Parallel Programming Lab, Week 9: Aparapi, briefly

This is not a full length lab practical, but should give you basic experience in using Aparapi for GPU programming - perhaps it may tempt you to use this interface for your development project.

The example given here is the matrix multiplication problem that we have used in the lectures.

First, try running this pretty naive sequential matrix multiplication code on the CPU:

```
public class Matmul {

    public static final int N = 1024 ;

    public static void main(String [] args) {

        float [] a = new float [N * N], b = new float [N * N] ;
        float [] c = new float [N * N] ;

        for (int i = 0 ; i < N ; i++) {
            for(int j = 0 ; j < N ; j++) {
                a [N * i + j] = i + j ;
                b [N * i + j] = i - j ;
            }
        }

        long startTime = System.currentTimeMillis() ;
        for(int i = 0 ; i < N; i++) {
            for(int j = 0 ; j < N ; j++) {
                float sum = 0 ;
                for(int k = 0 ; k < N ; k++) {
                    sum += a [N * i + k] * b [N * k + j] ;
                }
                c [N * i + j] = sum ;
            }
        }

        long endTime = System.currentTimeMillis() ;

        long timeMs = endTime - startTime ;

        System.out.println("Sequential matrix multiplication completed in "
                + timeMs + " milliseconds") ;
        System.out.println("Sequential performance = " +
                ((2L * N * N * N) / (1E6 * timeMs)) + " GFLOPS") ;
    }
}
```

This will probably report a fairly respectable performance between about 1 and 2 GFLOPS on the lab computers.

When we come to run this problem on the GPU, we will push to larger N values. In the above code, try increasing the value of this variable to 2048. You may, like me, observe a precipate drop in performance.

This performance cliff is probably associated with the data structures getting too large to comfortably fit in cache memory. Replace the the central loop nest that calculates c with this code:

```
float [] bt = new float [N * N] ;


for(int i = 0 ; i < N; i++)
    for(int j = 0 ; j < N ; j++)
        bt [N * i + j] = b [N * j + i] ;


for(int i = 0 ; i < N; i++) {
    for(int j = 0 ; j < N ; j++) {
        float sum = 0 ;
        for(int k = 0 ; k < N ; k++) {
            sum += a [N * i + k] * bt [N * j + k] ;
        }
        c [N * i + j] = sum ;
    }
}
```

Here we have simply introduced a temporary array bt that is initialized with the *transpose* of matrix b, and used this in the main loop (with bt indices swapped round in the inner loop so we get exactly the same result for c). Run this again with N = 2048. You may see a startling improvement in performance - a salutary lesson in the essential role of cache in performance of modern CPUs.

Moving on to Aparapi, create a new project in Netbeans, but this time select the *Maven* top level category, rather than just *Java*. The project type will still be *Java Application*.

Give the project a name like aparapiMatmul. You can change the Group Id from its default if you wish (if, for example, you have a favourite domain of your own).

Right click on the project folder called just Dependencies, select "Add Dependency...". The dependency we are going to add is Aparapi.

Set "Group ID" to com.aparapi, "Artifact ID" to aparapi and "Version" to 1.8.0, or whatever is currently documented at:

http://aparapi.com/introduction/getting-started.html

and leave other fields unchanged. Click "Add".

Right click on the project directory under "Source Packages", and add a new Java Class, which I will call "AparapiMatmul". The code under the package declaration can look like this:

```
import com.aparapi.Kernel;

import com.aparapi.ProfileInfo;

import com.aparapi.Range;

import com.aparapi.device.Device ;


public class AparapiMatmul {


    public static final int N = 2048 ;


    public static void main(String [] args) {


        float [] a = new float [N * N], b = new float [N * N] ;

        float [] c = new float [N * N] ;


        for (int i = 0 ; i < N ; i++) {

            for(int j = 0 ; j < N ; j++) {

                a [N * i + j] = i + j ;

                b [N * i + j] = i - j ;

            }

        }
```

```
Kernel kernel = new Kernel() {
    public void run() {
        int tx = getGlobalId(0) ;
        int ty = getGlobalId(1) ;

        float sum = 0 ;
        for(int k = 0 ; k < N ; k++) {
            sum += a [N * ty + k] * b [N * k + tx] ;
        }
        c [N * ty + tx] = sum ;
    }
} ;

long startTime = System.currentTimeMillis() ;

Device device = Device.best() ;

Range range = device.createRange2D(N, N) ;
kernel.execute(range) ;

long endTime = System.currentTimeMillis() ;

System.out.println("Device type = " +
                   device.getType());

long timeMs = endTime - startTime ;
System.out.println("Matrix multiplication completed in "
                    + timeMs + " milliseconds") ;
          System.out.println("Performance = " +
                    ((2L * N * N * N) / (1E6 * timeMs)) + " GFLOPS") ;
     }
}
```

Run this project as usual. Increase N, e.g. to 4096 and run again. Performance should increase with increasing problem size, and eventually eclipse CPU performance by one to two orders of magnitude.

For more documentation on using Aparapi, see:

http://aparapi.com/