An Embedded DSL for Stochastic Processes *

Research Article †

Michael Flænø Werk Joakim Ahnfelt–Rønne Ken Friis Larsen

Department of Computer Science (DIKU), University of Copenhagen, Denmark
{werk,ahnfelt,kflarsen}@diku.dk

Abstract

We present a domain specific language embedded in Haskell for specifying stochastic processes, called *SPL*. It is designed with the goal of matching the notation used in mathematical finance, where the price of a financial contract is specified using stochastic processes and distributions.

SPL is declarative in the sense that it is agnostic of the choice of discretization and of the computational model. We provide an implementation of *SPL* that performs Monte Carlo simulation using GPGPU, and we present data indicating that this gives a 100x speedup compared to hand-written sequential C, and that the speedup scales linearly with the number of available cores.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.3.4 [*Programming Languages*]: Code generation; D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; J.1 [*Computer Application*]: Financial

Keywords Embedded Domain Specific Language, Haskell, OpenCL, Monte-Carlo Simulation, Code Generation, Stochastic Processes, GPGPU.

1. Introduction

A financial contract is a set of conditions for the exchange of tradable assets between two parties - the *holder* and the *counter-party*. Commonly used tradable assets are cash and stocks but financial contracts themselves may also be traded. The condition that financial contracts are defined in terms of other tradable assets or measurable numbers has led to the names *derivative* and *underlying* where the financial contract is called *derivative* and the tradable assets or numbers it depends on are called *underlyings*.

As an example consider the so called *Asian option with floating strike*. The contract holder is here given the option to buy one specific stock in, say, one year's time from now, having to pay the average stock price between now and then, if choosing to exercise the option. The actual profit gained from holding this contract depends on how high the stock price will be in one year, compared to its average price. The actual profit is therefore unknown as the future stock price is unknown. But this should not keep anybody from making a qualified guess, and this is exactly what happens on the financial markets, where contracts like this are traded on a very large scale[1]. So the question is therefore, what is this contract *expected* to be worth to the holder?

In the remainder of this section we will look at how the domain experts model such prices stochastically and how we can do the same using our domain specific language *SPL*.

1.1 The way of the quants

In mathematical finance the uncertainties in the contract prices are often modelled using stochastic processes and distributions. A distribution can be seen as a value that is uncertain. It takes on different values with various probabilities. An example of this is the normal distribution, which may be any real number but is most likely to be close to zero. A stochastic process may be viewed as a function from time to a distribution. Like distributions can be used to model a uncertain prices, stochastic processes can be used to model the future prices of stocks or financial contracts.

The Brownian motion is a particular stochastic process that is often used to capture the uncertainties of reality, when modeling a stock or standard underlying. It can be defined iteratively as follows, in terms of the standard distribution $\mathcal{N}$:

$$\begin{aligned} \mathcal{W}(0) &= 0 \\ \mathcal{W}(t+\Delta t) &= \mathcal{W}(t) + \mathcal{N}\sqrt{\Delta t} \end{aligned} \tag{1}$$

The notion above make uses of some overloading that allows standard algebra to be performed directly on distribution. First the zero on the right hand side of the base case is not a real number but rather the distribution that is certain to be zero. The plus and the juxtaposition multiplication on the right hand side of the second line are overloaded to take distributions as operands and the value of the square root is lifted to a distribution as well. If we think of distributions as sets of pairs (x, p), where p is the probability for the distribution to take the value x, we can define overloading of the binary operators as

$$d_1 \oplus d_2 = \{(x \oplus y, pq) \mid (x,p) \in d_1, (y,q) \in d_2\} \tag{2}$$

With the brownian motion in hand we can model the price of a stock as the so called *standard underlying*, parameterized over the underlyings start price S, volatility v and the assumed continuous risk free interest rate r.

* This research has been supported by the Danish Strategic Research Council, Program Committee for Strategic Growth Technologies, for the research center 'HIPERFIT: Functional High Performance Computing for Financial Information Technology' (hiperfit.dk) under contract number 10-092299.

† This work was also presented at TFP 2012

FHPC'12, September 15, 2012, Copenhagen, Denmark.

[1] Hull[7] estimates the market size for derivatives to be above 600 trillions USD in June 2007.

$$U(t) \quad = \quad Se^{(r-\frac{1}{2}v^2)t+v\mathcal{W}(t)} \tag{3}$$

Let's return to the Asian option contract for a while. Models as the one above can be used to reason about the expected future value of the underlying stock at exercise time, but we also need to model the strike price being the arithmetic average on the stock price from now and one year forward. Once again the quantitative analysts write this using standard mathematical notation overloaded to distributions.

$$avg_C(t) \quad = \quad \frac{1}{t}\int_0^t C(t)dt \tag{4}$$

$avg_C(t)$ is the average of the process C from time 0 to time t. This notation allows us to write down the future profit from exercising the Asian option.

$$U(1) - avg_U(1) \tag{5}$$

But a rational holder will only exercise if the profit is positive, so the future payoff is therefore

$$max(0, U(1) - avg_U(1)) \tag{6}$$

This distribution models the uncertain price of the future payoff, but we are really interested in the current price. We therefore discount the distribution back one year adding the factor e^{-tr}, again assuming the continuous risk free interest rate r.

$$max(0, U(1) - avg_U(1)) \cdot e^{-1 \cdot r} \tag{7}$$

This expression models the price of the Asian option as a distribution using techniques found in mathematical finance. But we still haven't calculated its *expected value*, which is the average value of the distribution weighted by the probabilities. If the distribution is seen as a set of pairs (x_i, p_i) where $\Sigma p_i = 1$, then its weighted average is $\Sigma x_i p_i$, but as we are left with a continuous distribution, this calculation is not straight forward.

Numerous methods exist to calculate or estimate this expected value, but we haven't found a single efficient method being able to calculate expected values for nearly all the contract we could wish to express. The Monte Carlo method is however a method that is applicable for all the distributions that we can write using the normal distribution and the overloaded arithmetic we have just seen. The method is simply to take the average over a number of random samples from the distribution. This method is therefore only an approximation and to improve the accuracy one must increase the number of samples and thereby also the computational workload.

To take a random sample from the distribution modeling the price of the Asian option as declared in expression 7, we need to unroll the iterative definition of the brownian motion in expression 1, first deciding on a Δt. Let us for the sake of this example choose $\Delta t = \frac{1}{2}$. To avoid having to calculate the continuous average on the stock processes, we replace it with a discrete estimate using the Δt as the sample time interval.

$$\begin{aligned} b_0 &= \mathcal{W}(0) &&= 0 \\ b_{\frac{1}{2}} &= \mathcal{W}(\tfrac{1}{2}) &&= b_0 + \mathcal{N}\sqrt{\tfrac{1}{2}} \\ b_1 &= \mathcal{W}(1) &&= b_{\frac{1}{2}} + \mathcal{N}\sqrt{\tfrac{1}{2}} \end{aligned}$$

$$\begin{aligned} u_0 &= Se^{(r-\frac{v^2}{2})0+vb_0} &&= S \\ u_{\frac{1}{2}} &= Se^{(r-\frac{v^2}{2})\frac{1}{2}+vb_{\frac{1}{2}}} \\ u_1 &= Se^{(r-\frac{v^2}{2})1+vb_1} \end{aligned}$$

$$a \; = \; avg_U(1) \; = \; \frac{1}{1}\int_0^1 U(t)dt \; \approx \; \frac{1}{3}(u_0 + u_{\frac{1}{2}} + u_1)$$

$$s \; = \; max(0, u_1 - a) \cdot e^{-1 \cdot r}$$

We are now left with an expression s where the only distribution is the normal distribution. The sampling technique is then to replace all of these with a random sample from $\mathcal{N}$, thereby achieving a simple arithmetic expression on real numbers. It's however important that u_t is sampled only once and that this sampled value is used in both places where u_t is referred to. Otherwise, we would price the option as if the paid out stock price and the average strike price was based on two similar, but unrelated, stocks. To complete the simulation, we simply repeat this experiment, say, a million times, and calculate the average of the experiments.

At this point any programmer should feel comfortable implementing a pricer for Asian options as well as for many other contracts, if just given their price distribution expressed with the financial notation just seen. But how easy would it be to convince a domain expert afterwards that your implementation was in fact correct? How easy would it be to make changes to the contract being priced? These concerns calls for a DSL where one can specify the distributions and stochastic processes apart from the more complicated back-end code that finds expected values. And as we only need to implement a back-end once for all expressible distributions, we can afford the time to implement a efficient one running Monte Carlo simulation using massively parallel hardware, thereby remedying the fact that the Monte Carlo method is rather computationally heavy.

1.2 The Asian option in *SPL*

SPL is a DSL deeply embedded in Haskell designed for specifying stochastic processes and distributions. Processes and distributions are basic types, and there are a number of constructs for working on them. As we discussed in the previous section, when working with stochastic values, it's important to distinguish between working on the same sample of the stochastic value or different independent samples. For example, the Box–Muller [5] transform can produce a normal distribution from two *independant* uniform distributions, `u1` and `u2`. To talk about specific samples from a distribution, we use the *monadic bind* operator as in [3]:

```
normal :: Dist Real
normal = do
    u1 <- uniform
    u2 <- uniform
    return (sqrt (-2 * log u1) * cos (2 * pi * u2))
```

A stochastic process may, as noted above, be viewed as a function from time to a distribution. Another intuition is to watch a process as a distribution of *traces*, where a trace is a function from time (a real number) to a non-stochastic value. We choose this intuition in *SPL*, as this makes sampling from a classic distribution and obtaining a trace from a process the same thing, both achieved in *SPL* with the monadic bind. This gives reason for the following type aliases in *SPL*:

```
type Trace a = Time -> a
type Process a = Dist (Trace a)
```

We can construct processes by iterating a function, taking the value from the previous time step and producing a distribution of

values for the current time step. Recall the iterative definition of the Brownian motion in expression 1. In *SPL* this may be defined using the `iterate` construct, which is somewhat analogous to that of the Haskell standard library:

```
brownian :: Process Real
brownian = iterate 0 $ \w t dt -> do
    n <- normal
    return (w + n * sqrt dt)
```

In the above `w` is the previous value, `t` is the current time and `dt` is Δt. The relation to the definition of $\mathcal{W}$ should be clear.

To continue on the Asian option example, let us create a processes modelling the underlying stock corresponding to expression 3. Again the parameters are the initial stock price `s`, the continuous risk free interest rate `r` and the volatility `v`:

```
underlying :: Real -> Real -> Real -> Process Real
underlying s r v = do
    w <- brownian
    let u t = s * exp ((r - 0.5 * v^2) * t + v * w t)
    return u
```

Note how similar the `let` is to the definition of U. The rest of the code is about being specific about the parameters and the sampling, and a little bit of monadic overhead.

Recall that the price of the Asian option is in part based on the average of the underlying process. The `scan` construct allows such aggregation over the trace of a process and is analogous to `scanl` in Haskell. The following sums a trace and counts how many steps it has taken, and then divides the sum by the count to get the running average. Note that arithmetic is also available for traces of `Real`:

```
average :: Trace Real -> Trace Real
average p =
    let v = scan p (0, 0) $ \a v ->
            (first a + v, second a + 1) in
    first v / second v
```

With the average and a process modelling the underlying stock in hand, we can specify the Asian option price distribution equivalent to expression 7:

```
asian :: Real -> Real -> Real -> Time -> Dist Real
asian s r v t = do
    u <- underlying s r v
    return (max 0 (u t - average u t) * exp (-t * r))
```

Note that we use `u` *twice* here. It's thus important to make sure that both references are referring to the *same trace* of the `underlying` process, and indeed the monadic bind allows us to specify this. Contrast the mathematical notation used to specify U, where you have to guess from the context that this is the intention. Other than that, note how similar the right hand side of `return` is to the mathmatical notation.

Figure 1 plots a trace of `asian` along with the components that it's based on. It shows one possible outcome, but keep in mind that there are infinitely many to pick from.

To obtain the expected value of distributions specified in *SPL* we have implemented a back-end that simulates this result using the Monte Carlo method running on GPGPUs. The function `compile` compiles a *SPL* distribution into OpenCL[9] kernels and loads them so they are ready to be called. Besides from this side effect, `compile` returns another function that runs these kernels.

Simulating the estimated price for the Asian option could look as below, using 1 million simulations and a time step of one day's time. Note that we return both the expected value and the standard deviation (in that order, separated by $\pm$).

```
> expectedIO <- compile 1000000 (1/365)
                        (asian 0.8 0.4 0.05 1)
> expectedIO
0.08267 ± 0.1506
> expectedIO
0.08248 ± 0.1501
```

The reason for separating the compile and load phase from the invocation is that `compile` does not only take a distribution, it may also be called with a function that, given any number of `Real`-valued arguments, returns a distribution. This allows us to compile and load a kernel once and then invoke it several times with different parameters.

```
> expectedIO <- compile (1/365) asian
> expectedIO 0.8 0.4 0.05 1.0
0.08201 ± 0.1487
> expectedIO 50 0.2 0.05 0.5
1.976 ± 2.832
```

2. *SPL* - A stochastic process language

SPL is a language for specifying distributions and stochastic processes. Apart from the vocabulary of `Dist`, `Process` and `Trace`, *SPL* also has the primitive types `Real` and `Boolean`, as well as pairs. Additionally, there are types for talking about the length of time intervals (`Duration`) and absolute times (`Time`) measured as an offset from time zero:

```
type Duration = Real
type Time = Duration
```

By `Real` we do indeed mean the type embodying all real numbers $x \in \mathbb{R}$. Since $\mathbb{R}$ is continuous, we say that `Dist Real` is a distribution with *continuous sample space*. And since the `Time` is also $\mathbb{R}$, we say that `Process a` is an *infinite* stochastic process with *continuous time* and sample space.

This is by necessity; all of the examples of stochastic processes we have seen so far have had all of these properties. If say, we only had discrete time or discrete sample space, we would have to pick the discretization before we could specify those processes - and the specification would then only be valid for that particular choice of discretization.

For this reason, specification of processes and discretization and choice of computational model are seperate concerns in *SPL*. The latter decisions can be postponed until we want to get the expected value.

Figure 2 gives an almost complete list of the *SPL* constructs, most of which we have already seen used in the introduction. We allow the usual Haskell notation for constructing pairs, which internally uses the `pair` construct.

2.1 Embedding

SPL is deeply embedded in Haskell, meaning that *SPL* expressions are actually generating abstract syntax trees, which may then be translated to some other language, optimized and executed on another platform. In a later section we shall see an example of this using the OpenCL platform. To construct this AST we need all *SPL* expressions to be reifyable and thus cannot allow 'raw' type variables in the generated syntax tree.

```
data RealValue
type Real = Sample RealValue
type Boolean = Sample Bool
```

Haskell do not have a type for real numbers as they are not supported, so we define a new type `RealValue`, and let it be empty

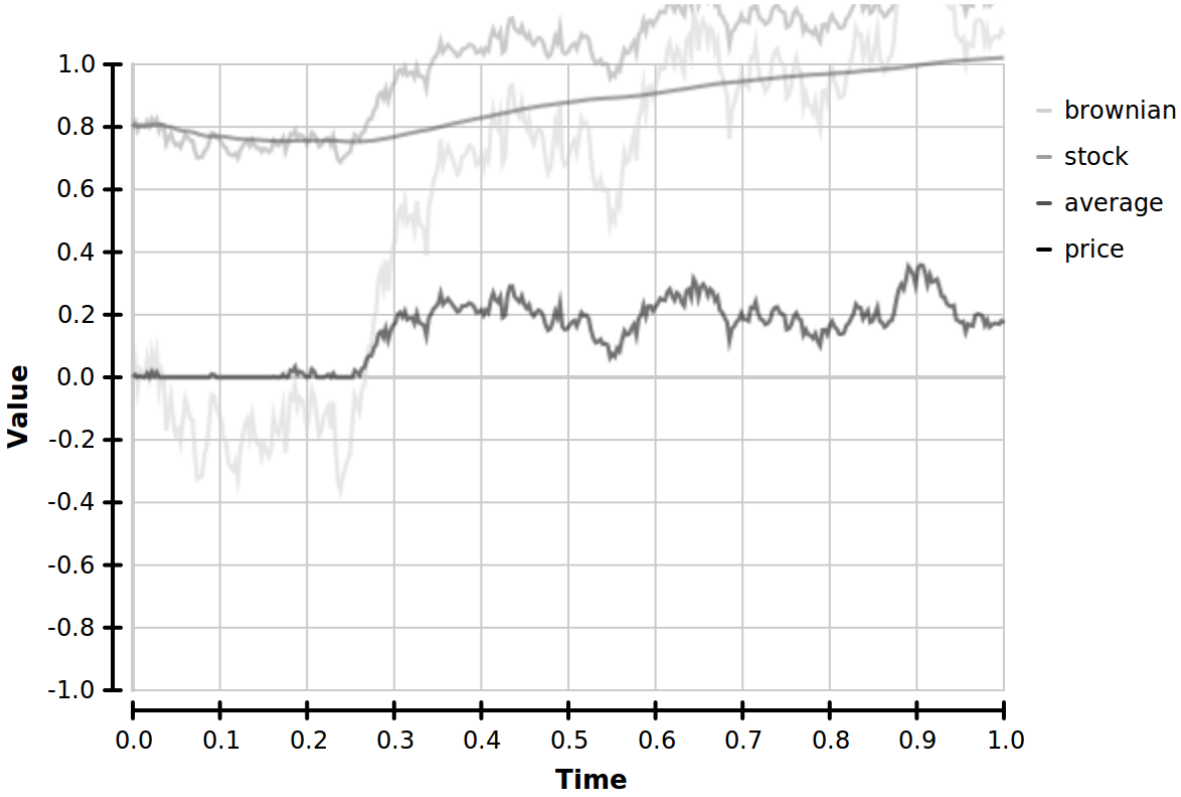


Figure 1. Illustrates how the `payoff` process in the definition of `asian r v s t` is constructed from a trace of the `brownian` process. The experiment was performed with the parameters `r` $= 0.05$, `v` $= 0.4$, `s` $= 0.8$ and $t = 0, 1/365 \ldots 1$. The stock trace is a function of the brownian trace starting at its initial price 0.8. The green average trace is the running average of the stock trace and the payoff is positive whenever the average is below the stock trace. The payoff is zero otherwise due to the `max 0`. The discounting lowers the payoff trace but only slightly as $e^{-1 \cdot 0.05} \approx 0.95$.

since we cannot use it from Haskell anyway. However, we can still map it to a discretization of our choice later on.

The support for real numbers is possible as *SPL* is purely symbolic and deeply embedded in Haskell. `Sample a` is our symbolic expression tree for non-stochastic values of type `a`, and we may for instance write:

```
irrational :: Real
irrational = sqrt 2 + pi
```

There's no loss of precision, because we simply get the syntax tree corresponding to the expression. It's only when we choose a computational model that we may loose precisions. The standard algebraic notation is achieved by overloading the standard arithmetic operators, ie:

```
instance Num Real where
    fromInteger = Real . fromInteger
    (+) = Binary Add
    ...
```

We do this for `Num`, `Fractional`, `Floating`, and a few new type classes for providing comparison, boolean logic, pairs and an if-statement called `when`. We lift most of these operators to traces. For example, the instance for `Num (Trace Real)` looks like:

```
instance Num a => Num (Trace a) where
    fromInteger i t = fromInteger i
    (+) a b t = a t + b t
    ...
```

Once we have the syntax tree, we can manipulate it and translate it to other syntax trees, such as that of OpenCL as we shall see later.

`Sample a` is shown in its full definition in figure 3. Fractional numbers and bool values from Haskell may be lifted to *SPL* value using the data constructors `Real` and `Boolean`. `Unary` and `Binary` allow standard arithmetic, comparisons, boolean logic and pairs. `Scan` aggregates over a trace taking an initial value and an aggregation function. This allows operations such as `average`.

`Distribution a` is our symbolic representation for stochastic values of type `a` corresponding to the exposed monad `Dist a`, having data constructors similar to monadic operation with the exception that `Bind` restricts the first type parameter to be `Symbolic`. This is required to ensure that the value can be reified.

We use restricted monads[2] to enable ordinary monadic facilities such as the do notation even though we have a constraint on our bind. Thus `Dist` is simply `Distribution` wrapped in `AsMonad` from `Control.RMonad.AsMonad`:

```
type Dist a = AsMonad Distribution a
```

3. Monte Carlo simulation on the GPU

While the syntax and types described so far is convenient for programming in *SPL*, it isn't obvious how to translate it to GPU code that does Monte Carlo simulation.

[2] Using the rmonad-0.7 package on Hackage.

```
Dist a

    instance Functor Dist where
        fmap    :: (a -> b) -> Dist a -> Dist b

    instance Monad Dist where
        return  :: a -> Dist a
        (>>=)   :: Dist a -> (a -> Dist b) -> Dist b

    uniform     :: Dist Real
    iterate     :: a -> (a -> Time -> Duration -> Dist a) -> Dist (Time -> a)

Sample a

    pair        :: Sample a -> Sample b -> Sample (a, b)
    first       :: Sample (a, b) -> Sample a
    second      :: Sample (a, b) -> Sample b
    abs,...     :: Sample a -> Sample a
    +,*,/,...   :: Sample a -> Sample a -> Sample a
    .==.,...    :: Sample a -> Sample a -> Sample Bool
    .&&.,...    :: Sample a -> Sample a -> Sample Bool
    when        :: Sample Bool -> Sample a -> Sample a -> Sample a
    scan        :: Trace b -> a -> (Like a -> Like b -> a) -> Trace (Like a)
```

Figure 2. The built-in constructs of *SPL*

```
data Sample :: * -> * where
    Real    :: (Fractional a) => a -> Sample RealValue
    Boolean :: Bool -> Sample Bool
    Unary   :: UnaryOperator a b -> Sample a -> Sample b
    Binary  :: BinaryOperator a b c -> Sample a -> Sample b -> Sample c
    If      :: Sample Bool -> Sample a -> Sample a -> Sample a
    Scan    :: Sample (Time -> b)
            -> Sample a
            -> (Sample a -> Time -> Duration -> Sample b -> Sample a)
            -> Sample (Time -> a)

data Distribution :: * -> * where
    Uniform :: Distribution (Sample RealValue)
    Iterate :: a -> (a -> Time -> Duration -> Distribution a) -> Distribution (Time -> a)
    Bind    :: Symbolic a => Distribution a -> (a -> Distribution b) -> Distribution b
    Return  :: a -> Distribution a
```

Figure 3. The complete *SPL* abstract syntax tree. None of the data constructors are exported directly to the *SPL* user, the *SPL* module provides alternative *smart*-constructors that should be more convenient to use.

In particular, we'd like to be able to take any `Distribution Real` and perform pseudo-random sampling from it in order to approximate the expected value. We target OpenCL which is a limited C-like language that in turn targets GPUs. However, this language has no concept of distributions, and as such we will have to translate that concept into something else.

It also has no concept of functions, traces or time, so these concepts will have to be mapped too. In the following we will describe how we overcome all of these obstacles.

3.1 From monads to side effects

In order to get rid of the restricted monad and eliminate the concept of distributions, we introduce an internal language. This langauge is an extension of `Sample` consisting of several additional constructors:

```
data Internal :: * -> * where
    -- All the constructors of Sample a, and:
    Trace :: (Time -> Sample a) -> Sample (Time -> a)
    Lookup :: Sample (Time -> a) -> Time -> Sample a
    Let :: Sample a -> (Sample a -> Sample b) -> Sample b
    Variable :: Int -> Sample a
    GenerateUniform :: Sample RealValue
    IterateLoop
        :: Sample a
        -> (Sample a -> Time -> Duration -> Sample a)
        -> Sample (Time -> a)
```

The constructs in here will be explained as they are needed. All of the functions that are required to go from a `Distribution` to `Internal` are provided by the type class `Symbolic`. Namely:

```
class Symbolic a where
    type Result a
    symbolic     :: a -> Internal (Result a)
    function     :: (a -> b) -> Internal (Result a) -> b
    distribution :: Distribution a -> Internal (Result a)
```

The `Result a` embodies the inner type of the translated syntax tree. The `symbolic` function translates fully applied values, `function` translates functions and `distribution` translates `Distribution`s.

For `Sample a`, `symbolic` and `function` are simply identity functions since `Internal a` is a superset of `Sample a`. For distributions, the uniform distribution becomes a value that when read has the side effect of generating a pseudo-random number from the distribution. For return and bind, the functions are simply applied recursively, although the strict `Let` is introduced in the case of `Bind` to ensure that the side effects, of the sampling the distribution, happens only once.

```
instance Symbolic (Sample a) where
    type Result (Sample a) = a
    symbolic = id
    function = id
    distribution d = case d of
        Uniform -> GenerateUniform
        Return a -> symbolic a
        Bind a f -> Let (distribution a) $
            \a -> distribution (function f a)
```

The case for pairs is straightforward:

```
instance Symbolic (Sample a, Sample b) where
    type Result (Sample a, Sample b) = (a, b)
    symbolic (a, b) = Binary Pair (symbolic a) (symbolic b)
    function f x = f (Unary First x, Unary Second x)
    distribution = distribution . fmap symbolic
```

The case for traces is a bit more involved. Namely we have `Trace` and `Lookup` converting to and from `Sample (Trace a)`. The `Iterate` construct is simply translated to the corresponding loop `IterateLoop` in `Internal`.

```
instance Symbolic (Time -> Sample a) where
    type Result (Time -> Sample a) = Time -> a
    symbolic f = Trace f
    function f x = f (Lookup x)
    distribution d = case d of
        Iterate z f -> IterateLoop (symbolic z) $
            \a t dt -> distribution (function f a t dt)
        Return a -> symbolic a
        Bind a f -> Let (distribution a) $
            \a -> distribution (function f a)
```

Thus we have eliminated `Distribution a` and are left with `Internal a` that has no "naked" type variables nor custom type constraints.

3.2 Removing higher order abstract syntax

We remove the functions from our syntax tree by converting to a typed De-Bruijn indexed representation with a type safe homogenous list constructed from iterated pairs as the environment. This is a type preserving translation as described in [2] and is what the `Variable` is for.

3.3 Generating pseudo-random numbers

We use the MWC64X pseudo-random number generator for OpenCL described in [11]. We implement `split` (analogous to the one in Haskell) by randomizing the internal state of the PRNG:

```
struct generator_t split(struct generator_t * generator)
{
    struct generator_t new_generator;
    new_generator.state.x =
        MWC64X_NextUint(&(generator->state));
    new_generator.state.c =
        MWC64X_NextUint(&(generator->state));
    return new_generator;
}
```

3.4 Representing trace values

One way of representing traces is to compute the values at all discretized time steps and store them in an array for later use. However, this would require us to know the number of traces and the maximum lookup time before runtime since OpenCL does not allow runtime allocation of memory. It would also waste a lot of space on storing intermediate values that will never be accessed again, and it's probably safe to assume that accessing arrays is slower than not accessing arrays, all else being equal.

We therefore choose to trade processer time for memory by recalculating the trace whenever we need it. When we encounter a `Let` on an `Internal (Trace a)`, we `split` the current pseudo-random number generator into two and store the additional generator in the environment. When encountering a `Variable` of type `Internal (Trace a)` we run the code for computing the trace locally using the generator pointed by the variable as the current generator.

3.5 Code generation for `Lookup`s on traces

We translate both the `Scan` and `IterateLoop` constructs at each usage spot by first initializing the accumulator variable to the initial value, and then updating the accumulator variable according to the update function inside a loop. The loop is an ordinary for loop that starts at time zero and counts up to some time t by Δt. Everything inside `f` in a `Lookup f t` that isn't below a deeper `Lookup f' t'` where $\texttt{t} \neq \texttt{t'}$ is updated in the same single for loop where $t = \texttt{t}$. The initialization and updates are performed corresponding to their position in postorder tree traversal, thus updating dependees before dependers. This is essentially loop fusion.

3.6 Potentially over-aggressive inlining

While the rest of the translation to OpenCL is straightforward, note that we do not attempt to recover sharing introduced by Haskell's bindings. Consequently these are inlined at all usage sites, which can lead to code explosion[3]. The Haskell compiler GHC offers a mechanism to detect sharing of values using the IO monad, and using this it would be possible to detect shared functions and generate them as functions in the target language, as in [6, 10]. However, we have left this optimisation for future work.

3.7 Execution and aggregation

For the variadic `compile`, we generate a kernel that takes the corresponding extra parameters of type `double`. The code in the kernel performs a single experiment, thus contributing one sample per thread to the complete Monte Carlo simulation.

Local memory is used at the end of each experiment to store the individual result. Once all threads are finished in a OpenCL thread group, we run another threaded program that aggregates the results, computing the average (expected value) and the standard deviation, and then stores these in global memory. These results are then aggregated by the host once all groups are finished, yielding a single pair `(expectedValue, standardDeviation)` as the final result.

Note that the final aggregation is the only time at which we use shared memory. Everything else uses purely local variables, which are likely mapped to registers. There is thus no potential for so-called bank conflicts or other memory related synchronization issues during the execution of an experiment.

3.8 Properties of the translation

In order to take full advantage of the OpenCL device, there are certain pitfalls that needs to be avoided, or some of the hardware threads will spend their time waiting. Memory must be accessed using specific patterns, or the access will be serialised due to so-called bank conflicts. Due to the SIMD architecture, conditionally executed code such as loops and if-statements will cause all the threads to wait for each other to execute the instructions that they shouldn't execute themselves. In order to make it easy to reason about the performance of *SPL* programs, we therefore guarantee the following properties of our translation:

- It does not introduce any additional conditional logic beyond that which is present in the source *SPL* program, which is only the `when` construct.
- It does not produce any load, store or synchronisation operations, except at the end of a simulation where the result is written to a buffer.

We achieve the second point by using local variables for all our storage. This is straightforward since all our data structures are fixed-size. This means that we can roughly reason about the performance an *SPL* program

[3] Though we have not yet experienced this in practice.

like this: every time you apply a trace to a new time, it creates a loop; every time you use `when` it performs as if both branches were executed; and everything else is constant time.

While this is a conservative approach (e,g,. `when true b c` never spends resources on computing `c`), it's also very straightforward to apply - indeed, one could write a function that took an *SPL* program and computed an asymptotic upper bound on the run-time according to the above definition.

4. Benchmark

We present here two experiments that tests the scalability and performance of the OpenCL back-end. The result of the first experiment indicates that we do in fact obtain the linear speedup we had hoped for. The second experiment shows that our OpenCL back-end runs 100 times faster than our own hand written sequential C code, on the hardware used. More tests and benchmarks are presented in [4].

The hardware used was a machine with one 2.67 GHz Intel Xeon X5550 CPU and one Tesla C2050 device. The machine is running Linux with CUDA 1.0 as the OpenCL implementation and the NVidia drive version 275.21. We run the C code on the CPU and expect only one of the 16 cores to be used. The OpenCL programs is only using the GPU.

It is common when testing whether a program scales, to investigate how the number of used cores influence the execution time for a fixed amount of work. In this scenario full scalability or utilisation is achieved when doubling the amount of cores halves the execution time.

We do however not know how to tell OpenCL to only use a certain amount of the cores on the device but we can exploit the fact that threads in one work group all are scheduled on the same multiprocessor. The Tesla C2050 card used in our benchmarks have 14 multiprocessors on board which means that running a kernel with a work group count of 14 should take no longer than having only 1, if full utilisation is meet. Running 15 work groups on the other hand would take twice as long as the last work group would have to wait until the other 14 groups had been executed in parallel. Fortunately, this is exactly the picture we see on in figure 4. It is not surprising to achieve full scalability as Monte Carlo simulations is *embarrassingly parallel* in nature.

The contract priced in the experiment is an Asian Call option with an exercise time of 5. The simulation is carries out using a time step of $1/356$ to simulate a step every day for five years, or 1825 points in time in total. In OpenCL terms we have used a work group size of 512 and a group count from 2 to 42, yielding from 1024 to 21504 simulations per data point.

Figure 5 shows a speedup plot compared against a handwritten sequential C implementation of Monte Carlo simulation specialised for the same Asian option used above, just with a shorter exercise time of one years time. As we can see in the graph, the generated OpenCL simulation running on the GPU is about 100 times faster than the sequential C implementation. A quick back-of-the-envelope comparing just processor speed says that we could expect a speedup of about $448 \cdot 1.15/2.67 \simeq 193$ as the C2050 has 448 cores each running at 1.15 GHz - thus we are in the right ballpark.

5. Related Work

While stochastic processes are well suited to express the *price* of financial contracts, they are perhaps not ideal for expressing the *intention* of financial contracts. For example, it may be hard to automate trading based on the stochastic processes, since choices like the option of doing a or b is lost in the translation to the price $max(a, b)$.

Peyton Jones, Eber, and Seward previously presented a language for writing down financial contracts in a high level and compositional manner[8], which focuses on the intention of the contracts. It uses a logical `or` construct to express the choice between two actions, making applications such as automated trading possible.

The price for these contracts is given as a function that takes a contract and a financial model and produces a stochastic process. *SPL* can express the prices of a subset of this language, namely those contracts that do not require nested forecasting[4]. Conversely, *SPL* supports some financial contracts that are not supported by the contract language, namely path-dependent financial contracts such as Asian options.

[4] American and Bermudan options are examples of financial contract that requires nested forecasting.

While we have benchmarks showing the absolute performance of the GPGPU backend[4], we do not yet have any relative benchmarks comparing our performance to alternative pricers for financial contracts. QuantLib [1] is a state of the art pricer for a pre-defined set of financial contract types, and it would be interesting to see a comparison to this library.

6. Conclusion

We have presented the language *SPL* designed to model distributions and stochastic processes and shown how the price of an Asian option can be expressed in *SPL* while comparing to how it is often done in the field of mathematical finance. The syntax of *SPL* is closely related to the mathematical notation, which is achieved by using overloaded arithmetic on samples and traces and by allowing lookup on traces by standard function application with the time as the argument. The main difference, in notation, is that *SPL* have a clear distinction between samples and distributions and do not allow arithmetic directly on distributions. We made the deviation from the mathematical notation to avoid certain errors, like forgetting to trace the underlying process before using it twice in an Asian option.

With *SPL* comes an OpenCL back-end that simulates the expected value of distributions using Monte Carlo simulation. We have sketched how the translations from *SPL* to OpenCL is performed.

Acknowledgments

The work presented in this article in mainly developed as a master thesis by the two first authors[4], during that work we have received generous guidance from multiple members of the HIPERFIT research center. In particular, Mogens Steffensen (Department of Math, University of Copenhagen), Carl Balslev Clausen (SimCorp), Martin Elsman (then SimCorp, now DIKU/HIPERFIT center), Dirk Bangert (Bangert Research) and Rolf Poulsen (Department of Math, University of Copenhagen). Brian Vinter (Niels Bohr Institute, University of Copenhagen) helped us with access to the hardware needed for the benchmark experiments. Likewise, David B. Thomas, Geoffrey Mainland, Manuel M. T. Chakravarty and Martin Dybdal were all helpful in providing early access to and answering inquiries about their libraries, which were used in experiments (not all reported in this paper).

References

[1] F. Ametrano, L. Ballabio, M. Bianchetti, N. D. Césaré, D. Eddelbuettel, N. Firth, N. Jean, C. Kenyon, R. Lichters, M. Marchioro, K. Spanderen, and J. Wang. QuantLib. `http://quantlib.org/index.shtml`, 2011. URL `http://quantlib.org/index.shtml`.

[2] M. M. T. Chakravarty. Converting a hoas term gadt into a de bruijn term gadt. `http://www.cse.unsw.edu.au/~chak/haskell/term-conv/`, 2009. URL `http://www.cse.unsw.edu.au/~chak/haskell/term-conv/`.

[3] M. Erwig and S. Kollmansberger. Functional pearls: Probabilistic functional programming in haskell. *J. Funct. Program.*, 16:21–34, January 2006. ISSN 0956-7968. doi: 10.1017/S0956796805005721. URL `http://portal.acm.org/citation.cfm?id=1114008.1114013`.

[4] M. Flænø Werk and J. Ahnfelt-Rønne. Pricing composable contracts using GPGPU. Master's thesis, Department of Computer Science, University of Copenhagen, 2011.

[5] George and M. E. Muller. A note on the generation of random normal deviates. *Ann. Math. Stat.*, 29(2):610–611, 1958.

[6] A. Gill. Type-safe observable sharing in haskell. In *Proceedings of the 2009 ACM SIGPLAN Haskell Symposium*, Sept. 2009. URL `http://www.ittc.ku.edu/csdl/fpg/sites/default/files/Gill-09-TypeSafeReification.pdf`.

[7] J. Hull. *Options, futures and other derivatives*. Prentice Hall finance series. Pearson/Prentice Hall, 2009. ISBN 9780136015864. URL `http://books.google.com/books?id=sEmQZoHoJCcC`.

[8] S. P. Jones, J.-M. Eber, and J. Seward. Composing contracts: an adventure in financial engineering (functional pearl). In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP '00, pages 280–292, New York, NY, USA, 2000.

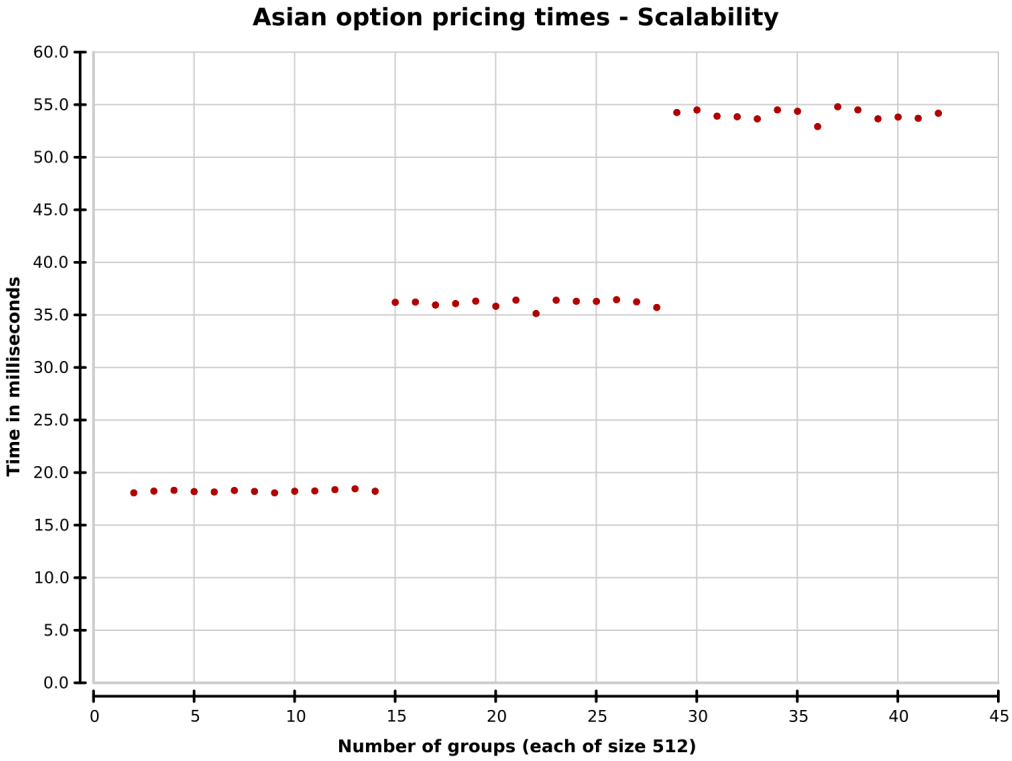
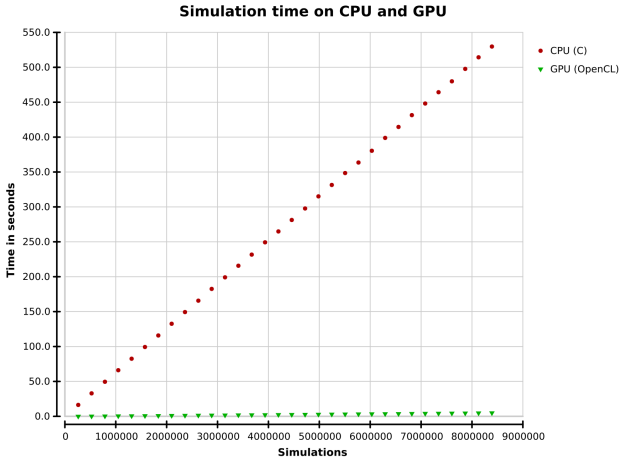
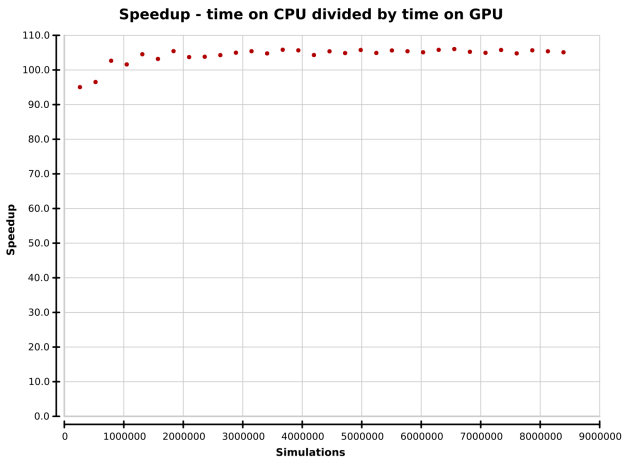


Figure 4. This graph shows the pricing times for an Asian call option valuated using 1024 to 21504 simulations. The number of simulations are indexed by the work group count to show the correlation with the 14 multiprocessors on the Tesla C2050 device. The graph shows that we may add simulations from additional work groups without increasing the executions time until the point where the work groups count reach the next multiple of 14. Each pricing experiment have been performed 101 times.



(a) The wall-clock time for the sequential C program is shown shown with the circle graph. Each data point is the median value of 13 simulations. The triangle graph shows the time for compiling, loading and execution the similar *SPL* program. Each data point is the median value of 101 simulations.



(b) Speedup compared to our sequential C implementation.

Figure 5. We compare the simulation time of the *SPL* OpenCL backend with sequential hand written C code performing the same Monte Carlo simulation. The OpenCL back-end uses a Tesla C2050 device while the C code are executed on a Intel Xeon X5550, compiled with gcc 4.4.6 with `-O3`. The simulation prices an Asian option with an exercise time of 1 and a time step of $1/365$.

ACM. ISBN 1-58113-202-6. doi: http://doi.acm.org/10.1145/351240.351267. URL http://doi.acm.org/10.1145/351240.351267.

[9] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008. URL http://khronos.org/registry/cl/specs/opencl-1.0.29.pdf.

[10] Simon, S. Marlow, and C. Elliott. Stretching the Storage Manager: Weak Pointers and Stable Names in Haskell. In *Implementation of Functional Languages*, pages 37–58, 1999. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.34.1948.

[11] D. B. Thomas. The MWC64X random number generator. http://www.doc.ic.ac.uk/~dt10/research/rngs-gpu-mwc64x.html, 2011. URL http://www.doc.ic.ac.uk/~dt10/research/rngs-gpu-mwc64x.html.