's calls of course).

Now we get to **code_length** which tells how many bytes of code we have, and **code[]** itself which contains the bytecodes that represent this method.

We will ignore the **exception_table** stuff (except to question why the exception information was not placed in an attribute?, i.e if a method does not contain any exception handlers why do we need to waste 2 bytes on it… if it were an Attribute it could have existed only if needed)

You will note that **Code** attribute has a nested list of attributes. These guys like recursive structures don't they. In this nested list of attributes one will find LocalLineNumber tables (mapping bytecode offsets to named local variables) and a bunch of Generic related stuff. You can also see how Generics were added with minimal ClassFile modifications, this Attribute mechanism allows new attributes to be defined whilst allowing JVM's that do not have a clue about Generic's to at least correctly parse the class file.

So we have the Code array. How do we parse out the instructions.

Step 1: Converting a sequence of bytes into a list of instructions representing the code.

The Java Virtual Machine spec defines the bytecodes for the JVM.

The Instruction Set represents a virtual stack-based machine with instructions taking one or more bytes to encode. This is *not* a RISC style instruction set. In most cases the length of each instruction can be decoded based upon the first byte, but some instructions (switch specifically) requires quite a bit of work. To further complicate the decoding there is a **wide** modifier which effects the next instruction (sigh), This means that we need to defer decoding to the second byte and it's immediate values.

So essentially the first pass is to step through the bytes and determine the length and the encoding and add this encoding to a list. Then step over the immediate operands and pick up the next instruction.

We repeat this process for each byte in the bytecode until we have a list of decoded instructions.

Step 2: Extraction of higher level program structures (essentially an IR)

This proved considerably more difficult than the previous stage. We initially looked at how Jode/Mocha did this and it seemed that we had a lot of code based upon the analysis of sequences of instructions.

After a while we came up with a very fast way of doing this.

Because the JVM is a Stack based machine, we can use this fact to help us recreate an IR.

In the appendix of this doc we have the javap output from a piece of code. Avoid looking at the source code that follows it ;) we will try to decode it from the bytecode.

Here are the first 25 bytes from javap.

```
    0: aload_0
    1: iconst_0
    2: invokevirtual  #15; //Method getGlobalSize:(I)I
    5: iconst_4
    6: imul
    7: istore_1
    8: aload_0
    9: iconst_0
   10: invokevirtual  #16; //Method getGlobalId:(I)I
   13: iconst_4
   14: imul
   15: istore_2
   16: fconst_0
   17: fstore_3
   18: fconst_0
   19: fstore   4
   21: fconst_0
   22: fstore   5
   24: iconst_0
   25: istore   6
```

We are attempting to extract higher level structure from this sequence hopefully as we decode each instruction.

Let us assume that we have already decoded a list of instructions. Now we will visit them in order to determine how to fold them.

```
    0: aload_0
```

This instruction pushes the object reference in slot 0 of the local variable table onto the stack. In the case of a virtual method (which we are indeed decoding) slot 0 contains the object reference 'this'. So we push 'this' onto the stack. Remember variable '0' is the hidden 'this' passed as arg 0 of every virtual method. The args of the method will occupy slot's [1…n], then the local variables of the method. Annoyingly (but consistently ☺ if we recall the constant pool) doubles and longs take two slots…

Clearly this first instruction cannot possibly consume any stack (who is pushing it?), however we can't cheat and must determine from the bytecode specification that aload_0 does not consume any stack operands. It consumes 0 and pushes 1.

Next..

```
    1:     iconst_0
```

The instruction iconst_o pushes the integer constant '0' on the stack. It consumes 0 and pushes 1.

Next…

```
    2:     invokevirtual    #15; //Method getGlobalSize:(I)I
```

Here we have a virtual invoke (represents a virtual method call). From the signature '(I)' you can see that this consumes one stack argument. We just pushed this and 0 on the stack. Because invoke virtual is *not* used for static calls (calls to static methods) a call to this method consumes argcount + 1 operands. The 0 we just pushed is the arg and 'this' we pushed previously is the instance that contains the method we are calling. So we basically are calling a method contained in this instance and passing 0. Actually as the comment from javap tells us, we are invoking "this.getGlobalSize(0)".

So this invokevirtual consumes two operands, clearly the previous instructions 'must' have produced the operands that this instruction is consuming (we will see later that **must** is too strong an assumption, but stay with me here).

So let us use the instructions in the list we have collected so far as 'proxies' for the operands that they are expecting to produce.

So if invokevirtual takes two operands and there are two instructions before it (and they both push one operand each) then collect these instructions and indent them relative to the invokevirtual.

```
    2: invokevirtual 15; //Method getGlobalSize:(I)I
        0: aload_0
        1: iconst_0
```

We have essentially nominated the aload_0 and iconst_0 instructions as 'children' of the invokevirtual.

Next…

```
    5: iconst_4
```

Here we have another integer constant push, this time we are pushing the integer value '4'. This instruction does not consume any stack operands. So we'll just add it to the list

```
    2: invokevirtual 15; //Method getGlobalSize:(I)I
        0: aload_0
        1: iconst_0
    5: iconst_4
```

Next…

```
    6: imul
```

This is a binary operator which pops two integers and pushes the product.

If you look at our list of instructions (ignoring the intents) and we treat the last two instructions as children of our new imul we get this

```
    6: imul
        2: invokevirtual 15; //Method getGlobalSize:(I)I
            0: aload_0
            1: iconst_0
        5: iconst_4
```

As you can probably now see, we are building an expression tree.

The imul is taking the result of a call to getGlobalSize(0) and mutliuplying by 4. It is then pushing the result onto the stack.

We will carry on.

```
    7: istore_1
```

This instruction pops an integer from the operand stack and stores it in slot 1 of the local variable table.

So it consumes the operand pushed by the last instruction, and (ignoring indents) we are now left with

```
    7: istore_1
        6: imul
           2: invokevirtual 15; //Method getGlobalSize:(I)I
               0: aload_0
               1: iconst_0
```

```
            5: iconst_4
```

So this is basically saying that we are assigning getGlobalSize()/4 to variable slot #1 (remember slot 0 was ‘this’).

You can probably see that if we continue this approach we end up building a list of expression trees who’s roots are all instructions that never ‘push’ anything onto the stack. These tend to be

```
[slot 6] = 0;
```

Of course slot[0] we know is 'this' so we really have

```
[slot 1] = this.getGlobalSize(0)* 4;
[slot 2] = this.getGlobalId(0) * 4;
[slot 3] = 0f;
[slot 4] = 0f;
[slot 5] = 0f;
[slot 6] = 0;
```

From the LocalVariableTable for each method we can resolve the actual textual names for the slots at any particular time.

Javap provides a dump of the LocalVariableTable which we can use to do this manually

```
 LocalVariableTable:
  Start  Length  Slot  Name   Signature
   53     116      7    dx       F
   73      96      8    dy       F
   93      76      9    dz       F
  121      48     10    invDist  F
  141      28     11    s        F
   27     148      6    i        I
    0     361      0    this     com.amd.javalabs.opencl.auto.NaiveNBodyKernel
    8     353      1    count    I
   16     345      2    globalId I
   18     343      3    accx     F
   21     340      4    accy     F
   24     337      5    accz     F
```

Sure enough if we look up slot 0 we see that the name of the variable (between pc offset 0 and 361) is indeed 'this' and it is of type '`com.amd.javalabs.opencl.auto.NaiveNBodyKernel`'. `Similarly` slot 1is an integer (I) called count. So we can replace all uses of slot 1 with count. Furthermore, because this is the first assignment we need to declare the variable count.

```
int count = this.getGlobalSize(0)* 4;
[slot 2] = this.getGlobalId(0) * 4;
[slot 3] = 0f;
[slot 4] = 0f;
[slot 5] = 0f;
[slot 6] = 0;
```

We can do this for each of the other slots and we get

```
int count = this.getGlobalSize(0)* 4;
int globalId = this.getGlobalId(0) * 4;
float accx = 0f;
float accy = 0f;
float accz = 0f;
int i = 0;
```

If we compare this to the real code ….

```
      int count = getGlobalSize(0) * 4;
      int globalId = getGlobalId(0) * 4;

      float accx = 0.f;
      float accy = 0.f;
      float accz = 0.f;

      for (int i = 0; i < count; i += 4) {
```

You can see we are onto something…

The last assignment **i6** = **0** turns out to be the declaration of the integer variable inside the for loop. Note that we have no indication so far that we are in a for loop. This requires a little more analysis.

Lets decode some more, lets look at the next section of bytecode

```
   24:  iconst_0
   25:  istore  6
   27:  iload   6
   29:  iload_1
   30:  if_icmpge       175
   33:  aload_0
   34:  getfield        #7; //Field pos_xyzm:[F
   37:  iload   6
   39:  iconst_0
   40:  iadd
   41:  faload
   42:  aload_0
   43:  getfield        #7; //Field pos_xyzm:[F
   46:  iload_2
   47:  iconst_0
   48:  iadd
   49:  faload
   50:  fsub
   51:  fstore  7

   Yada yada yada


   159: fload   5
   161: fload   11
   163: fload   9
   165: fmul
   166: fadd
   167: fstore  5
   169: iinc    6, 4
   172: goto    27
   175: fload_3
   176: ldc     #2; //float 0.0050f
   178: fmul
   179: fstore_3
   180: fload   4
```

We will walk through the ‘instructions as operands’ transformations so we can see what this will look like when we come to analyze it.

Here is the end of the instruction list we had last time

```
     25: istore   6
       24: iconst_0
```

The next instruction is

```
   27: iload 6
```

Which consumes no operands but pushes an operand. As does

```
   29: iload_1
```

So now we have

```
    25: istore    6
      24: iconst_0

     27:  iload   6
     29:  iload_1
```

Next we have

```
   30:  if_icmpge       175
```

Which pops two integers and conditionally (compare greater than or equals >=) branches to 175

So again we treat the previous two instructions as if they were the operands for the conditional branch and we get

```
     25: istore   6
        24: iconst_0
    30: if_icmpge 175
        27:  iload   6
        29:  iload_1
```

Next we have

```
   33:  aload_0
```

We have seen this before, it pushes the object reference in slot 0 (‘this’) on the stack. It consumes nothing so add it to the list.

```
   25: istore    6
        24: iconst_0
    30: if_icmpge 175
        27:  iload   6
        29:  iload_1
   33:  aload_0
```

Next

```
  34:  getfield        #7; //Field pos_xyzm:[F
```

This instruction pushes the value of the field (or reference if it is an object/array) onto the operand stack, it consumes the stack top to determine the instance from which the field value is to be extracted. So we indent the previous instruction which basically indicates that the reference is **`this.pos_xym[]`**

```
   25: istore   6
       24: iconst_0
   30: if_icmpge 175
       27:  iload   6
       29:  iload_1
   34:  getfield        #7; //Field pos_xyzm:[F
       33:  aload_0
```

Next we have

```
  37:  iload   6
  39:  iconst_0
  40:  iadd
```

This turns into

```
    40: iadd
      37: iload   6
      39:iconst_0
```

Giving us

```
   25: istore   6
       24: iconst_0
   30: if_icmpge 175
       27:  iload   6
       29:  iload_1
   34:  getfield        #7; //Field pos_xyzm:[F
       33:  aload_0
   40: iadd
      37: iload   6
      39:iconst_0
```

Next we have

```
  41:  faload
```

Which basically is a float array access which assumes the stack contains an array field reference and an integer. It pushes the accessed value. So we indent the previous two instructions (ignoring indented instructions) under this

```
   25: istore   6
```

```
       24: iconst_0
    30: if_icmpge 175
        27:  iload   6
        29:  iload_1
    41: faload
        34: getfield        #7; //Field pos_xyzm:[F
            33:  aload_0
        40: iadd
            37: iload   6
            39:iconst_0
```

The following sequence is almost identical to the previous.

```
   42:  aload_0
   43:  getfield        #7; //Field pos_xyzm:[F
   46:  iload_2
   47:  iconst_0
   48:  iadd
   49:  faload
```

It is another array reference.

```
   49: faload
       43: getfield        #7; //Field pos_xyzm:[F
           42: aload_0
       48: iadd
           46: iload_2
           47: iconst_0
```

So now we have

```
    25: istore   6
       24: iconst_0
    30: if_icmpge 175
        27:  iload   6
        29:  iload_1
    41: faload
        34: getfield        #7; //Field pos_xyzm:[F
            33:  aload_0
        40: iadd
            37: iload   6
            39:iconst_0
    49: faload
       43: getfield        #7; //Field pos_xyzm:[F
           42: aload_0
       48: iadd
           46: iload_2
           47: iconst_0
```

And now we come to

```
   50:  fsub
```

Which is a binary operator subtracting two operands and pushing result.

```
    25: istore   6
```

```
        24: iconst_0
    30: if_icmpge 175
        27:  iload   6
        29:  iload_1
    50:  fsub
        41: faload
            34: getfield        #7; //Field pos_xyzm:[F
                33:  aload_0
            40: iadd
                37: iload   6
                39:iconst_0
        49: faload
            43: getfield        #7; //Field pos_xyzm:[F
                42: aload_0
            48: iadd
                46: iload_2
                47: iconst_0
```

And finally

```
   51:  fstore  7
```

Which is a store to the variable in slot 7 from the top of the stack. Giving us

```
    25: istore   6
        24: iconst_0
    30: if_icmpge 175
        27:  iload   6
        29:  iload_1
    51:  fstore  7
        50:  fsub
            41: faload
                34: getfield        #7; //Field pos_xyzm:[F
                    33:  aload_0
                40: iadd
                    37: iload   6
                    39: iconst_0
            49: faload
                43: getfield        #7; //Field pos_xyzm:[F
                    42: aload_0
                48: iadd
                    46: iload_2
                    47: iconst_0
```

Lets just put the yada yada yada on the list ;)

Just a few more bytes and we will take another look

The next three instructions consume nothing but push three float variable references on the operand stack so we'll just add them to the list

```
    25: istore   6
        24: iconst_0
    30: if_icmpge 175
        27:  iload   6
        29:  iload_1
    51:  fstore  7
        50:  fsub
            41: faload
                34: getfield        #7; //Field pos_xyzm:[F
                    33:  aload_0
                40: iadd
                    37: iload   6
                    39: iconst_0
            49: faload
                43: getfield        #7; //Field pos_xyzm:[F
                    42: aload_0
                48: iadd
                    46: iload_2
                    47: iconst_0
         YADA YADA YADA
    159: fload   5
    161: fload   11
    163: fload   9
```

Next we have

```
     165: fmul
```

Which pops the top two operands (and pushes the product) so we will indent the last two instructions

```
    25: istore   6
       24: iconst_0
    30: if_icmpge 175
        27:  iload   6
        29:  iload_1
    51:  fstore  7
        50:  fsub
            41: faload
                34: getfield        #7; //Field pos_xyzm:[F
                    33:  aload_0
                40: iadd
                    37: iload   6
                    39: iconst_0
            49: faload
                43: getfield        #7; //Field pos_xyzm:[F
                    42: aload_0
                48: iadd
                    46: iload_2
                    47: iconst_0
         YADA YADA YADA
    159: fload   5
    165: fmul
         161: fload   11
         163: fload   9
```

And now

```
    166: fadd
```

Which pops the top two operands (and pushes sum) so we will indent the last two instructions

```
 25: istore   6
    24: iconst_0
30: if_icmpge 175
    27:  iload   6
    29:  iload_1
```

```
         166: fadd
              159: fload   5
              165: fmul
                   161: fload   11
                   163: fload   9
```

Next

```
   169: iinc    6, 4
```

Which consumes nothing and pushes nothing. It add's 4 to the integer variable in slot 6. So it goes onto the list

Penultimiately….

```
   172: goto    27
```

Which consumes nothing and pushes nothing. It is an unconditional branch

And finally

```
   175: fload_3
```

Which we will ignore except to note that it exists at 175 (which is an earlier branch target)

Now lets look at this final sequence

```
    25: istore   6
       24: iconst_0
    30: if_icmpge 175
        27:  iload   6
        29:  iload_1
    51:  fstore  7
        50:  fsub
            41: faload
                34: getfield        #7; //Field pos_xyzm:[F
                   33:  aload_0
                40: iadd
                    37: iload   6
                    39: iconst_0
            49: faload
                43: getfield        #7; //Field pos_xyzm:[F
                    42: aload_0
                48: iadd
                    46: iload_2
                    47: iconst_0
         YADA YADA YADA
    167: fstore  5
         166: fadd
              159: fload   5
              165: fmul
                   161: fload   11
                   163: fload   9
    169: iinc    6, 4
```

```
172: goto    27

175: fload_3
```

For brevity (finally you say) lets write the root of each branch and the text equivalent of what the tree that it represents turns into

```
 25: istore 6        i=0
30:  if_icmpge 175   i < count
51:  fstore 7        dx=pos_xyzm[i+0]-pos_xyzm[globalId+0];

     YADA YADA YADA
167: fstore 5        accz = accz + s * dz;
169: iinc 6, 4       i += 4
172: goto 27
175: fload_3
```

Can you see the for loop?

```
 25: istore 6        i=0
30:  if_icmpge 175   i < count
51:  fstore 7        dx=pos_xyzm[i+0]-pos_xyzm[globalId+0];

     YADA YADA YADA
167: fstore 5        accz = accz + s * dz;
169: iinc 6, 4       i += 4
172: goto 27
175: fload_3
```

It turns out that once we collect all these expression trees, we can not only use the tree's to actually create expression statements (as text), but we can also use patterns of 'roots' to locate higher level program structures.

For example, if we find a store followed by a conditional forward branch and the instruction before the target of the conditional forward branch is a goto which branches to a target between the original store and the conditional forward branch then we have a for loop.

I know it doesn't scan well but it is a pattern that we can detect fairly simply by just traversing the roots.

Once we have detected a pattern of roots as a construct, it also turns out that all of the 'control expressions' needed by the higher level construct (in this case a for loop) are all at hand because they themselves are all roots.

The patterns for high level constructs (`if(){}, if(){}else{})` can also be generalized in the same way.

Distinguishing between a while loop and a for loop turns out to be hard (and in some cases are correctly interchangeable), and we need to use the LocalVariableTable to arbitrate.

Links/References

http://www.cs.toronto.edu/~yijun/literature/paper/beyls99europvm.pdf
http://code.google.com/p/jsr308-langtools/wiki/AnnotationsOnStatements
http://code.google.com/edu/parallel/mapreduce-tutorial.html
http://llvm.org/devmtg/2009-10/OpenCLWithLLVM.pdf

Appendix