Leveraging Aparapi to Help Improve Financial Java Application Performance

Shrinivas Joshi, Software Performance Engineer

Abstract

Graphics Processing Unit (GPU) and Accelerated Processing Unit (APU) offload of generic Java applications has been made possible in the past by mechanisms such as Java bindings for OpenCL™/CUDA. Recently, frameworks such as Aparapi have made it extremely easy for Java applications to leverage hardware resources of OpenCL™ capable devices. By refactoring JQuantlib API to use Aparapi kernels, a process that took no more than two man weeks, for sufficiently large problem sizes we were able to achieve up to 20x improvements in performance of a sample Java based quantitative finance application. This article discusses how to implement and tune Aparapi kernels for data-parallel operations within Java applications and the performance gains that can be achieved using Aparapi framework.

Introduction

AMD recently announced the open source release of the Aparapi project. Aparapi aims at improving Java application performance by offloading data parallel operations to an OpenCL™ capable device such as a GPU. Aparapi supports GPUs from most major vendors. If you are not familiar with the Aparapi project we highly recommend you to read the following blog post by Gary Frost from AMD.


While we were working on Aparapi it was evident that Aparapi would help us gain orders-of-magnitude performance improvements on embarrassingly data-parallel operations with relatively smaller data footprint. This applied to “classic” GPU-friendly workloads such as Nbody and Mandelbrot amongst others. We were interested in exploring how Aparapi could help a generic data parallel application which may not necessarily be scientific in nature.

In this article we will talk about how we used the Aparapi project to achieve substantial performance gains on a sample financial Java application which uses a quantitative finance library called JQuantlib. We will discuss details of the approach we took to - identify data parallel opportunities, implement Aparapi kernels to offload those data parallel operations to an OpenCL™ capable device, and optimize the kernels. We will also talk about the performance improvements that were achieved using this approach.
**Workload**

This section contains details of the workload that we used for the purpose of this study. Many of you, especially those with experience in quantitative finance domain, may know that JQuantlib is a Java based API that is used to derive valuation of financial instruments such as options, shares, bonds, futures etc. It is derived from C++ based Quantlib API. For more details on JQuantlib API please refer to the documentation at [http://www.jquantlib.org/index.php/Main_Page](http://www.jquantlib.org/index.php/Main_Page)

We knew that some of the algorithms used for option pricing were data parallel in nature. So our first choice was to start with examples leveraging option pricing related API classes of JQuantlib. Fortunately, we found a sample Java program on the JQuantlib website which exercised different option pricing methods.

See [http://www.jquantlib.org/maven2/sites/jquantlib/xref/org/jquantlib/examples/EquityOptions.html](http://www.jquantlib.org/maven2/sites/jquantlib/xref/org/jquantlib/examples/EquityOptions.html) for the source code of this sample program. We modified this program so that it exercised only Binomial method based option pricing algorithms, also only for American options type. We also added code to capture execution time of these different option pricing algorithms. Appendix A has complete source code of our modified sample program.

**Methodology for Implementing Aparapi Kernels**

In this section we will go through the steps one could take to - identify opportunities for using Aparapi kernels in their Java workloads, implement Aparapi kernels and finally tune their implementation for better performance.

**Characteristics of code that can benefit from GPU offload**

Before we go in to the details of how to identify opportunities for using Aparapi kernels, let us understand the characteristics of operations that can benefit from GPU offload. First of all, the operations that you are trying to speed up need to be data parallel so that they can leverage large number of processing cores available on a GPU. Secondly, there is a trade-off between the cost incurred by data transfer between the CPU and GPU and the gains achieved by parallelizing operations on the GPU. Thus your application needs to spend enough time in computational operations to mitigate the cost of data transfer. These data parallel operations also need to be execution hot spots of your application for it to see sizeable gains from GPU offload. This was indeed the case with our JQuantlib based application.

**Identifying GPU friendly operations within your workload**

If you already know that there are parallelizable operations within your workload then that is a good place to start. You can also use profiling tools to identify execution hot spots and then look through the corresponding source code to see if they are or could be parallelized. In our case, we used AMD CodeAnalyst profiler and noticed that our sample application was stressing JQuantlib API methods which were indeed parallelizable. Generally, you would notice that tight loops performing some operations over primitive data types or primitive arrays could be parallelized. Certain seemingly sequential
operations could be parallelized with appropriate changes in the implementation logic; certain operations seem sequential in nature but their associative nature allows for parallelization.

**Implementing Aparapi kernels for data parallel operations**

Once you identify code sites that are better suited to be executed on GPU, you can start looking at ways to replace them with Aparapi kernels. There are certain restrictions on the kind of code that can be used in Aparapi kernels. For example, you can only use primitive data types, you can only use single dimensional arrays and you can generally only make calls to other methods declared in the same class as the initial `run()` method, amongst others.


**JQuantlib Aparapi Kernels**

In case of JQuantlib, we started with looking at one data-parallel code site at a time. Typically these code sites were 1 or 2 lines of code within a method of a particular class. We started defining corresponding Aparapi kernel class within the classes containing the parallelizable code. We defined setter methods within these kernel classes for passing in input data to the kernel. We instantiated an object of this kernel class within the static initializer of the containing class so that we could just make a call to kernel `execute()` method when control reached the original data parallel code site. Now we could replace the original code with code that generally would do 3 operations in the following sequence:

- Extract primitive input data from Java objects and call kernel setter methods to pass this data to the kernels.
- Execute the kernel.
- Extract output generated by kernels and assign it to appropriate variables in the original code.

Following is a general structure of some of the JQuantlib classes before and after we introduced Aparapi kernels:

**Original code**

```java
public class JQClass{
  private final double instanceVar1;
  private final double instanceVar2;
  public JQClass () {
    // initialize instanceVar1 and instanceVar2
  }

  public void jqmethod(final int i, final JQObject inputBuffer, final JQObject outputBuffer) {
    // data parallel loop
    // here JQObject is a container type of class which holds a double array amongst other variables
    for (int j = 0; j < i; j++) {
      // write to appropriate elements of double array in outputBuffer
      // using results derived from appropriate elements of double array
    }
  }
}
```
Code after integrating Aparapi kernel (see bold-faced text)

```java
import com.amd.aparapi.Kernel;

public class JQClass{

    private final double instanceVar1;
    private final double instanceVar2;
    public JQClass () {
        // initialize instanceVar1 and instanceVar2
    }

    private static class JQClassKernel extends Kernel{
        private double[] kernelInputBuffer;
        private double[] kernelOutputBuffer;
        private double kernelInstanceVar1;
        private double kernelInstanceVar2;

        /** setter for kernelInputBuffer */
        /** setter for kernelOutputBuffer */
        /** setter for kernelInstanceVar1 */
        /** setter for kernelInstanceVar2 */

        @Override public void run() {
            int gid = getGlobalId();
            // set kernelOutputBuffer[gid] to appropriate value derived
            // using kernelInputBuffer, kernelInstanceVar1 and
            // kernelInstanceVar2
        }
    }

    static JQClassKernel jqckernel = new JQClassKernel();

    public void jqmethod(final int i,
        final JObject inputBuffer,
        final JObject outputBuffer) {
        double[] _inputbuffer = //extract double array from inputBuffer object;
        double[] _outputbuffer = new double[size_of_outputBuffer_array];
        jqckernel.setInputBuffer(_inputbuffer);
        jqckernel.setOutputBuffer(_outputbuffer);
        jqckernel.setInstanceVar1(instanceVar1);
        jqckernel.setInstanceVar2(instanceVar2);
        jqckernel.execute(i);
        // assign kernel modified values of _outputbuffer to the primitive
        // double array of outputBuffer object
    }
}
```
**Tuning Aparapi kernels**

A problem encountered with the above mentioned approach was that we did not see substantial gains from GPU offload. This section talks about some of the ways to tune Aparapi kernel implementation help achieve substantial gains from GPU offload.

**Increase compute and reduce data transfer**

The reason we didn't see gains from GPU offload was that we were incurring a lot of data transfer cost and were doing far too little compute operations on the GPU. We were also incurring this cost at multiple code sites. It became clear that the way to address this issue was to find a mechanism to increase compute operations per kernel and to also reduce the associated data transfer cost. Merging logic from multiple kernels in to fewer numbers of kernels was the way to achieve this effect. To get the implementation of these merged kernels right we had to define new *getter* methods in some of the JQuantlib API classes. These *getter* methods allowed us to query certain data from the classes containing the merged kernel, which was not possible before. In our case it was fairly easy to merge kernels because the control flow was such that one data parallel code site was followed by call to another data parallel code site with little or no side effect (between the calls) on the data variables involved. Merged kernels allowed us to increase compute operations within the kernel and to also reduce the data transfer cost, both of which were bottlenecks previously. However, note that in this process of merging multiple kernels we had to lose some object oriented qualities of the JQuantlib API. In that, we defined *getter* methods where they were not required previously and we consolidated certain operations performed by multiple Java classes in fewer number of classes.

**Explicit buffer management**

We also came across scenarios where Aparapi's explicit buffer management feature came in handy. This feature allows more fine-grained user control of *which* data gets transferred between host and OpenCL device (and vice versa) and *when* the data transfer happens.

At certain code sites we were not reading output buffers on the host side between successive kernel executions. Also, we were only reading either of the output buffers depending on some runtime condition. Using explicit buffer management during these situations led to substantial performance gains. In our case the code pattern that allowed us to leverage explicit buffer management looked like this:

```java
final double[] bufferA = new double [SIZE];
final double[] bufferB = new double [SIZE];
Kernel kernel= new Kernel(){
    // read/write bufferA OR bufferB depending on some condition
};
boolean someCondition = // set the value at run time
for (int i=start; i < end; i++){
    kernel.execute(i);
}
if(someCondition){
    // read bufferA
} else {
    // read bufferB
}
```
Without explicit buffer management, bufferA and bufferB got transferred from GPU to the host/CPU between every execution of the kernel in the for loop. Also both bufferA and bufferB got transferred after the for loop irrespective of run-time value of someCondition variable. We could easily modify this to use explicit buffer management as (see bold faced text):

```java
final double[] bufferA = new double [SIZE];
final double[] bufferB = new double [SIZE];
Kernel kernel= new Kernel(){
    // read/write bufferA OR bufferB depending on some condition
};
kernel.setExplicit(true); // tells Aparapi that the user will
    // control buffer management for this kernel
kernel.put(bufferA); // transfer bufferA to the OpenCL device
kernel.put(bufferB); // transfer bufferB to the OpenCL device

boolean someCondition = // set the value at run time
    // neither bufferA nor bufferB will be transferred back to the host
for (int i=start; i < end; i++){
    kernel.execute(i);
}
if(someCondition) {
    kernel.get(bufferA); // get only bufferA transferred from OpenCL device
    //read bufferA
} else {
    kernel.get(bufferB); // get only bufferB transferred from OpenCL device
    //read bufferB
}
```

This approach avoided unnecessary buffer transfers between successive executions of the kernel. It also allowed selective transfer of either of the buffers based on the runtime condition.

For additional code patterns that could benefit from explicit buffer management and more details on this feature please see [http://code.google.com/p/aparapi/wiki/NewFeatures](http://code.google.com/p/aparapi/wiki/NewFeatures)

Another general suggestion on kernel tuning is to try to move host side computation logic between successive kernel executions into the kernels itself. That way you make your kernels more compute intensive and you also don’t have to cross GPU-CPU boundary as much as you would need to otherwise.

**Empirical Data**

We ran our sample application using the stock JQuantlib 0.2.3 release and compared it with our JQuantlib version that included changes for using Aparapi kernels. We call performance numbers collected using the stock JQuantlib version as Reference numbers (REF for short). We call performance numbers collected using modified JQuantlib version as GPU numbers.
Experimental Setup
Our experiments were performed on a system with following hardware and software configuration:

- AMD Phenom™ II X6 1090T @ 3.2GHz processor
- 8 GB DDR3 1600MHz RAM
- MSI 890FXA-GD70 motherboard
- ATI Radeon™ HD 5870 graphics card
- Windows® 7 Ultimate 32-bit OS
- JDK version 1.7.0 (b147 with HotSpot Server VM build 21.0-b17)
- Aparapi 2011_09_13 release
- AMD Catalyst version 11.9
- AMD APP SDK v2.5
- JQuantlib version 0.2.3

We looked at performance of all the different binomial option pricing methods namely Tian, Jarrow-Rudd, Trigeorgis, CoxRossRubinstein, AdditiveEQPBinomialTree, Joshi4 and LeisenReimer exercised by the original sample application. We exercised each of these option pricing methods for 5 iterations and took the average of the last 4 iterations for deriving their execution time. We exclude numbers from the first iteration to filter out class loading and JIT compilations noise on the Java side and Java bytecode to OpenCL™ conversion cost on the Aparapi side which however happens only prior to the first execution of the kernel.

Table 1 and Table 2 contain execution time in milliseconds for different Binomial option pricing methods at different global size values. Table 1 contains data for global sizes 64 through 1024. Table 2 contains data for global sizes 2048 through 32768. The “Performance Gains” section of these tables show the speed-up achieved by GPU offload as compared to the REF performance. A ratio of less than 1 indicates that the REF performance was better than GPU performance. A ratio of 1 indicates that they were at par and a ratio of greater than 1 indicates the speed-up achieved by GPU offload. The “Global Size” here indicates the number of steps used by the option pricing algorithms. The number of steps maps to the granularity at which the upward and downward movement of options are computed.

As you can see in Table 1 and Table 2, as the global size increases GPU offload starts to outperform reference implementation. This is because the gains achieved by GPU offload starts to overcome the cost of data transfer between the CPU and the GPU. On some of the option pricing algorithms a speed-up of as much as 20x was achieved using Aparapi. Also, GPU starts to outperform REF implementation at global sizes as small as 512. Note that for smaller problem sizes one could continue using reference implementation since it is possible to toggle Aparapi kernel usage by means of user defined run-time flags of the JVM process.
Note: In some cases we saw REF implementation taking 0 milliseconds. We have replaced those data points with 0.1 millisecond instead to get meaningful performance difference numbers. There are 5 such instances and all of them are in Table 1.

<table>
<thead>
<tr>
<th>Global Size</th>
<th>REF 64</th>
<th>GPU 64</th>
<th>REF 128</th>
<th>GPU 128</th>
<th>REF 256</th>
<th>GPU 256</th>
<th>REF 512</th>
<th>GPU 512</th>
<th>REF 1K</th>
<th>GPU 1K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tian</td>
<td>3.75</td>
<td>7.75</td>
<td>7.75</td>
<td>11.75</td>
<td>19.5</td>
<td>23.25</td>
<td>66.25</td>
<td>50.75</td>
<td>249.75</td>
<td>101.3</td>
</tr>
<tr>
<td>Jarrow-Rudd</td>
<td>0.1</td>
<td>8</td>
<td>4</td>
<td>11.75</td>
<td>3.75</td>
<td>19.5</td>
<td>15.5</td>
<td>46.75</td>
<td>50.5</td>
<td>93.5</td>
</tr>
<tr>
<td>Trigeorgis</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>11.75</td>
<td>7.75</td>
<td>19.5</td>
<td>15.5</td>
<td>47</td>
<td>50.75</td>
<td>89.75</td>
</tr>
<tr>
<td>CoxRossRubinstein</td>
<td>0.1</td>
<td>7.75</td>
<td>3.75</td>
<td>11.5</td>
<td>0.1</td>
<td>23.5</td>
<td>15.75</td>
<td>43</td>
<td>50.75</td>
<td>89.75</td>
</tr>
<tr>
<td>AdditiveEQPBinomialTree</td>
<td>0.1</td>
<td>3.75</td>
<td>0.1</td>
<td>11.75</td>
<td>3.75</td>
<td>23.25</td>
<td>15.5</td>
<td>47</td>
<td>89.75</td>
<td></td>
</tr>
<tr>
<td>Joshi4</td>
<td>3.75</td>
<td>7.75</td>
<td>7.75</td>
<td>11.75</td>
<td>15.5</td>
<td>23.5</td>
<td>66.25</td>
<td>46.75</td>
<td>245.75</td>
<td>97.5</td>
</tr>
<tr>
<td>LeisenReimer</td>
<td>4</td>
<td>7.75</td>
<td>7.75</td>
<td>15.5</td>
<td>27.5</td>
<td>66.25</td>
<td>46.75</td>
<td>249.75</td>
<td>101.3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Performance Gains</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Size</td>
</tr>
<tr>
<td>Tian</td>
</tr>
<tr>
<td>Jarrow-Rudd</td>
</tr>
<tr>
<td>Trigeorgis</td>
</tr>
<tr>
<td>CoxRossRubinstein</td>
</tr>
<tr>
<td>AdditiveEQPBinomialTree</td>
</tr>
<tr>
<td>Joshi4</td>
</tr>
<tr>
<td>LeisenReimer</td>
</tr>
</tbody>
</table>

Table 1: Comparison of REF and GPU execution time for global sizes 64 through 1024

<table>
<thead>
<tr>
<th>Global Size</th>
<th>REF 2K</th>
<th>GPU 2K</th>
<th>REF 4K</th>
<th>GPU 4K</th>
<th>REF 8K</th>
<th>GPU 8K</th>
<th>REF 16K</th>
<th>GPU 16K</th>
<th>REF 32K</th>
<th>GPU 32K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tian</td>
<td>967.25</td>
<td>217.00</td>
<td>3872.75</td>
<td>511.00</td>
<td>15713.00</td>
<td>1299.00</td>
<td>64266.30</td>
<td>4006.00</td>
<td>262902.00</td>
<td>13357.75</td>
</tr>
<tr>
<td>Jarrow-Rudd</td>
<td>171.75</td>
<td>179.50</td>
<td>670.75</td>
<td>538.75</td>
<td>2718.50</td>
<td>768.25</td>
<td>10970.80</td>
<td>1954.00</td>
<td>43785.50</td>
<td>5179.00</td>
</tr>
<tr>
<td>Trigeorgis</td>
<td>171.75</td>
<td>179.30</td>
<td>674.75</td>
<td>355.00</td>
<td>2707.75</td>
<td>768.50</td>
<td>10970.80</td>
<td>1942.00</td>
<td>43876.00</td>
<td>5109.00</td>
</tr>
<tr>
<td>CoxRossRubinstein</td>
<td>175.50</td>
<td>179.50</td>
<td>667.00</td>
<td>358.80</td>
<td>2749.75</td>
<td>768.30</td>
<td>11023.00</td>
<td>1954.00</td>
<td>44759.50</td>
<td>5109.00</td>
</tr>
<tr>
<td>AdditiveEQPBinomialTree</td>
<td>175.50</td>
<td>179.50</td>
<td>670.75</td>
<td>359.00</td>
<td>2710.75</td>
<td>768.30</td>
<td>11177.80</td>
<td>1958.00</td>
<td>44664.00</td>
<td>5172.00</td>
</tr>
<tr>
<td>Joshi4</td>
<td>971.25</td>
<td>210.50</td>
<td>3868.75</td>
<td>511.00</td>
<td>15725.00</td>
<td>1299.00</td>
<td>64245.50</td>
<td>3787.00</td>
<td>263657.00</td>
<td>13685.00</td>
</tr>
<tr>
<td>LeisenReimer</td>
<td>963.25</td>
<td>210.50</td>
<td>3868.75</td>
<td>511.00</td>
<td>15725.00</td>
<td>1299.00</td>
<td>64245.50</td>
<td>3787.00</td>
<td>263657.00</td>
<td>15355.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Performance Gains</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Size --&gt;</td>
</tr>
<tr>
<td>Jarrow-Rudd</td>
</tr>
<tr>
<td>Trigeorgis</td>
</tr>
<tr>
<td>CoxRossRubinstein</td>
</tr>
<tr>
<td>AdditiveEQPBinomialTree</td>
</tr>
<tr>
<td>LeisenReimer</td>
</tr>
</tbody>
</table>

Table 2: Comparison of REF and GPU execution time for global sizes 2048 through 32768
Figure 1 and Figure 2 show performance difference between REF and GPU implementation for Tian option pricing type.

Figure 1: Performance comparison between REF and GPU mode of Tian method for global sizes 64 through 1024.
Conclusion

Aparapi is a simple and a very effective mechanism of leveraging GPU compute resources for general purpose Java computations. If you have an application that is data parallel in nature then it is worthwhile to investigate how you could integrate it with Aparapi framework. Using Aparapi we were able to speed up a sample Java based financial application by as much as 20x. We hope that this article has provided you with sufficient directions that you could use to get started with integrating Aparapi into your applications.

Acknowledgements

We would like to thank Vincent Hindriksen, CEO of StreamComputing for providing us insightful comments and suggestions on this article. We would also like to thank the whole Runtimes group at AMD, Austin for their comments and suggestions.

Appendix A

Appendix A contains source code for the sample application that was used for performance measurements noted in Table 1 and Table 2.

Figure 2: Performance comparison between REF and GPU mode of Tian method for global sizes 2048 through 32768
package org.jquantlib.samples;

import org.jquantlib.Settings;
import org.jquantlib.daycounters.Actual365Fixed;
import org.jquantlib.daycounters.DayCounter;
import org.jquantlib.exercise.AmericanExercise;
import org.jquantlib.exercise.Exercise;
import org.jquantlib.instruments.Option;
import org.jquantlib.instruments.Payoff;
import org.jquantlib.instruments.PlainVanillaPayoff;
import org.jquantlib.instruments.VanillaOption;
import org.jquantlib.methods.lattices.AdditiveEQPBinomialTree;
import org.jquantlib.methods.lattices.CoxRossRubinstein;
import org.jquantlib.methods.lattices.JarrowRudd;
import org.jquantlib.methods.lattices.Joshi4;
import org.jquantlib.methods.lattices.LeisenReimer;
import org.jquantlib.methods.lattices.Tian;
import org.jquantlib.methods.lattices.Trigeorgis;
import org.jquantlib.pricingengines.vanilla.BinomialVanillaEngine;
import org.jquantlib.processes.BlackScholesMertonProcess;
import org.jquantlib.quotes.Handle;
import org.jquantlib.quotes.Quote;
import org.jquantlib.quotes.SimpleQuote;
import org.jquantlib.termstructures.BlackVolTermStructure;
import org.jquantlib.termstructures.YieldTermStructure;
import org.jquantlib.termstructures.volatilities.BlackConstantVol;
import org.jquantlib.termstructures.yieldcurves.FlatForward;
import org.jquantlib.time.Calendar;
import org.jquantlib.time.Date;
import org.jquantlib.time.Month;
import org.jquantlib.time.calendars.Target;

/**
 * Calculates equity option values with a number of methods
 *
 * @see http://quantlib.org/reference/_equity_option_8cpp-example.html
 * @author Richard Gomes
 * @author Shrinivas Joshi
 */
public class AparapiEquityOptions implements Runnable{

    public static void main(final String[] args) {
        new AparapiEquityOptions().run();
    }

    public void run() {
        double aNPV = Double.NaN;
        // set up dates
        final Calendar calendar = new Target();
        final Date todaysDate = new Date(15, Month.May, 1998);
        final Date settlementDate = new Date(17, Month.May, 1998);
        new Settings().setEvaluationDate(todaysDate);

        // our options
        final Option.Type type = Option.Type.Put;
        final double strike = 40.0;
        final double underlying = 36.0;
        /* @Rate */ final double riskFreeRate = 0.06;
        final double volatility = 0.2;
        final double dividendYield = 0.00;

        final Date maturity = new Date(17, Month.May, 1999);
        final DayCounter dayCounter = new Actual365Fixed();

        // Define exercise for American Options
        final Exercise americanExercise = new AmericanExercise(settlementDate, maturity);

        // bootstrap the yield/dividend/volatility curves
```java
final Handle<Quote> underlyingH = new Handle<Quote>(new SimpleQuote(underlying));
final Handle<YieldTermStructure> flatDividendTS = new Handle<YieldTermStructure>(new FlatForward(settlementDate, dividendYield, dayCounter));
final Handle<YieldTermStructure> flatTermStructure = new Handle<YieldTermStructure>(new FlatForward(settlementDate, riskFreeRate, dayCounter));
final Handle<BlackVolTermStructure> flatVolTS = new Handle<BlackVolTermStructure>(new BlackConstantVol(settlementDate, calendar, volatility, dayCounter));

final Payoff payoff = new PlainVanillaPayoff(type, strike);

final BlackScholesMertonProcess bsmProcess = new BlackScholesMertonProcess(underlyingH, flatDividendTS, flatTermStructure, flatVolTS);

// American Options
final VanillaOption americanOption = new VanillaOption(payoff, americanExercise);

// Analytic formulas:

int timeSteps = Integer.getInteger("numSteps", 256);
int iterations = Integer.getInteger("iterations", 10);

System.out.println("Number of iterations " + iterations);

long iterTime;
if (Boolean.getBoolean("JarrowRudd")) {
    System.out.println("Binomial Jarrow-Rudd");
    for (int i = 0; i < iterations; i++) {
        // Binomial JarrowRudd
        americanOption.setPricingEngine(new BinomialVanillaEngine<JarrowRudd>(bsmProcess, timeSteps){
            iterTime = System.currentTimeMillis();
            aNPV = americanOption.NPV();
            System.out.println("Binomial Jarrow-Rudd Iteration " + (i + 1) + " took " + (System.currentTimeMillis() - iterTime));
        });
    }
    System.out.println("American option " + aNPV);
}

if (Boolean.getBoolean("Tian")) {
    System.out.println("Binomial Tian");
    for (int i = 0; i < iterations; i++) {
        // Binomial Tian
        americanOption.setPricingEngine(new BinomialVanillaEngine<Tian>(bsmProcess, timeSteps){
            iterTime = System.currentTimeMillis();
            aNPV = americanOption.NPV();
            System.out.println("Binomial Tian Iteration " + (i + 1) + " took " + (System.currentTimeMillis() - iterTime));
        });
    }
    System.out.println("American option " + aNPV);
}
```
System.out.println("American option " + aNPV);

if (Boolean.getBoolean("Joshi4")) {
    System.out.println("Binomial Joshi4");
    for (int i = 0; i < iterations; i++) {
        BinomialVanillaEngine<Joshi4>(bsmProcess, timeSteps);
        iterTime = System.currentTimeMillis();
        aNPV = americanOption.NPV();
        System.out.println("Binomial Joshi4 Iteration " + (i + 1) + " took " + (System.currentTimeMillis() - iterTime));
    }
    System.out.println("American option " + aNPV);
}

if (Boolean.getBoolean("LeisenReimer")) {
    System.out.println("Binomial LeisenReimer");
    for (int i = 0; i < iterations; i++) {
        BinomialVanillaEngine<LeisenReimer>(bsmProcess, timeSteps);
        iterTime = System.currentTimeMillis();
        aNPV = americanOption.NPV();
        System.out.println("Binomial LeisenReimer Iteration " + (i + 1) + " took " + (System.currentTimeMillis() - iterTime));
    }
    System.out.println("American option " + aNPV);
}

if (Boolean.getBoolean("Trigeorgis")) {
    System.out.println("Binomial Trigeorgis");
    for (int i = 0; i < iterations; i++) {
        BinomialVanillaEngine<Trigeorgis>(bsmProcess, timeSteps);
        iterTime = System.currentTimeMillis();
        aNPV = americanOption.NPV();
        System.out.println("Binomial Trigeorgis Iteration " + (i + 1) + " took " + (System.currentTimeMillis() - iterTime));
    }
    System.out.println("American option " + aNPV);
}

if (Boolean.getBoolean("CoxRossRubinstein")) {
    System.out.println("Binomial CoxRossRubinstein");
    for (int i = 0; i < iterations; i++) {
        BinomialVanillaEngine<CoxRossRubinstein>(bsmProcess, timeSteps);
        iterTime = System.currentTimeMillis();
        aNPV = americanOption.NPV();
        System.out.println("Binomial CoxRossRubinstein Iteration " + (i + 1) + " took ")
+ (System.currentTimeMillis() - iterTime));
} System.out.println("American option " + aNPV);
}

if (Boolean.getBoolean("AdditiveEQPBinomialTree")) {
    System.out.println("Binomial AdditiveEQPBinomialTree");
    for (int i = 0; i < iterations; i++) {
        BinomialAdditiveEQPBinomialTree
            americanOption.setPricingEngine(new
                BinomialVanillaEngine<AdditiveEQPBinomialTree>{bsmProcess, timeSteps}{
                
                    iterTime = System.currentTimeMillis();
                    aNPV = americanOption.NPV();
                    System.out.println("Binomial AdditiveEQPBinomialTree
                        Iteration " + (i + 1) + " took "
                        + (System.currentTimeMillis() - iterTime));
                
            } System.out.println("American option " + aNPV);
    }
}