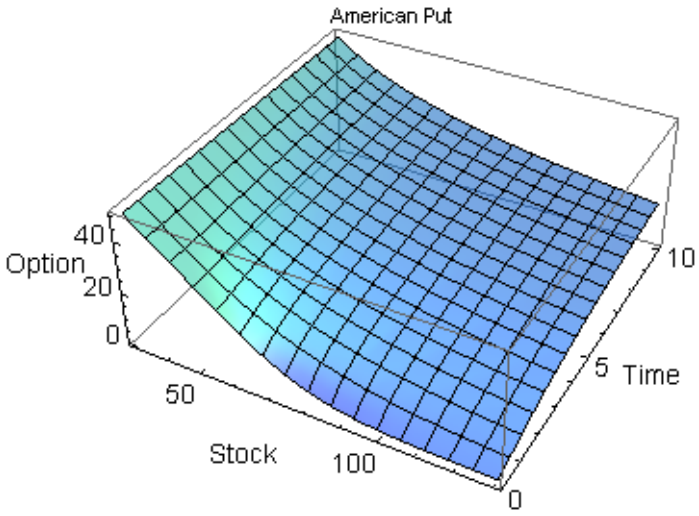
Wolfram Finance Platform

Mathematica as a Practical Platform for GPU-Accelerated Finance

Abdul Dakkak, abduld@wolfram.com

Kernel Developer
Wolfram Research Inc.



"Using parallel programing over CPU and computation with GPU to evaluate an American Put."

What is Wolfram Finance Platform

Innovative automation

Produce expert results without expertise

Integrated all-in-one platform

Save time and money, and access greater innovative capabilities compared to standalone tools

Hybrid numeric-symbolic methodology

Get more accurate results than from any other tool

Multi-paradigm language

Compute as you think and save time

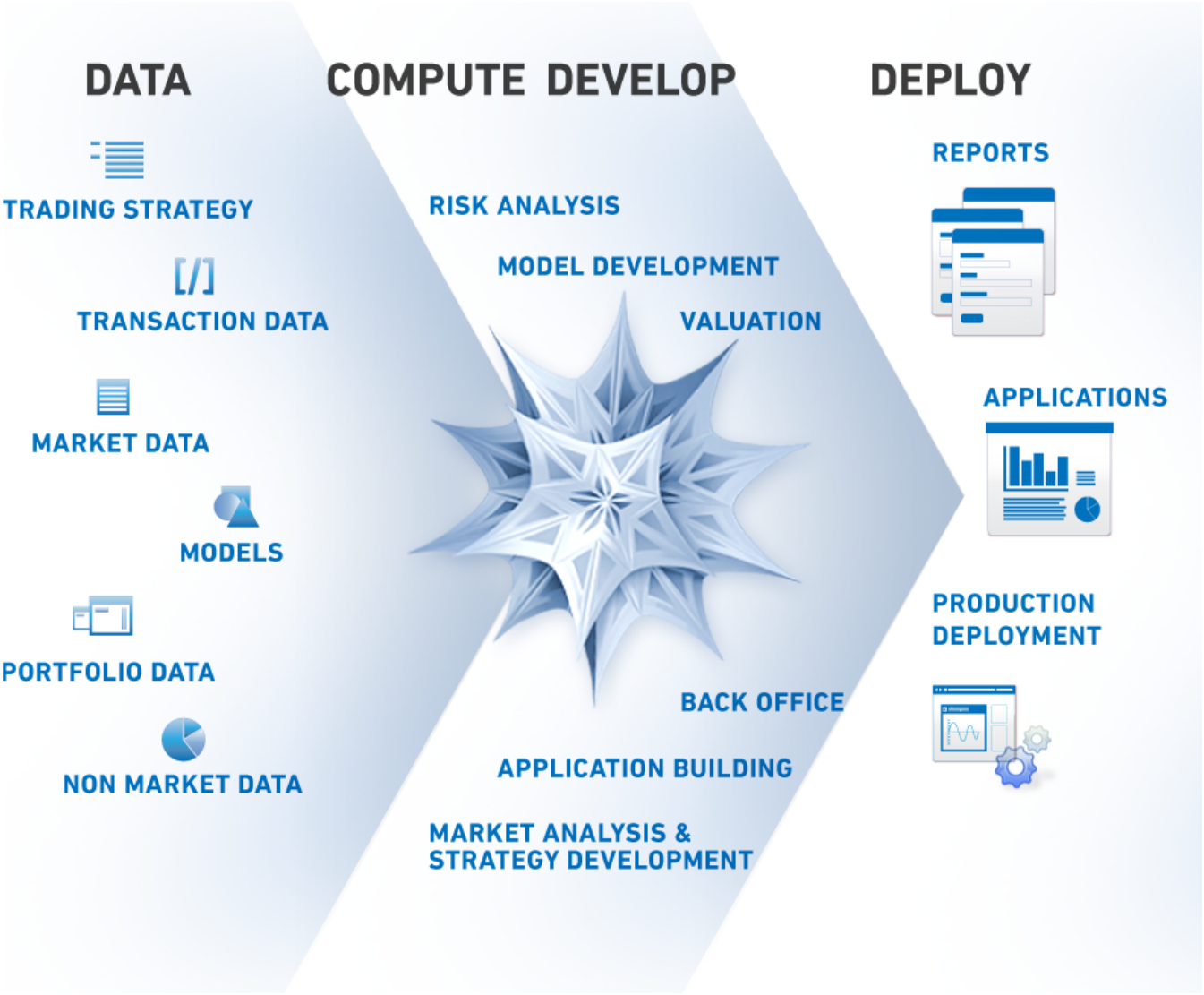
Built-in knowledge

Access the world's largest pool of up-to-date computable knowledge

Document centric workflow

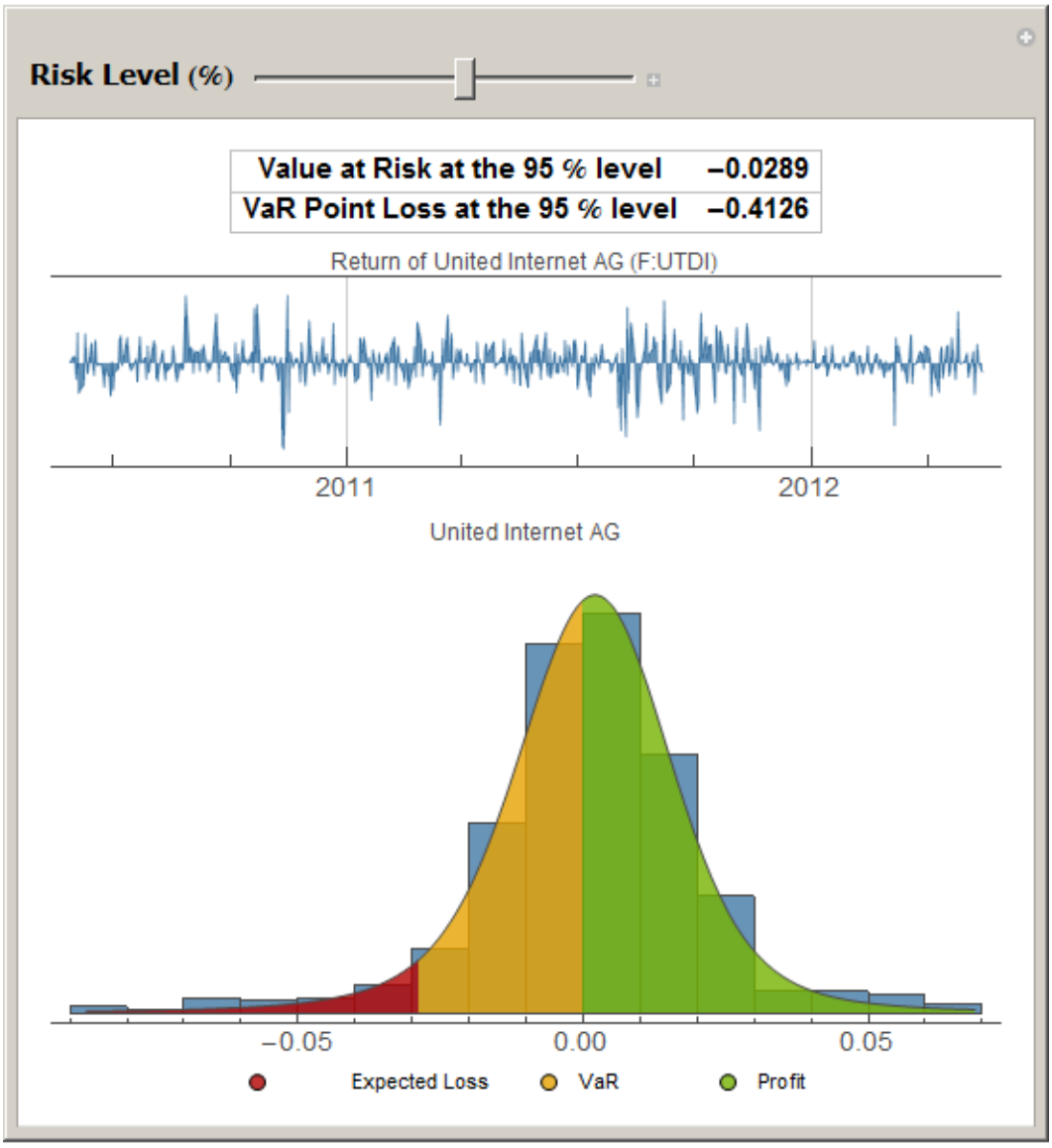
Document your work as you go, organize explanations, data, code, and outputs in a single file

Basic Idea of Wolfram Finance Platform



Calculate Value at Risk on the Fly

```
DynamicModule[{eRetdist, H, VaR, varPointLoss, α, β, μ, σ},

 Manipulate[
  eRetdist = EstimatedDistribution[retF[[1, All, 2]], StableDistribution[1, α, β, μ, σ]];
  VaR = InverseSurvivalFunction[eRetdist, risk * 0.01]; varPointLoss = priceF[[1, All, 2]][[-1]] (Exp[VaR] - 1);

  Grid[{{Style[Grid[{{"Value at Risk at the " <> ToString[risk] <> " % level", PaddedForm[VaR, {6, 4}]},
         {"VaR Point Loss at the " <> ToString[risk] <> " % level", PaddedForm[varPointLoss, {6, 4}]}},
        Frame → {False, All}, FrameStyle → GrayLevel[.75]], Bold, wfpfont, 15]}, {DateListPlot[retF[[1]],
       PlotLabel → "Return of " <> ToString@nameF[[1]] <> " (" <> ToString@symbF[[1]] <> ")", datelistplotsty]},
    {Labeled[
      Show[
       Histogram[retF[[1, All, 2]], Automatic, "PDF", PlotLabel → nameF[[1]], histogramsty],

       Plot[{PDF[eRetdist, x], Which[VaR ≤ x ≤ 0, PDF[eRetdist, 0]]}, {x, Min[retF[[1, All, 2]]], VaR},
        Evaluate[biggerplotsty], Filling → {1 → 0}, FillingStyle → expectedlosscolor],

       Plot[{PDF[eRetdist, x]}, {x, VaR, 0}, Evaluate[biggerplotsty], Filling → {1 → 0}, FillingStyle → varcolor],

       Plot[{PDF[eRetdist, x], Which[VaR ≤ x ≤ 0, PDF[eRetdist, 0]]},
        {x, 0, Max[retF[[1, All, 2]]]}, Evaluate[biggerplotsty], Filling → {1 → 0}, FillingStyle → profitcolor]



      ], histlabels]}
   }],

  {{risk, 95, Style["Risk Level (%)", Bold, 15]}, 90, 99, .05}]]
```



Model Development - Symbolic Model for Yield Curve

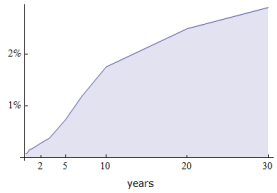
US Treasury Yield

Input interpretation:

United States | treasury yield curve

Results:

More



(Tuesday, May 15, 2012)

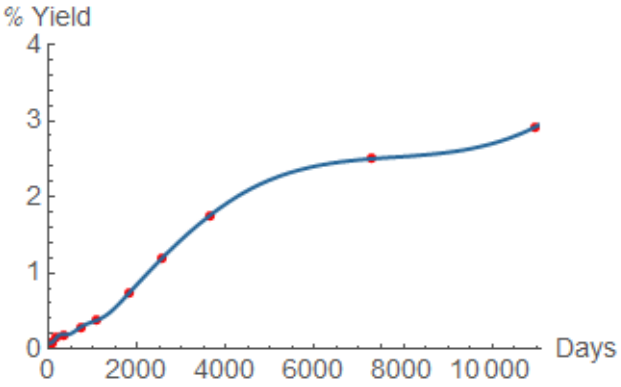
3-month treasury bill	0.09%
1-year treasury bill	0.19%
2-year treasury note	0.29%
5-year treasury note	0.74%
10-year treasury note	1.76%
30-year treasury bond	2.91%

WolframAlpha

```
interpIntRates = Interpolation[intRates, Method → "Spline"]
```

```
InterpolatingFunction[{{30., 10 950.}}, <>]
```

```
Row[{Show[ListPlot[intRates, AxesLabel → {"Days", "% Yield"}, AxesStyle → Darker@Gray,
     PlotStyle → {Red, PointSize[0.02]}, PlotRange → {{0, 11 100}, {0, 4}}, BaseStyle → wfpfont, ImageSize → 320],
    Plot[interpIntRates[t], {t, 0, 11 100}, PlotStyle → Directive[Thick, wfpblue[1]]]], Spacer[10],
   Style[Grid[Prepend[intdata, {"US Treasury Yield", SpanFromLeft}], Dividers → {False, All}], Darker@Gray, 10, wfpfont]}]
```



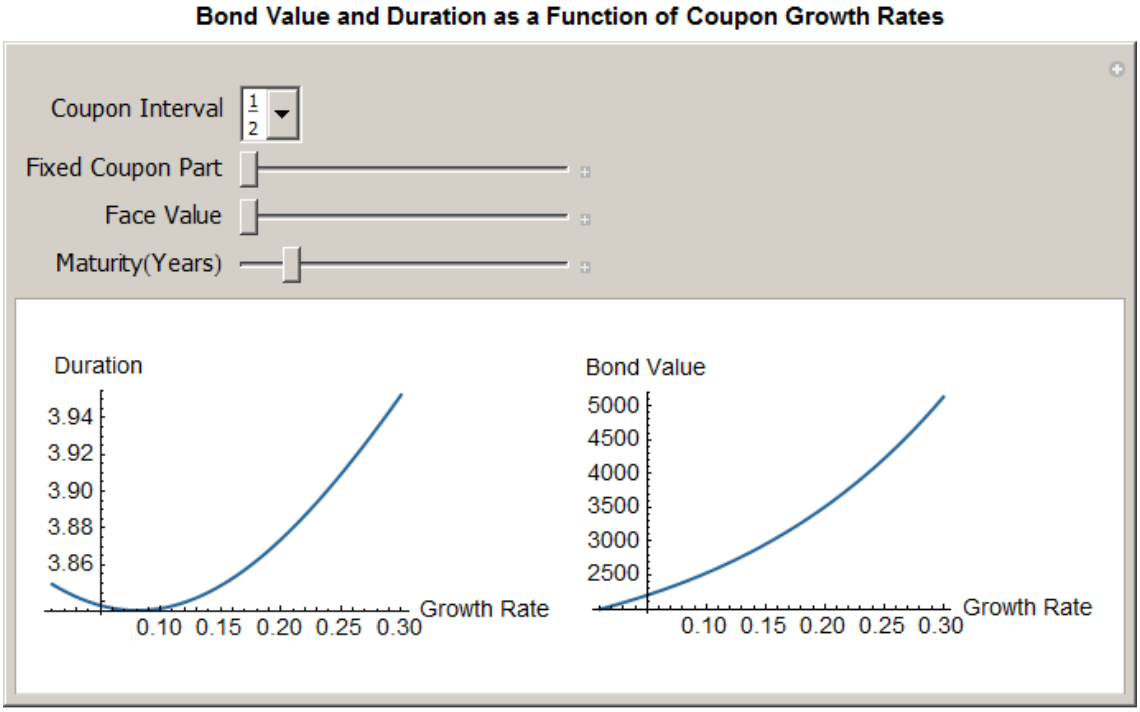
US Treasury Yield	
1-month treasury bill	0.08
3-month treasury bill	0.09
6-month treasury bill	0.15
1-year treasury bill	0.19
2-year treasury note	0.29
3-year treasury note	0.38
5-year treasury note	0.74
7-year treasury note	1.19
10-year treasury note	1.76
20-year treasury bond	2.5
30-year treasury bond	2.91

Convert above list of pairs into a list of rules giving the yield curve specification, say a corporation wants to issue a 7 year bond

```
FinancialBond[{"FaceValue" → 1000, "Coupon" → 0.1, "Maturity" → {2019, 5, 10}, "CouponInterval" → 1/2},
 {"InterestRate" → Apply[Rule, Map[{First[#] / 365, #[[2]] / 100} &, intRates], {1}],
  "Settlement" → {2012, 5, 10}, "DayCountBasis" → "Actual/365"}, {"Value", "Duration"}]
```

```
{1600.85, 5.59701}
```

```
Labeled[Module[{dur, val}, Manipulate[{dur, val} =
    FinancialBond[{"FaceValue" → par, "Coupon" → (c (1 + g)^(#/d-1) &), "Maturity" → DatePlus[Take[DateList[], 3], {n, "Year"}],
      "CouponInterval" → d}, {"InterestRate" → Apply[Rule, Map[{First[#] / 365, #[[2]] / 100} &, intRates], {1}],
      "Settlement" → Take[DateList[], 3], "DayCountBasis" → "Actual/365"}, {"Duration", "Value"}]; Quiet[
    GraphicsRow[{Plot[dur, {g, .01, .3}, AxesLabel → {"Growth Rate", "Duration"}, PlotStyle → Directive[Thick, wfpblue[1]],
       BaseStyle → wfpfont], Plot[val, {g, .01, .3}, AxesLabel → {"Growth Rate", "Bond Value"},
       PlotStyle → Directive[Thick, wfpblue[1]], BaseStyle → wfpfont]}, ImageSize → {640, 210}]],
   {{d, 1 / 2, "Coupon Interval"}, 1 / 6, 1, 1 / 6, ControlType → PopupMenu}, {{c, 100, "Fixed Coupon Part"}, 100, 300},
   {{par, 1000, "Face Value"}, 1000, 10 000}, {{n, 5, "Maturity(Years)"}, 1, 30}]],
 Style["Bond Value and Duration as a Function of Coupon Growth Rates",
  15, Bold, wfpfont], Top]
```



CUDALink and *OpenCLLink*

The Black–Scholes equation can be implemented using *CUDALink* with the following simple kernel.

```
code = "
#define N(x)        (erf((x)/sqrt(2.0))/2+0.5)
__global__ void blackScholes(Real_t * call, Real_t
     * S, Real_t * X, Real_t * T, Real_t * r, Real_t * q, Real_t * sigma, mint length) {
    int ii = threadIdx.x + blockIdx.x*blockDim.x;
    if (ii < length) {
        Real_t d1 = (log(S[ii]/X[ii])+(r[ii]-q[ii]+(pow(sigma[ii],(Real_t)2.0)/2)*T[ii]))/(sigma[ii]*sqrt(T[ii]));
        Real_t d2 = d1 - sigma[ii]*sqrt(T[ii]);
        call[ii] = S[ii]*exp(-q[ii]*T[ii])*N(d1) - X[ii]*exp(-r[ii]*T[ii])*N(d2);
    }
}";

bs = CUDAFunctionLoad[code, "blackScholes", {{_Real}, {_Real, "Input"},
   {_Real, "Input"}, {_Real, "Input"}, {_Real, "Input"}, {_Real, "Input"}, {_Real, "Input"}, _Integer}, 128]
```

```
CUDAFunction[<>, blackScholes, {{_Real}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, _Integer}]
```

```
bs[{0.0}, {35.0}, {35.0}, {1.0}, {0.08}, {0.02}, {0.3}, 1]
```

```
{{5.04898}}
```

```
FinancialDerivative[{"European", "Call"}, {"StrikePrice" → 35, "Expiration" → 1},
 {"CurrentPrice" → 35, "InterestRate" → 0.08, "Dividend" → 0.02, "Volatility" → 0.30}]
```

```
5.04898
```

The function can then be called with the following data.

```
numberOfOptions = 32;
call = ConstantArray[0.0, numberOfOptions];
spotPrices = RandomReal[{25.0, 35.0}, numberOfOptions];
strikePrices = RandomReal[{20.0, 40.0}, numberOfOptions];
expiration = RandomReal[{0.1, 10.0}, numberOfOptions];
interest = RandomReal[{0.03, 0.07}, numberOfOptions];
dividend = RandomReal[{0.01, 0.04}, numberOfOptions];
volatility = RandomReal[{0.10, 0.50}, numberOfOptions];

bs[call, spotPrices, strikePrices, expiration, interest, dividend, volatility, numberOfOptions]
```

```
{{4.97615, 12.1326, 3.29129, 5.93951, 5.60493, 5.44203, 12.2797, 9.44719, 5.30445, 9.77898, 8.48708, 6.36197, 7.73191, 3.07612, 4.83732, 6.41501,
  4.75431, 3.9362, 9.051, 1.8371, 6.94856, 0.00122445, 14.047, 3.54369, 12.0045, 7.35156, 7.93202, 7.03639, 10.5494, 5.16126, 2.86671, 2.75056}}
```

FinancialDerivative

What is FinancialDerivative?

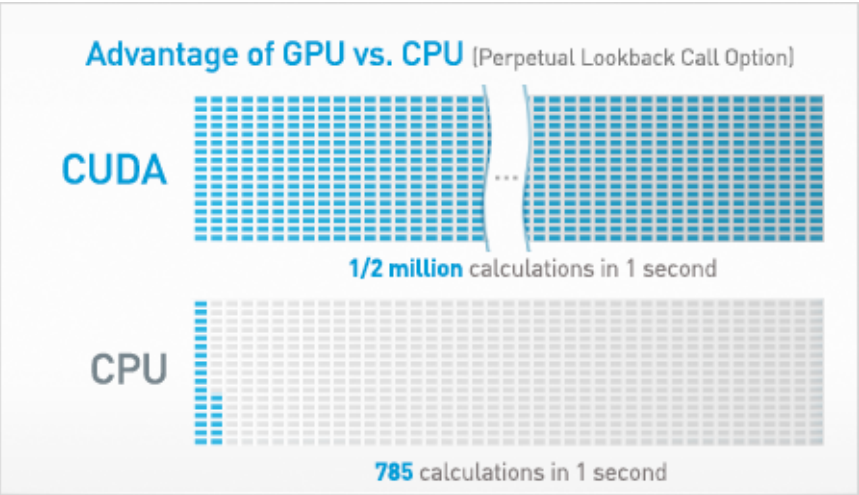
- A superfunction introduced in *Mathematica* 8 dedicated to pricing and calculating sensitivities of various options.
- Supports 95 styles of options, most with both American and European exercise style.
- The underlying model for all options supported by FinancialDerivative is the Black-Scholes-Merton model.
- Automatically CUDA-accelerated without change of usage or syntax.

```
FinancialDerivative[{"European", "Call"}, {"StrikePrice" → 50.00, "Expiration" → 1},
 {"InterestRate" → 0.1, "Volatility" → 0.5, "CurrentPrice" → 50, "Dividend" → 0.05}, {"Value", "Greeks"}]
```

```
{10.3649, {Delta → 0.605772, Gamma → 0.0142776, Rho → 19.9237, Theta → -4.93969, Vega → 17.847}}
```

Financial Derivative Speed-Up with CUDA

What Speedup do we Achieve?

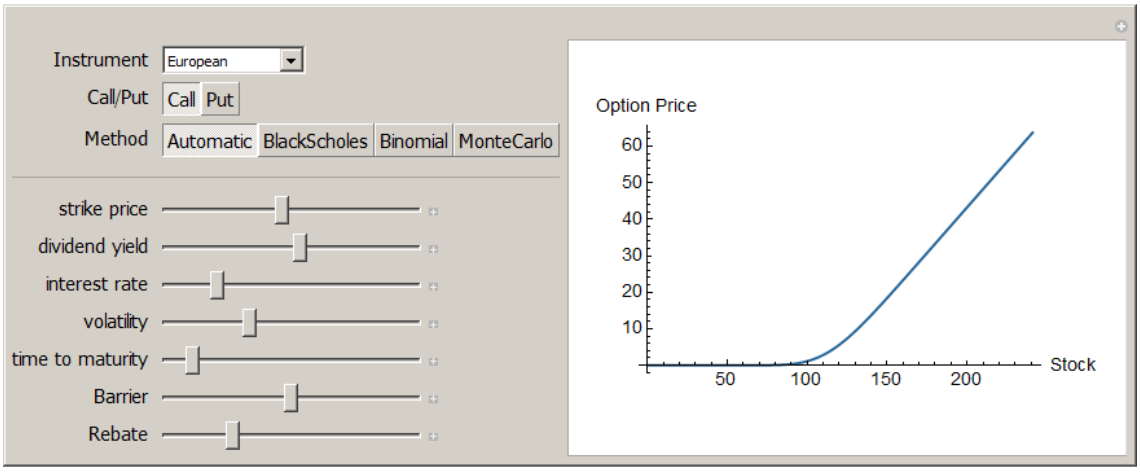


Actual Performance Numbers

Plotting values using FinancialDerivative with variable inputs

Here we can use CUDA estimation to study varying derivative instrument types along with their methods of calculation and graphing the results as stock prices vary.

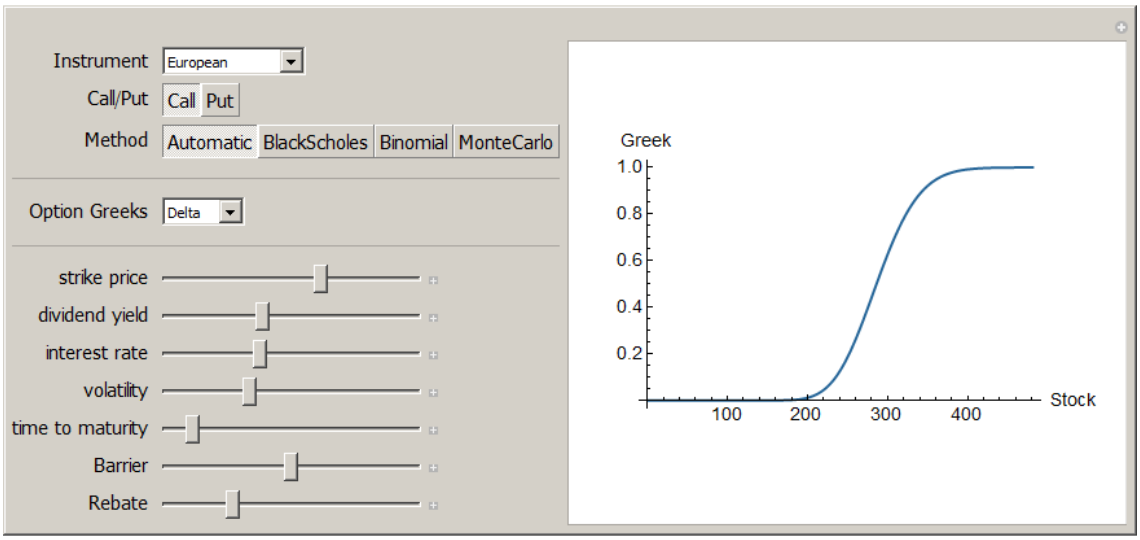
```
Manipulate[ListPlot[
  Quiet[FinancialDerivative[{instrument, type}, {"StrikePrice" → K1, "Expiration" → T, "Barriers" → B}, {"Dividend" → δ,
     "InterestRate" → r, "Volatility" → σ, "CurrentPrice" → Range[40., 160., 0.5], "Rebate" → Reb}, Method → method]],
  Joined → True, AxesLabel -> {"Stock", "Option Price"}, ImageSize → 400, ImagePadding → {{40, 50}, {25, 25}},
  PlotRange → All, PlotStyle → Directive[Thick, wfpblue[1]], BaseStyle → wfpfont],
 {{instrument, "European", "Instrument"}, {"European", "American", "AsianArithmetic", "AsianGeometric",
   "BarrierDownIn", "BarrierDownOut", "BarrierUpIn", "BarrierUpOut", "LookbackFixed", "LookbackFloating"}},
 {{type, "Call", "Call/Put"}, {"Call", "Put"}}, {{method, Automatic, "Method"},
  {Automatic, "BlackScholes", "Binomial", "MonteCarlo"}}, Delimiter, {{K1, 100., "strike price"}, 50., 150.},
 {{δ, .004, "dividend yield"}, .002, .05}, {{r, .05, "interest rate"}, .02, .1},
 {{σ, .3, "volatility"}, .2, .5}, {{T, .1, "time to maturity"}, .001, 1.},
 {{B, 100., "Barrier"}, 50., 150.}, {{Reb, 3., "Rebate"}, 0.5, 10.}]
```



Plotting Greeks using FinancialDerivative with variable inputs

In the same manner as above we can also study the greeks of various derivatives using `CUDAFinancialDerivative`.

```
Manipulate[ListPlot[Quiet[FinancialDerivative[{instrument, type}, {"StrikePrice" → K1, "Expiration" → T, "Barriers" → B},
    {"Dividend" → δ, "InterestRate" → r, "Volatility" → σ, "CurrentPrice" → Range[40., 160., 0.25], "Rebate" → Reb},
    greek, Method → method]], Joined → True, AxesLabel -> {"Stock", "Greek"}, ImageSize → 400,
  ImagePadding → {{40, 50}, {25, 25}}, PlotRange → All, PlotStyle → Directive[Thick, wfpblue[1]], BaseStyle → wfpfont],
 {{instrument, "European", "Instrument"}, {"European", "American", "AsianArithmetic", "AsianGeometric",
   "BarrierDownIn", "BarrierDownOut", "BarrierUpIn", "BarrierUpOut", "LookbackFixed", "LookbackFloating"}},
 {{type, "Call", "Call/Put"}, {"Call", "Put"}}, {{method, Automatic, "Method"},
  {Automatic, "BlackScholes", "Binomial", "MonteCarlo"}}, Delimiter, {{greek, "Delta", "Option Greeks"},
  {"Charm", "Color", "Delta", "Gamma", "Rho", "Speed", "Theta", "Theta", "Vanna", "Vega", "Zomma"}},
 Delimiter, {{K1, 100., "strike price"}, 50., 150.}, {{δ, .004, "dividend yield"}, .002, .05},
 {{r, .05, "interest rate"}, .02, .1}, {{σ, .3, "volatility"}, .2, .5},
 {{T, .1, "time to maturity"}, .001, 1.}, {{B, 100., "Barrier"}, 50., 150.}, {{Reb, 3., "Rebate"}, 0.5, 10.}]
```



Automatic Code Generation in Finance

In the above examples we have either used the built-in `FinancialDerivative` or directly written CUDA code to establish derivative functions. But in this section we show how it is possible to go directly from *Mathematica* code, which is very close to the actual mathemat ical equations, and construct CUDA financial functions. This transformation can be done automatically by using the function `generateCUDAKernel` to create CUDA code which is then wrapped in `ToCCodeString` and acted upon by `CUDAFunctionLoad` to create a `CUDAFunction` object that utilizes the GPU.

The Black–Scholes Equation

```
ClearAll[S, sigma, r, q, T, x, X]
```

$$d_1 = \frac{T\left(r - q + \frac{sigma^2}{2}\right) + Log\left[\frac{S}{X}\right]}{\sqrt{T}\ sigma};$$

$$d_2 = d_1 - sigma\sqrt{T};$$

```
BlackScholes = CDF[NormalDistribution[0, 1], d1] S e^(-q T) - CDF[NormalDistribution[0, 1], d2] X e^(-r T) // FullSimplify
```

$$\frac{1}{2} e^{-(q+r)\,T}\left(e^{r\,T} S\left(1 + Erf\left[\frac{\left(-q + r + \frac{sigma^2}{2}\right) T + Log\left[\frac{S}{X}\right]}{\sqrt{2}\ sigma\sqrt{T}}\right]\right) - e^{q\,T} X\, Erfc\left[\frac{\left(2q - 2r + sigma^2\right) T - 2\,Log\left[\frac{S}{X}\right]}{2\sqrt{2}\ sigma\sqrt{T}}\right]\right)$$

FinancialDerivative computes the Black–Scholes formula for the European call option with the following parameters.

S	⇔	CurrentPrice
X	⇔	StrikePrice
σ	⇔	Volatility
T	⇔	Expiration
r	⇔	InterestRate

Calculating the Greeks

```
greeks = {"Vanilla" → HoldForm[BlackScholes], "Delta" → HoldForm[∂S BlackScholes], "Vega" → HoldForm[∂sigma BlackScholes],
    "Theta" → HoldForm[-∂T BlackScholes], "Rho" → HoldForm[∂r BlackScholes], "Gamma" → HoldForm[∂S,S BlackScholes]};
```

This shows a table of how the Greeks are defined.

Vanilla	BlackScholes
Delta	$\frac{\partial \text{BlackScholes}}{\partial S}$
Vega	$\frac{\partial \text{BlackScholes}}{\partial \text{sigma}}$
Theta	$-\frac{\partial \text{BlackScholes}}{\partial T}$
Rho	$\frac{\partial \text{BlackScholes}}{\partial r}$
Gamma	$\frac{\partial^2 \text{BlackScholes}}{\partial S \partial S}$

Generate Symbolic Code for Greeks

```
Manipulate[ReleaseHold[g /. greeks], {{g, "Delta", "Greek"}, First /@ greeks, ControlType → RadioButton}]
```

Greek ○ Vanilla ◉ Delta ○ Vega ○ Theta ○ Rho ○ Gamma

$$\frac{1}{2} e^{-(q+r)\,T} \left(\frac{e^{r\,T-\frac{\left(\left(-q+r+\frac{\text{sigma}^2}{2}\right)T+\text{Log}\left[\frac{S}{X}\right]\right)^2}{2\,\text{sigma}^2\,T}}}{\text{sigma}\sqrt{T}} \sqrt{\frac{2}{\pi}} - \frac{e^{q\,T-\frac{\left((2\,q-2\,r+\text{sigma}^2)\,T-2\,\text{Log}\left[\frac{S}{X}\right]\right)^2}{8\,\text{sigma}^2\,T}}}{S\,\text{sigma}\sqrt{T}} \sqrt{\frac{2}{\pi}}\,X + e^{r\,T}\left(1+\text{Erf}\left[\frac{\left(-q+r+\frac{\text{sigma}^2}{2}\right)T+\text{Log}\left[\frac{S}{X}\right]}{\sqrt{2}\,\text{sigma}\sqrt{T}}\right]\right) \right)$$

Generate CUDA Code for Greeks

```
Manipulate[ToCCodeString[generateCUDAKernel[g, ReleaseHold[g /. greeks]]],
 {{g, "Delta", "Greek"}, First /@ greeks, ControlType → RadioButton}]
```

Greek ○ Vanilla ○ Delta ◉ Vega ○ Theta ○ Rho ○ Gamma

```
__device__ Real_t Vega_compute(Real_t S, Real_t X, Real_t T, Real_t r, Real_t q, Real_t sigma)
{
return (Real_t) 0.5 * exp(-1 * (q + r) * T) * (2 * exp(q * T + (Real_t) -0.125 * pow(sigma, -2) * pow(T, -1) * pow((2 * q + -2 * r + pow(sigma, 2))
  * T + -2 * log(S * pow(X, -1)), 2)) * pow((Real_t) 3.1415926535897932385, (Real_t) -0.5) * X * (pow(2, (Real_t) -0.5) * pow(T, (Real_t)
  0.5) + (Real_t) -0.5 * pow(2, (Real_t) -0.5) * pow(sigma, -2) * pow(T, (Real_t) -0.5) * ((2 * q + -2 * r + pow(sigma, 2)) * T + -2 * log(S
  * pow(X, -1)))) + 2 * exp(r * T + (Real_t) -0.5 * pow(sigma, -2) * pow(T, -1) * pow((-1 * q + r + (Real_t) 0.5 * pow(sigma, 2)) * T +
  log(S * pow(X, -1)), 2)) * pow((Real_t) 3.1415926535897932385, (Real_t) -0.5) * S * (pow(2, (Real_t) -0.5) * pow(T, (Real_t) 0.5) + -1 *
  pow(2, (Real_t) -0.5) * pow(sigma, -2) * pow(T, (Real_t) -0.5) * ((-1 * q + r + (Real_t) 0.5 * pow(sigma, 2)) * T + log(S * pow(X, -1)))));
}
__global__ void Vega(Real_t* call, Real_t* S, Real_t* X, Real_t* T, Real_t* r, Real_t* q, Real_t* sigma, mint length)
{
int index = threadIdx.x + blockIdx.x * blockDim.x;
if( index < length)
{
call[index] = Vega_compute(S[index], X[index], T[index], r[index], q[index], sigma[index]);
}
}
```

Load Function Using `CUDAFunctionLoad`

```
src = Map[ToCCodeString[generateCUDAKernel[First[#], ReleaseHold[Last[#]]]] &, greeks] // StringJoin;
```

```
greekMethods = First /@ greeks
```

{Vanilla, Delta, Vega, Theta, Rho, Gamma}

```
(europeanOption[#] = CUDAFunctionLoad[src, #, {{_Real}, {_Real, "Input"}, {_Real, "Input"}, {_Real, "Input"}, {_Real, "Input"},
      {_Real, "Input"}, {_Real, "Input"}, _Integer}, 128, "Defines" → {"mint" → "int"}]) & /@ greekMethods
```

```
{CUDAFunction[<>, Vanilla, {{_Real}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, _Integer}],
 CUDAFunction[<>, Delta, {{_Real}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, _Integer}],
 CUDAFunction[<>, Vega, {{_Real}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, _Integer}],
 CUDAFunction[<>, Theta, {{_Real}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, _Integer}],
 CUDAFunction[<>, Rho, {{_Real}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, _Integer}],
 CUDAFunction[<>, Gamma, {{_Real}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, _Integer}]}
```

Compute using `CUDAFunction`

```
optN = 32;
Svec = RandomReal[{20.0, 40.0}, optN];
Xvec = RandomReal[{20.0, 40.0}, optN];
Tvec = RandomReal[{0.1, 10.0}, optN];
Rvec = RandomReal[{0.02, 0.1}, optN];
Qvec = RandomReal[{0.01, 0.06}, optN];
Sigmavec = RandomReal[{0.1, 0.4}, optN];
```

```
call = ConstantArray[0.0, optN];
```

Call one of the functions.

```
europeanOption["Vanilla"][call, Svec, Xvec, Tvec, Rvec, Qvec, Sigmavec, optN]
```

```
{{16.9132, 1.20813, 1.48503, 6.80763, 11.4671, 10.8681, 4.33004, 2.50536, 9.85394, 5.86879, 3.75218, 7.87551, 9.245, 17.8332, 4.32345, 1.22938,
  4.37446, 0.160262, 6.63909, 9.83546, 3.46838, 9.84554, 12.4982, 8.62116, 8.28838, 5.92863, 10.1807, 6.93587, 0.968407, 6.96443, 7.62789, 13.41}}
```

Or use all methods.

```
Grid[{Column[{#} ~Join~ First@europeanOption[#][call, Svec, Xvec, Tvec, Rvec, Qvec, Sigmavec, optN]] & /@ greekMethods},
 Frame -> All]
```

Vanilla	Delta	Vega	Theta	Rho	Gamma
16.9132	0.729666	17.7204	-0.589816	76.3488	0.00473015
1.20813	0.236876	14.543	-0.249571	24.8362	0.0270251
1.48503	0.215647	14.0727	-0.173399	22.1851	0.0162938
6.80763	0.610208	15.8837	-1.65439	25.0051	0.0220693
11.4671	0.71511	10.3906	-0.0957816	71.7124	0.0085364
10.8681	0.724156	12.2657	-0.66156	26.9938	0.0139603
4.33004	0.46297	16.3123	-0.114334	41.7827	0.0177839
2.50536	0.361689	22.9876	-0.415279	50.2922	0.0271937
9.85394	0.632306	5.90244	-0.125193	107.535	0.00461024
5.86879	0.489044	11.8885	-0.0354158	40.4821	0.0097744
3.75218	0.534991	12.2391	-0.278409	20.4235	0.0335277
7.87551	0.461354	20.3353	-0.154808	66.6192	0.00744245
9.245	0.503566	25.3633	0.0697394	76.0661	0.00802566
17.8332	0.773907	10.5919	-0.0225931	56.7279	0.0047061
4.32345	0.454391	23.5809	-0.0722079	45.9714	0.0233263
1.22938	0.299386	7.58969	-1.69848	5.20369	0.0454128
4.37446	0.443692	14.6625	-0.128804	35.3595	0.0156116
0.160262	0.0740174	2.6842	-0.84073	0.87496	0.0269677
6.63909	0.664851	14.5083	0.0681704	115.307	0.0170765
9.83546	0.895938	3.95943	-0.162123	19.1822	0.0140497
3.46838	0.395343	18.3771	0.0731004	39.7724	0.0203994
9.84554	0.742949	17.2147	-0.736283	67.4564	0.016458
12.4982	0.684043	15.1729	-0.242569	56.2039	0.00722092
8.62116	0.716193	14.892	-0.599201	51.0875	0.0134021
8.28838	0.624503	26.5251	-0.432134	85.9143	0.0145926
5.92863	0.501006	17.9471	-0.436411	45.1671	0.0153854
10.1807	0.871774	5.19847	-1.70718	14.8225	0.0163692
6.93587	0.575503	17.8519	-0.72709	38.4493	0.0177487
0.968407	0.236554	12.2791	-0.241238	19.659	0.0349644
6.96443	0.608765	23.1074	-0.321388	64.6511	0.0200453
7.62789	0.556913	20.3026	-0.453401	44.3497	0.0136024
13.41	0.675564	25.2488	-1.22763	58.3069	0.00992842

Define New Options

In the above example we showed how it is possible to use the functions `generateCUDAKernel` and `CUDAFunctionLoad` to create the well-known Black–Scholes derivatives and their Greeks as CUDA functions that utilize the GPU. In this section we will show that the same method can be used to construct a `CUDAFunction` for any derivative that admits a mathematical representation. Here we consider building

the options for cash or nothing calls and puts, asset or nothing calls and puts, future calls and puts, and one touch calls and puts.

```
ClearAll[S, sigma, r, q, T, X, cashOrNothingCall, cashOrNothingPut, assetOrNothingCall, assetOrNothingPut]
```

```
(* Pg 408 in Paul Wilmott on quantitative finance,Volume 2 By Paul Wilmott *)
d1 = (Log[S/X] + (r - q + sigma^2/2) T) / (sigma Sqrt[T]);
d2 = (Log[S/X] + (r - q - sigma^2/2) T) / (sigma Sqrt[T]);
d3 = (Log[S/X] + (r - q + sigma^2/2) T) / (sigma Sqrt[T]);
d4 = (Log[S/X] + (r - q - sigma^2/2) T) / (sigma Sqrt[T]);
d5 = (Log[S/X] - (r + q - sigma^2/2) T) / (sigma Sqrt[T]);
d6 = (Log[S/X] - (r + q + sigma^2/2) T) / (sigma Sqrt[T]);
d7 = (Log[S/X^2] + (r - q - sigma^2/2) T) / (sigma Sqrt[T]);
d8 = (Log[S/X^2] - (r + q + sigma^2/2) T) / (sigma Sqrt[T]);
```

Binary Options

```
binaryOpts = {
    cashOrNothingCall → Exp[-r T] * CDF[NormalDistribution[0, 1], d2],
    cashOrNothingPut → Exp[-r T] * CDF[NormalDistribution[0, 1], - d2],
    assetOrNothingCall → S Exp[-q T] * CDF[NormalDistribution[0, 1], - d1],
    assetOrNothingPut → S Exp[-q T] * CDF[NormalDistribution[0, 1], - d1]
  } // FullSimplify
```

$$\left\{\text{cashOrNothingCall} \to \frac{1}{2} e^{-r\,T}\left(1 + \text{Erf}\left[\frac{\left(-q + r - \frac{\text{sigma}^2}{2}\right) T + \text{Log}\left[\frac{S}{X}\right]}{\sqrt{2}\ \text{sigma}\ \sqrt{T}}\right]\right), \text{cashOrNothingPut} \to \frac{1}{2} e^{-r\,T}\,\text{Erfc}\left[\frac{\left(-q + r - \frac{\text{sigma}^2}{2}\right) T + \text{Log}\left[\frac{S}{X}\right]}{\sqrt{2}\ \text{sigma}\ \sqrt{T}}\right],\right.$$

$$\left.\text{assetOrNothingCall} \to \frac{1}{2} e^{-q\,T}\,S\,\text{Erfc}\left[\frac{\left(-q + r + \frac{\text{sigma}^2}{2}\right) T + \text{Log}\left[\frac{S}{X}\right]}{\sqrt{2}\ \text{sigma}\ \sqrt{T}}\right], \text{assetOrNothingPut} \to \frac{1}{2} e^{-q\,T}\,S\,\text{Erfc}\left[\frac{\left(-q + r + \frac{\text{sigma}^2}{2}\right) T + \text{Log}\left[\frac{S}{X}\right]}{\sqrt{2}\ \text{sigma}\ \sqrt{T}}\right]\right\}$$

Load Function using CUDAFunctionLoad

```
binarysrc = ToCCodeString[
   generateCUDAKernel[ToString[First[#]], Last[#]] & /@ binaryOpts
 ]
```

```
__device__ Real_t cashOrNothingCall_compute(Real_t S, Real_t X, Real_t T, Real_t r, Real_t q, Real_t sigma)
{
return (Real_t) 0.5 * exp(-1 * r * T) * (1 + erf(pow(2, (Real_t) -0.5) * pow(sigma,
  -1) * pow(T, (Real_t) -0.5) * ((-1 * q + r + (Real_t) -0.5 * pow(sigma, 2)) * T + log(S * pow(X, -1)))));
}
__global__ void cashOrNothingCall(Real_t* call, Real_t* S, Real_t* X, Real_t* T, Real_t* r, Real_t* q, Real_t* sigma, mint length)
{
int index = threadIdx.x + blockIdx.x * blockDim.x;
if( index < length)
{
call[index] = cashOrNothingCall_compute(S[index], X[index], T[index], r[index], q[index], sigma[index]);
}
}
__device__ Real_t cashOrNothingPut_compute(Real_t S, Real_t X, Real_t T, Real_t r, Real_t q, Real_t sigma)
{
return (Real_t) 0.5 * exp(-1 * r * T) * erfc(pow(2, (Real_t) -0.5) * pow(sigma,
  -1) * pow(T, (Real_t) -0.5) * ((-1 * q + r + (Real_t) -0.5 * pow(sigma, 2)) * T + log(S * pow(X, -1))));
}
__global__ void cashOrNothingPut(Real_t* call, Real_t* S, Real_t* X, Real_t* T, Real_t* r, Real_t* q, Real_t* sigma, mint length)
{
int index = threadIdx.x + blockIdx.x * blockDim.x;
if( index < length)
{
call[index] = cashOrNothingPut_compute(S[index], X[index], T[index], r[index], q[index], sigma[index]);
}
}
__device__ Real_t assetOrNothingCall_compute(Real_t S, Real_t X, Real_t T, Real_t r, Real_t q, Real_t sigma)
{
return (Real_t) 0.5 * exp(-1 * q * T) * S * erfc(pow(2, (Real_t) -0.5) *
  pow(sigma, -1) * pow(T, (Real_t) -0.5) * ((-1 * q + r + (Real_t) 0.5 * pow(sigma, 2)) * T + log(S * pow(X, -1))));
}
__global__ void assetOrNothingCall(Real_t* call, Real_t* S, Real_t* X, Real_t* T, Real_t* r, Real_t* q, Real_t* sigma, mint length)
{
int index = threadIdx.x + blockIdx.x * blockDim.x;
if( index < length)
{
call[index] = assetOrNothingCall_compute(S[index], X[index], T[index], r[index], q[index], sigma[index]);
}
}
__device__ Real_t assetOrNothingPut_compute(Real_t S, Real_t X, Real_t T, Real_t r, Real_t q, Real_t sigma)
{
return (Real_t) 0.5 * exp(-1 * q * T) * S * erfc(pow(2, (Real_t) -0.5) *
  pow(sigma, -1) * pow(T, (Real_t) -0.5) * ((-1 * q + r + (Real_t) 0.5 * pow(sigma, 2)) * T + log(S * pow(X, -1))));
}
__global__ void assetOrNothingPut(Real_t* call, Real_t* S, Real_t* X, Real_t* T, Real_t* r, Real_t* q, Real_t* sigma, mint length)
{
int index = threadIdx.x + blockIdx.x * blockDim.x;
if( index < length)
{
call[index] = assetOrNothingPut_compute(S[index], X[index], T[index], r[index], q[index], sigma[index]);
}
}
```

```
binaryMethods = ToString[First[#]] & /@ binaryOpts
```

```
{cashOrNothingCall, cashOrNothingPut, assetOrNothingCall, assetOrNothingPut}
```

```
(binaryOption[#] = CUDAFunctionLoad[binarysrc, #, {{_Real}, {_Real, "Input"}, {_Real, "Input"}, {_Real, "Input"},
      {_Real, "Input"}, {_Real, "Input"}, {_Real, "Input"}, _Integer}, 128, "Defines" → {"mint" → "int"}]) & /@ binaryMethods
```

```
{CUDAFunction[<>, cashOrNothingCall, {{_Real}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, _Integer}],
 CUDAFunction[<>, cashOrNothingPut, {{_Real}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, _Integer}],
 CUDAFunction[<>, assetOrNothingCall, {{_Real}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, _Integer}],
 CUDAFunction[<>, assetOrNothingPut, {{_Real}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, _Integer}]}
```

Compute using CUDAFunction

```
call = ConstantArray[0.0, optN];
put = ConstantArray[0.0, optN];
```

Call some of the functions.

```
First@binaryOption["cashOrNothingCall"][call, Svec, Xvec, Tvec, Rvec, Qvec, Sigmavec, optN]
```

```
{0.2191, 0.11057, 0.0745183, 0.417547, 0.567987, 0.578992, 0.249996, 0.171098, 0.391074, 0.208796, 0.412429, 0.184721, 0.309608, 0.559796, 0.36755, 0.187817, 0.220314,
 0.0517234, 0.585971, 0.852562, 0.297838, 0.578256, 0.360769, 0.238086, 0.287646, 0.280894, 0.764139, 0.374492, 0.130079, 0.445342, 0.300947, 0.358738}
```

```
First@binaryOption["assetOrNothingPut"][put, Svec, Xvec, Tvec, Rvec, Qvec, Sigmavec, optN]
```

```
{3.44167, 11.1113, 10.256, 11.1862, 2.73453, 6.71631, 5.80733, 12.1397, 0.892985, 2.92205, 8.29557, 5.31096, 7.25709, 2.57871, 14.3795, 15.7153,
 5.40515, 26.38, 3.06393, 2.00181, 9.08817, 6.21391, 3.74516, 3.69673, 7.31142, 7.31579, 3.26433, 8.89579, 10.4803, 9.15246, 8.33134, 9.91796}
```

Or use all methods.

```
Grid[{Column[{#} ~ Join ~ First@binaryOption[#][call, Svec, Xvec, Tvec, Rvec, Qvec, Sigmavec, optN]] & /@ binaryMethods},
 Frame -> All]
```

cashOrNothingCall	cashOrNothingPut	assetOrNothingCall	assetOrNothingPut
0.2191	0.179313	3.44167	3.44167
0.11057	0.533824	11.1113	11.1113
0.0745183	0.673516	10.256	10.256
0.417547	0.481119	11.1862	11.1862
0.567987	0.11649	2.73453	2.73453
0.578992	0.335201	6.71631	6.71631
0.249996	0.486194	5.80733	5.80733
0.171098	0.4934	12.1397	12.1397
0.391074	0.0359452	0.892985	0.892985
0.208796	0.261825	2.92205	2.92205
0.412429	0.542439	8.29557	8.29557
0.184721	0.262332	5.31096	5.31096
0.309608	0.482519	7.25709	7.25709
0.559796	0.179728	2.57871	2.57871
0.36755	0.542198	14.3795	14.3795
0.187817	0.745482	15.7153	15.7153
0.220314	0.436213	5.40515	5.40515
0.0517234	0.929632	26.38	26.38
0.585971	0.157797	3.06393	3.06393
0.852562	0.104481	2.00181	2.00181
0.297838	0.577287	9.08817	9.08817
0.578256	0.236968	6.21391	6.21391
0.360769	0.287147	3.74516	3.74516
0.238086	0.263528	3.69673	3.69673
0.287646	0.432182	7.31142	7.31142
0.280894	0.346155	7.31579	7.31579
0.764139	0.170005	3.26433	3.26433
0.374492	0.410925	8.89579	8.89579
0.130079	0.529645	10.4803	10.4803
0.445342	0.449317	9.15246	9.15246
0.300947	0.522957	8.33134	8.33134
0.358738	0.458898	9.91796	9.91796

Future Options

```
futuresrc = ToCCodeString[{
   generateCUDAKernel["FutureCall",
    S * Exp[-(r + q) * T] * CDF[NormalDistribution[0, 1], d1] - X * Exp[-r * T] * CDF[NormalDistribution[0, 1], d2]
   ],
   generateCUDAKernel["FuturePut",
    X * Exp[-r * T] * CDF[NormalDistribution[0, 1], -d2] - S * Exp[-(r + q) * T] * CDF[NormalDistribution[0, 1], -d1]
   ]
  }]
```

```
__device__ Real_t FutureCall_compute(Real_t S, Real_t X, Real_t T, Real_t r, Real_t q, Real_t sigma)
{
return (Real_t) -0.5 * exp(-1 * r * T) * X * erfc(-1 * pow(2, (Real_t) -0.5) * pow(sigma, -1) * pow(T, (Real_t) -0.5) * ((-1 *
  q + r + (Real_t) -0.5 * pow(sigma, 2)) * T + log(S * pow(X, -1)))) + (Real_t) 0.5 * exp((-1 * q + -1 * r) * T) * S * erfc(-1 * pow(2,
  (Real_t) -0.5) * pow(sigma, -1) * pow(T, (Real_t) -0.5) * ((-1 * q + r + (Real_t) 0.5 * pow(sigma, 2)) * T + log(S * pow(X, -1))));
}
__global__ void FutureCall(Real_t* call, Real_t* S, Real_t* X, Real_t* T, Real_t* r, Real_t* q, Real_t* sigma, mint length)
{
int index = threadIdx.x + blockIdx.x * blockDim.x;
if( index < length)
{
call[index] = FutureCall_compute(S[index], X[index], T[index], r[index], q[index], sigma[index]);
}
}
__device__ Real_t FuturePut_compute(Real_t S, Real_t X, Real_t T, Real_t r, Real_t q, Real_t sigma)
{
return (Real_t) 0.5 * exp(-1 * r * T) * X * erfc(pow(2, (Real_t) -0.5) * pow(sigma, -1) * pow(T, (Real_t) -0.5) * ((-1 *
  q + r + (Real_t) -0.5 * pow(sigma, 2)) * T + log(S * pow(X, -1)))) + (Real_t) -0.5 * exp((-1 * q + -1 * r) * T) * S * erfc(pow(2,
  (Real_t) -0.5) * pow(sigma, -1) * pow(T, (Real_t) -0.5) * ((-1 * q + r + (Real_t) 0.5 * pow(sigma, 2)) * T + log(S * pow(X, -1))));
}
__global__ void FuturePut(Real_t* call, Real_t* S, Real_t* X, Real_t* T, Real_t* r, Real_t* q, Real_t* sigma, mint length)
{
int index = threadIdx.x + blockIdx.x * blockDim.x;
if( index < length)
{
call[index] = FuturePut_compute(S[index], X[index], T[index], r[index], q[index], sigma[index]);
}
}
```

Load Function using `CUDAFunctionLoad`

```
futureMethods = {"FutureCall", "FuturePut"};

(futureOption[#] = CUDAFunctionLoad[futuresrc, #, {{_Real}, {_Real, "Input"}, {_Real, "Input"}, {_Real, "Input"},
      {_Real, "Input"}, {_Real, "Input"}, {_Real, "Input"}, _Integer}, 128, "Defines" → {"mint" → "int"}]) & /@ futureMethods
```

```
{CUDAFunction[<>, FutureCall, {{_Real}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, _Integer}],
 CUDAFunction[<>, FuturePut, {{_Real}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, _Integer}]}
```

Compute using `CUDAFunction`

```
call = ConstantArray[0.0, optN];
put = ConstantArray[0.0, optN];
```

Call some of the functions.

```
First@futureOption["FutureCall"][call, Svec, Xvec, Tvec, Rvec, Qvec, Sigmavec, optN]
```

```
{1.96232, -0.597339, 0.382075, 4.71997, 2.74825, 8.58191, 1.74334, -0.578995, -2.31667, 0.429812, 3.18325, -0.342644, 5.50109, 10.1752, 2.74556, 0.768687,
 1.10246, 0.120405, 1.48419, 8.61639, 2.12114, 4.54831, 5.23041, 1.24967, 3.51029, 0.306306, 8.35938, 2.89833, -0.678018, 4.87627, 4.73979, 8.49489}
```

Or use all methods.

```
Grid[{Column[{#} ~Join~ First@futureOption[#][put, Svec, Xvec, Tvec, Rvec, Qvec, Sigmavec, optN]] & /@ futureMethods},
 Frame -> All]
```

FutureCall	FuturePut
1.96232	5.12633
-0.597339	11.5194
0.382075	18.4702
4.71997	5.84156
2.74825	1.44379
8.58191	2.99299
1.74334	6.37275
-0.578995	11.2194
-2.31667	0.665298
0.429812	4.14918
3.18325	3.72366
-0.342644	7.54828
5.50109	7.91237
10.1752	1.80664
2.74556	6.33094
0.768687	7.86733
1.10246	6.65163
0.120405	9.65461
1.48419	1.35091
8.61639	0.356707
2.12114	6.23534
4.54831	2.64785
5.23041	4.05551
1.24967	4.97454
3.51029	7.9074
0.306306	6.68433
8.35938	0.838634
2.89833	6.04874
-0.678018	8.84381
4.87627	4.78499
4.73979	8.38024
8.49489	9.21416

One Touch Options

```
onetouchsrc = ToCCodeString[
  generateCUDAKernel["OneTouchCall",
   If[S < X,
    (X / S Exp[-q T])^(2 * r / sigma^2) * CDF[NormalDistribution[0, 1], d5] + (S Exp[-q T] / X) * CDF[NormalDistribution[0, 1], d1],
    1
   ]
  ]
 ]
```

```
__device__ Real_t OneTouchCall_compute(Real_t S, Real_t X, Real_t T, Real_t r, Real_t q, Real_t sigma)
{
return S < X ? (Real_t) 0.5 * pow(exp(-1 * q * T) * pow(S, -1) * X, 2 * r * pow(sigma, -2)) * erfc(-1 * pow(2, (Real_t) -0.5) * pow(sigma, -1) * pow(T, (Real_t)
  -0.5) * (-1 * (q + r + (Real_t) -0.5 * pow(sigma, 2)) * T + log(S * pow(X, -1)))) + (Real_t) 0.5 * exp(-1 * q * T) * S * pow(X, -1) * erfc(-1 *
  pow(2, (Real_t) -0.5) * pow(sigma, -1) * pow(T, (Real_t) -0.5) * ((-1 * q + r + (Real_t) 0.5 * pow(sigma, 2)) * T + log(S * pow(X, -1)))) : 1;
}
__global__ void OneTouchCall(Real_t* call, Real_t* S, Real_t* X, Real_t* T, Real_t* r, Real_t* q, Real_t* sigma, mint length)
{
int index = threadIdx.x + blockIdx.x * blockDim.x;
if( index < length)
{
call[index] = OneTouchCall_compute(S[index], X[index], T[index], r[index], q[index], sigma[index]);
}
}
```

Load function using `CUDAFunctionLoad`

```
oneTouchOption = CUDAFunctionLoad[onetouchsrc, "OneTouchCall",
  {{_Real}, {_Real, "Input"}, {_Real, "Input"}, {_Real, "Input"}, {_Real, "Input"}, {_Real, "Input"},
   {_Real, "Input"}, _Integer}, 128, "Defines" → {"mint" → "int"}, "ShellOutputFunction" -> Print]
```

```
CUDAFunction[<>, OneTouchCall, {{_Real}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, {_Real, Input}, _Integer}]
```

```
C:\Users\abduld\AppData\Roaming\Mathematica\ApplicationData\CUDALink\
  BuildFolder\abduldwinlap-1328\Working-abduldwinlap-1328-4856-6>call
  "c:\Program Files (x86)\Microsoft Visual
  Studio 10.0\VC\vcvarsall.bat" amd64
Setting environment for using Microsoft Visual Studio 2010 x64 tools.
CUDAFunction-401.cu
CUDAFunction-401.cu
tmpxft_00001e74_00000000-3_CUDAFunction-401.cudafe1.gpu
tmpxft_00001e74_00000000-10_CUDAFunction-401.cudafe2.gpu
```

Compute using `CUDAFunction`

```
call = ConstantArray[0.0, optN];
```

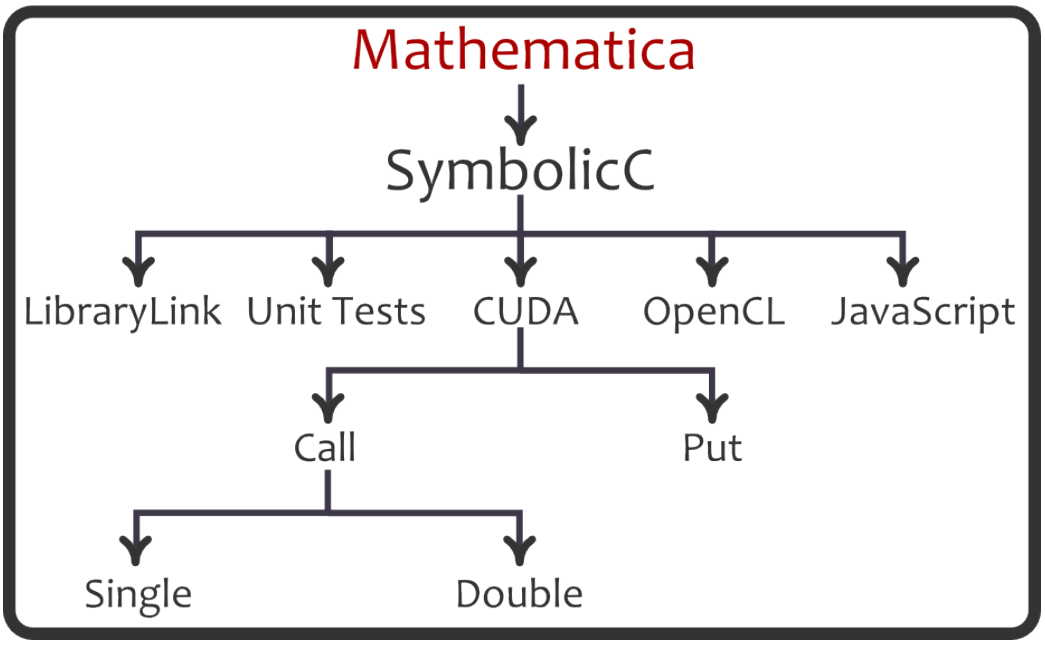
Call the function.

```
oneTouchOption[call, Svec, Xvec, Tvec, Rvec, Qvec, Sigmavec, optN]
```

```
{{0.90806, 0.165213, 0.248219, 1., 1., 1., 0.675294, 0.309112, 1., 0.567439, 1., 0.445925,
  1., 1., 1., 0.471485, 0.565668, 0.123636, 1., 1., 1., 1., 1., 0.871803, 0.932245, 0.530269, 1., 1., 0.168201, 1., 1., 1.}}
```

Code Generation

Used extensively in our development process. It allows us to write code once and transform into to multiple languages as we need them.



We write code like this

```
addOption["EuropeanVanilla",
 {
  F → "FuturePrice",
  X → "StrikePrice",
  V → "Volatility",
  R → "InterestRate",
  T → "ExpiryDate"
 }, {
   ToSymbolicC[{
  If[T < 0 || S < 0 || X <= 0 || V <= 0 || R < 0 || Q < 0,
   Return[
    Call -> Limits[Real, "NAN"],
    Put -> Limits[Real, "NAN"]
    ]
   ];
  CDeclare[{cnd1, cnd2, expRT, expQT}];
  If[S == 0, Return[Call -> 0, Put -> X*expRT]];
  If[T == 0, Return[Call -> Max[S - X, 0], Put -> Max[X - S, 0]]];
  Return[
   Call -> S*expQT*cnd1 - X*expRT*cnd2,
   Put -> X*expRT*(1 - cnd2) - S*expQT*(1 - cnd1)
   ]
  }]
 }, ...
]
```

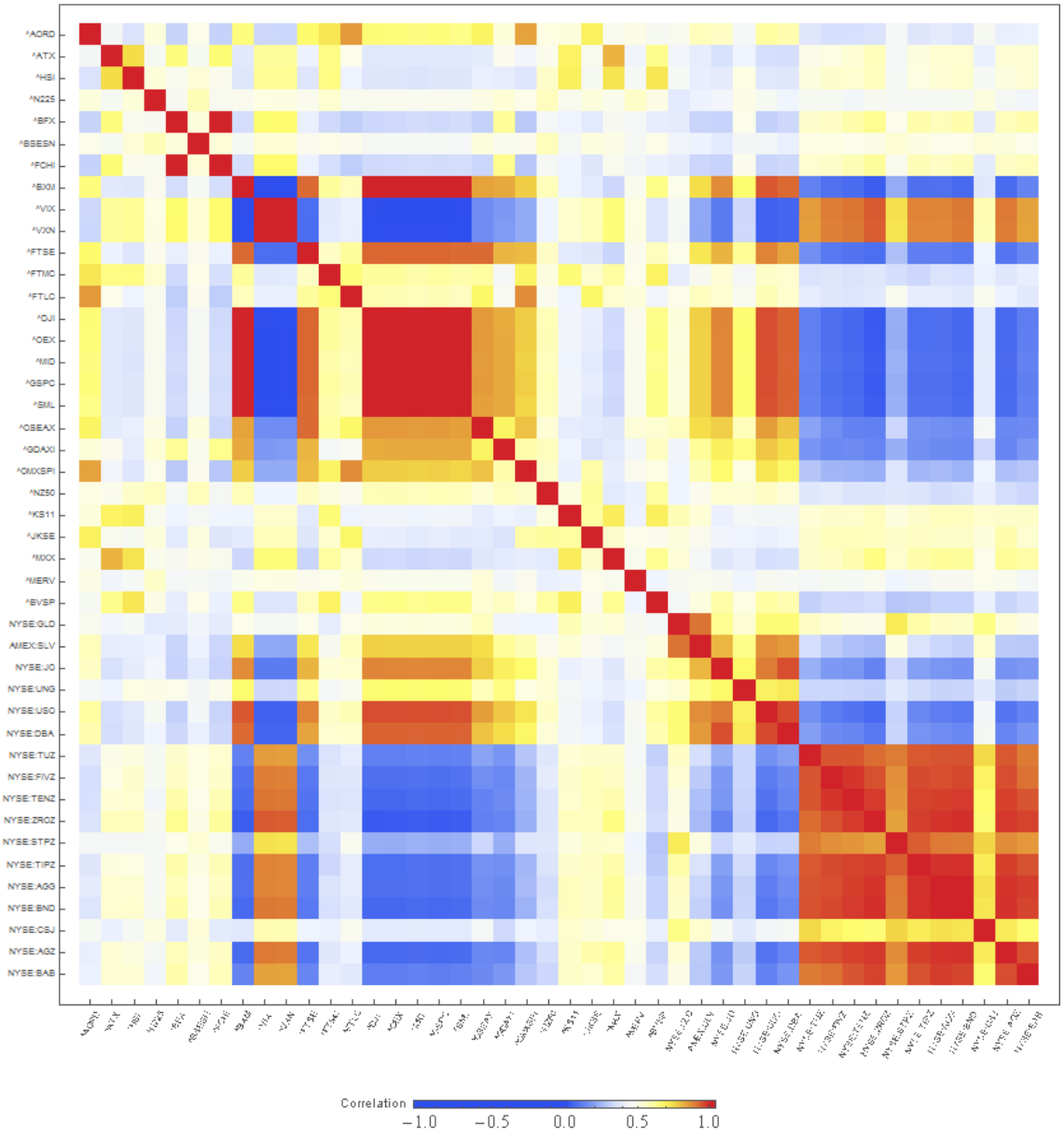
And this gets generated for CUDA

```
static double iEuropeanVanilla_callValue (double S, double X, double V, double R, double Q, double T)
{
    if (T < 0 || S < 0 || X <= 0 || V <= 0 || R < 0 || Q < 0)
    {
        return NAN;
    }
    double d1, d2;
    d1 = (T * ((R + 0.5 * (V * V)) - Q) + log (S / X)) / (sqrt (T) * V);
    d2 = (T * ((R + 0.5 * (V * V)) - Q) + log (S / X)) / (sqrt (T) * V) - V * sqrt (T);
    double cnd1 = 0.5 * erfc (-0.707106781186 * d1);
    double cnd2 = 0.5 * erfc (-0.707106781186547 * d2);
    double expRT = exp (-(R * T));
    double expQT = exp (-(Q * T));
    if (S == 0)
    {
        return 0;
    }
    if (T == 0)
    {
        return max (S - X, 0);
    }
    return S * expQT * cnd1 - X * expRT * cnd2;
}
```

Strategy Development - Using Graph Theory to Understand Market Relationships

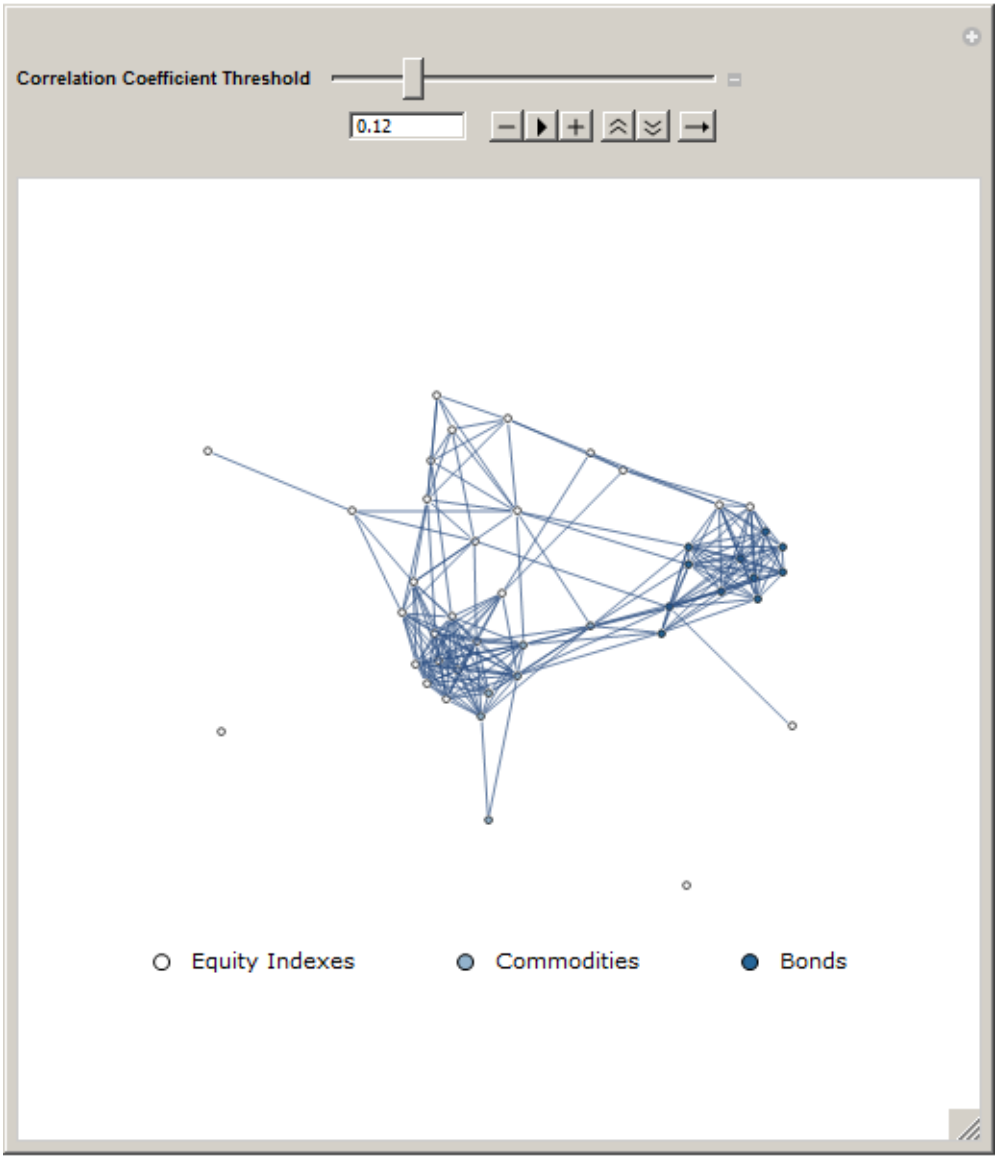
Traditional Matrix View of Correlations Between Asset Classes

```
Labeled[ArrayPlot[Correlation[portcor], ColorFunction → "TemperatureMap",
  FrameTicks → {Transpose[{Range[Length[bigport]], Style[#, wfpfont, 7] & /@ bigport}],
    Transpose[{Range[Length[bigport]], Rotate[#, Pi / 3] & /@ Map[Style[#, wfpfont, 7] &, bigport]}]},
  FrameStyle → Darker@Gray, AspectRatio → 1, ImageSize → 350], Graphics[
  {Thickness[.5], Line[Table[{i, 0}, {i, -1, 1, 0.05}]], VertexColors → Table[ColorData["TemperatureMap", n], {n, -1, 1, 0.05}]]},
  AspectRatio → 1 / 40, PlotRangeClipping → True, Frame → True, FrameStyle → Darker@Gray,
  FrameTicks → {{None, None}, {Automatic, None}}, FrameLabel → {{Style["Correlation", wfpfont, 10], None}, {None, None}},
  RotateLabel → False, ImageSize → 300, ImageMargins → {{0, 0}, {0, 20}}]]
```



Apply Graph Theory to Finance

```
Manipulate[Labeled[AdjacencyGraph[Table[Tooltip[n, bigportnames[[n]]], {n, 1, Length[bigportnames], 1}], portm[θ], graphsty],
  graphlabels], {{θ, .36, Style["Correlation Coefficient Threshold", Bold, 9, wfpfont]}, 0, .6, .01, Appearance → "Open"},
 SaveDefinitions → True, ContentSize → {500, 500}, Alignment → Center]
```

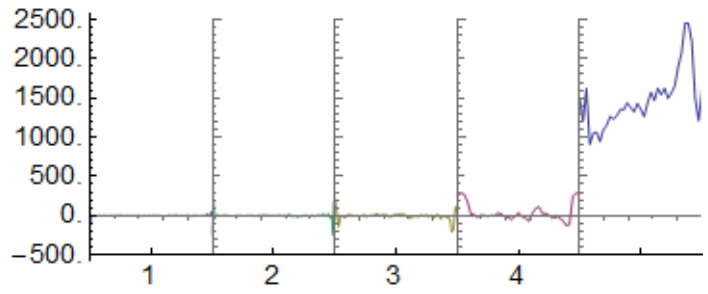


Wavelet Analysis

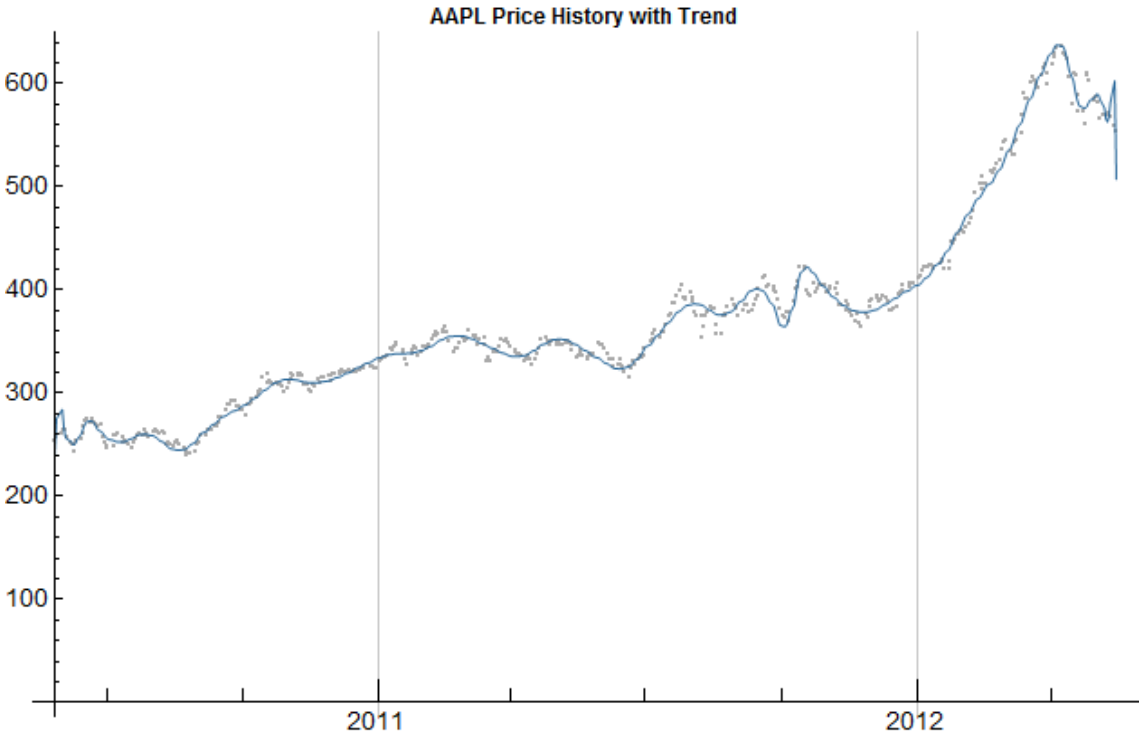
```
DateListPlot[priceUS[[1]], Joined → True, Frame → False, BaseStyle → wfpfont, PlotStyle → wfpblue[1], PlotLabel → nameUS[[1]]]
```


```
dwd = DiscreteWaveletTransform[priceUS[[1, All, 2]], BiorthogonalSplineWavelet[3, 3], 4];
```

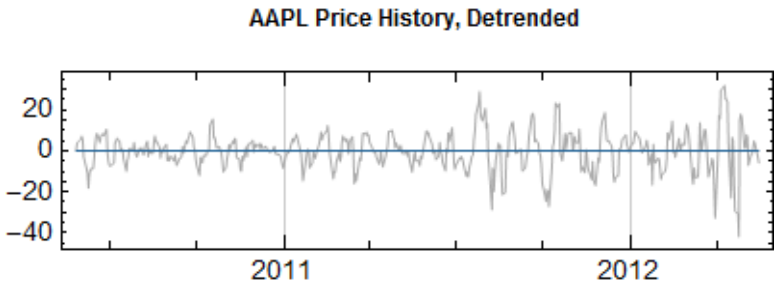
```
WaveletListPlot[dwd, PlotLayout → "CommonYAxis", BaseStyle → wfpfont]
```



```
tr = InverseWaveletTransform[WaveletThreshold[dwd, {"Hard", 100}]];
trendplot = DateListPlot[Transpose[{priceUS[[1, All, 1]], tr}],
   Joined → True, Frame → False, PlotStyle → wfpblue[1], BaseStyle → wfpfont];
AAPLPriceTrendPlot = Show[DateListPlot[priceUS[[1]], Frame → False, PlotStyle → {Lighter@Gray, PointSize[0.003]},
   BaseStyle → wfpfont, PlotLabel → Style["AAPL Price History with Trend", Bold], ImageSize → 400], trendplot]
```




```
dwddt = DiscreteWaveletTransform[priceUS[[1, All, 2]], SymletWavelet[2], 4, Padding → "Extrapolated"];
WaveletListPlot[dwddt, PlotLayout → "CommonYAxis", BaseStyle → wfpfont];
dtr = InverseWaveletTransform[WaveletMapIndexed[# 0.0 &, dwddt, {___, 0}]];
AAPLDetrendedPlot =
 Show[DateListPlot[Transpose[{priceUS[[1, All, 1]], dtr}], Joined → True, AspectRatio → 1 / 4, BaseStyle → wfpfont,
   PlotStyle → Lighter@Gray, ImageSize → 400, PlotLabel → Style["AAPL Price History, Detrended", Bold], PlotRange → All],
  DateListPlot[{{priceUS[[1, All, 1]][[1]], Mean[dtr]}, {priceUS[[1, All, 1]][[-1]], Mean[dtr]}},
   Joined → True, PlotStyle → wfpblue[1]]]
```



Sentiment Analysis

```
CreateDocument@Import["http://feeds.reuters.com/reuters/businessNews", "rss"]
```

NotebookObject [Reuters: Business News–rss]

```
news = Import["http://feeds.reuters.com/reuters/businessNews"];
result = StringCases[Cases[news, _String, Infinity], ___ ~~ ToString[Euro] ~~ ___, IgnoreCase → True];
Manipulate[DynamicModule[{news = Import["http://feeds.reuters.com/reuters/businessNews"],
    result = StringCases[Cases[news, _String, Infinity], ___ ~~ ToString[input] ~~ ___, IgnoreCase → True]},
  Style[Column[Flatten[result], Dividers → {False, All}, Spacings → {Automatic, 2}], wfpfont]],
 {{input, Euro, Style["Search Field", Bold, 15, wfpfont]}, gain}, ControlType → InputField, SaveDefinitions → True]
```

What is Wolfram Finance Platform?

Make financial computation accessible for the masses

Make financial computation run fast using either multicore CPUs or multiple GPUs

Make financial computation coherent and correct using an all-in-one solution and arbitrary precision computation

Recommended Next Steps

Try *Mathematica* for free:
http://www.wolfram.com/mathematica/trial

Schedule a technical demo:
www.wolfram.com/mathematica-demo

Learn more about CUDA programming in *Mathematica*:

U.S. and Canada:
1-800-WOLFRAM (965-3726)
info@wolfram.com

Europe:
+44-(0)1993-883400
info@wolfram.co.uk

Outside U.S. and Canada (except Europe and Asia):
+1-217-398-0700

Asia:
+81-(0)3-3518-2880

info@wolfram.com info@wolfram.co.jp

Q&A

Whitepapers are also available in the front about *Mathematica*'s GPU support

Initialization

Color

```
wfpfont = FontFamily → "Helvetica";
wfpblue[o_] := RGBColor[{.15, .4, .6, o}];
```

Load packages

```
Needs["CUDALink`"]

LaunchKernels[];
```

Code

```
CUDAParallelFoldList[data_, stride_, n_] := Module[{src, kernelName, parallelFold},
   If[! Divisible[stride, 64], Return[$Failed]];

   src = "
__global__ void parallelFold(Real_t *data, unsigned int nData, unsigned int stride)
{
    Real_t runningTotal = static_cast<Real_t>(0.0);
    for (int ii=0; ii < nData; ii += stride)
    {
        runningTotal += data[ii + blockIdx.x*blockDim.x + threadIdx.x];
        data[ii + blockIdx.x*blockDim.x + threadIdx.x] = runningTotal;
    }
}
";
   kernelName = "parallelFold";

   parallelFold = CUDAFunctionLoad[src, kernelName, {{_Real, _, "InputOutput"}, _Integer, _Integer}, 64];

   parallelFold[data, n, stride, stride];

   data
  ];

srcf = FileNameJoin[{$CUDALinkPath, "SupportFiles", "random.cu"}];

CUDAInverseCND = CUDAFunctionLoad[{srcf}, "InverseCND", {{_Real, _, "InputOutput"}, _Integer, _Integer}, 256];
```

```
CUDAMemoryBrownianMotion[period_Real, steps_Integer: 1000, init_: 0.] :=
  Module[{res, mems}, mems = CUDAMemoryLoad[RandomVariate[NormalDistribution[0, Sqrt[period / steps]], steps]];
   res = CUDAMemoryGet[CUDAFoldList[Plus, init, mems]];
   CUDAMemoryUnload[mems];
   res];

CUDABrownianMotionPaths3[period_Real, steps_Integer: 1000, paths_Integer] :=
  Module[{res, mems}, res = Map[CUDAMemoryGet[CUDAFoldList[Plus, 0., mems = CUDAMemoryLoad[
           RandomVariate[NormalDistribution[0, Sqrt[period / steps]], steps]]] &, Range[paths]]; CUDAMemoryUnload[mems];
   res];

CUDABrownianMotionPaths4[period_Real, steps_Integer: 1000, paths_Integer] :=
  Module[{mems, res}, mems = Map[CUDAMemoryLoad, RandomVariate[NormalDistribution[0, Sqrt[period / steps]], {paths, steps}]];
   res = Map[CUDAMemoryGet, ParallelMap[CUDAFoldList[Plus, 0., #] &, mems]];
   CUDAMemoryUnload /@ mems;
   res];

CUDABrownianMotionPaths6[period_Real, steps_Integer: 1000, paths_Integer] := Map[Subtract[#, First[#]] &,
   Partition[CUDAFoldList[Plus, 0., RandomVariate[NormalDistribution[0, Sqrt[period / steps]], steps * paths]], steps]];

CUDABrownianMotionPaths[period_Real, steps_Integer, paths_Integer] :=
  Module[{nSamples, dOutput, hOutput}, If[! Divisible[paths, 64], Return[$Failed]];
   nSamples = steps * paths;
   SeedRandom[1234, Method → "MKL"];
   dOutput = CUDAMemoryLoad[RandomVariate[NormalDistribution[0, Sqrt[period / steps]], nSamples]];
   CUDAParallelFoldList[dOutput, paths, nSamples];
   hOutput = CUDAMemoryGet[dOutput];
   CUDAMemoryUnload[dOutput];
   Transpose[Partition[hOutput, paths]]];

CUDABrownianMotionPaths1[period_Real, steps_Integer, paths_Integer] :=
  Module[{nSamples, dOutput, hOutput}, If[! Divisible[paths, 64], Return[$Failed]];
   nSamples = steps * paths;
   dOutput = CUDAMemoryLoad[RandomReal[period, nSamples]];
   CUDAInverseCND[dOutput, nSamples, 1];
   CUDAParallelFoldList[dOutput, paths, nSamples];
   hOutput = CUDAMemoryGet[dOutput];
   CUDAMemoryUnload[dOutput];
   Transpose[Partition[hOutput, paths]] * Sqrt[period / steps]];
```

Define a minimal *Mathematica*-to-C translator:

```
Needs["SymbolicC`"]

SetAttributes[ToSymbolicC, HoldAll]
ToSymbolicC[(op : (Plus | Subtract | Times | Divide))[args___]] := COperator[op, ToSymbolicC /@ {args}]
ToSymbolicC[If[cond_, truestmt_, falsestmt_]] :=
 CConditional[ToSymbolicC[cond], ToSymbolicC[truestmt], ToSymbolicC[falsestmt]]
ToSymbolicC[Power[E, args___]] := CCall["exp", ToSymbolicC /@ {args}]
ToSymbolicC[(op : (Power | Log | Log10 | Exp | Sqrt))[args___]] := CStandardMathOperator[op, ToSymbolicC /@ {args}]
ToSymbolicC[(op : (Erf | Erfc | Max | Min))[args___]] := CCall[ToLowerCase[SymbolName[op]], ToSymbolicC /@ {args}]
ToSymbolicC[(op : (Less | Greater | Equal | Unequal | LessEqual | GreaterEqual))[args___]] :=
 COperator[op, ToSymbolicC /@ {args}]
ToSymbolicC[(op : (CauchyDistribution | ChiSquareDistribution | ExponentialDistribution |
        LaplaceDistribution | LogNormalDistribution | NormalDistribution | WeibullDistribution))[args___]] :=
 CCall[ToLowerCase[SymbolName[op]], {"rngState", Sequence[ToSymbolicC /@ {args}]}]
ToSymbolicC[(op : UniformDistribution)[args___]] := CCall[ToLowerCase[SymbolName[op]], {"rngState", ToSymbolicC @@ args}]
ToSymbolicC[(op : TransformedDistribution)[args___]] :=
 ToSymbolicC[Release[First[{args}] /. ((First[#] → Last[#]) & /@ Last[{args}])]]
ToSymbolicC[(op : (BrownianMotion | JumpDiffusionProcess))[args1___][args2___]] :=
 CCall[CMember[ToLowerCase[SymbolName[op]], "step"], {"rngState", Sequence[ToSymbolicC /@ {args1, args2}]}]
ToSymbolicC[(op : Part)[args___]] := CArray[First@{args}, ToSymbolicC[FullSimplify[Last@{args} - 1]]]
ToSymbolicC[x : (_Real | _Rational)] := CCast["Real_t", N[x]]
ToSymbolicC[x : (E | Pi)] := CCast["Real_t", N[x, 20]]
ToSymbolicC[args_List] := ToSymbolicC /@ args
ToSymbolicC[args__] := args
```

Define a minimal CUDA kernel code generator.

```
generateCUDAKernel[name_, f_] :=
 {
  CFunction[{"__device__", "Real_t"}, name <> "_compute",
   {{"Real_t", "S"}, {"Real_t", "X"}, {"Real_t", "T"}, {"Real_t", "r"}, {"Real_t", "q"}, {"Real_t", "sigma"}},
   CReturn[ToSymbolicC[f]]
  ],
  SymbolicCUDAFunction[name, {{CPointerType["Real_t"], "call"}, {CPointerType["Real_t"], "S"},
    {CPointerType["Real_t"], "X"}, {CPointerType["Real_t"], "T"}, {CPointerType["Real_t"], "r"},
    {CPointerType["Real_t"], "q"}, {CPointerType["Real_t"], "sigma"}, {"mint", "length"}}, CBlock[{
     SymbolicCUDADeclareIndexBlock[1],
     CIf[COperator[Less, {"index", "length"}],
      CAssign[CArray["call", "index"], CCall[name <> "_compute", CArray[#, "index"] & /@ {"S", "X", "T", "r", "q", "sigma"}]]
     ]
    }]
  ]
 }

Needs["GPUTools`SymbolicGPU`"]

generateGPUKernel[name_, f_] :=
 {
  CFunction[{"__device__", "Real_t"}, name <> "_compute",
   {{"Real_t", "S"}, {"Real_t", "X"}, {"Real_t", "T"}, {"Real_t", "r"}, {"Real_t", "q"}, {"Real_t", "sigma"}},
   CReturn[GPUExpression[f]]
  ],
  SymbolicCUDAFunction[name, {{CPointerType["Real_t"], "call"}, {CPointerType["Real_t"], "S"},
    {CPointerType["Real_t"], "X"}, {CPointerType["Real_t"], "T"}, {CPointerType["Real_t"], "r"},
    {CPointerType["Real_t"], "q"}, {CPointerType["Real_t"], "sigma"}, {"mint", "length"}}, CBlock[{
     SymbolicCUDADeclareIndexBlock[1],
     CIf[COperator[Less, {"index", "length"}],
      CAssign[CArray["call", "index"], CCall[name <> "_compute", CArray[#, "index"] & /@ {"S", "X", "T", "r", "q", "sigma"}]]
     ]
    }]
  ]
 }

generateFoldFunction[name_, fun_, vars_] :=
 {
  CFunction[{"__device__", "Real_t"}, name <> "_fold", {"Real_t", SymbolName[#]} & /@ vars,
   CBlock[{
     CReturn[ToSymbolicC[fun]]
    }]
  ]
 }

generateEulerMethod[drift_, diff_, solveFor_, wrt_, buffer_] :=
 {
  Table[
   CAssign[CArray[buffer, ii - 1], COperator[Plus, {
      COperator[Times, {"d" <> SymbolName[First@wrt], ToSymbolicC[Evaluate[drift[[ii]]]]}],
      COperator[Times,
       {CCall["sqrt", {"d" <> SymbolName[First@wrt]}], CArray[buffer, ii - 1], ToSymbolicC[Evaluate[diff[[ii]]]]}]
     }]],
   {ii, 1, Length@drift}
  ],
  Table[
   CAssign[ToSymbolicC[(First@solveFor)[[ii]]],
    COperator[Plus, {ToSymbolicC[(First@solveFor)[[ii]]], CArray[buffer, ii - 1]}]],
   {ii, 1, Length@drift}
  ],
  CAssign[SymbolName[First@wrt], COperator[Plus, {SymbolName[First@wrt], "d" <> SymbolName[First@wrt]}]]
 }
```

```
generateSimulationFunction[name_, drift_, diff_, valueFun_, solveFor_, wrt_, vars_, nSteps_, method_] :=
 CFunction[{"__device__", "void"}, name <> "_simulate",
  Join[{"Real_t", SymbolName[#]} & /@ vars, {{"Real_t", "_start"}, {"Real_t", "_end"}, {"curandState*", "rngState"},
    {"Real_t&", "foldValue"}, {"Real_t*", "corrMat"}, {"Real_t*", First@solveFor}, {"Real_t*", "_buffer"}}],
  CBlock[{
    Table[
     CAssign[ToSymbolicC[(First@solveFor)[[ii]]], ToSymbolicC[Evaluate[(Last@solveFor)[[ii]]]]],
     {ii, 1, Length@drift}
    ],
    CDeclare["mint", "ii"],
    CDeclare["Real_t", SymbolName[First@wrt]],
    CAssign[SymbolName[First@wrt], "_start"],
    CDeclare["Real_t", "d" <> SymbolName[First@wrt]],
    CAssign["d" <> SymbolName[First@wrt], COperator[Minus, {"_end", "_start"}]],
    CAssign["d" <> SymbolName[First@wrt], COperator[Divide, {"d" <> SymbolName[First@wrt], nSteps}]],
    CAssign["foldValue", 0],
    CFor[CAssign["ii", 0], COperator[Less, {"ii", nSteps}], COperator[Increment, {"ii"}],
     CBlock[{
       CCall["generateCenteredMultiNormal", {"rngState", "corrMat", Length@drift, "_buffer"}],
       Switch[method,
        "Euler",
        generateEulerMethod[drift, diff, solveFor, wrt, "_buffer"],
        _,
        None
       ],
       CAssign["foldValue", CCall[name <> "_fold", Join[{"foldValue", ToSymbolicC[valueFun]}, vars]]]
      }]
    ]
   }]
 ]

generateSamplingFunction[name_, wrt_, vars_, dim_, blockSize_] :=
 CFunction[{"extern \"C\"", "__global__", "void"}, name <> "_sampler",
  Join[{"Real_t", #} & /@ vars, {{"Real_t*", "gCorrMat"}, {"mint", "seed"}, {"mint", "numSamples"}, {"Real_t*", "res"}}],
  CBlock[{
    CDeclare["mint", "index"],
    CAssign["index",
     COperator[Plus, {CMember["threadIdx", "x"], COperator[Times, {CMember["blockIdx", "x"], CMember["blockDim", "x"]}]}]],
    CDeclare["curandState", "rngState"],
    CCall["curand_init", {CCast["unsigned long long", "seed"], "index", 0, CAddress["rngState"]}],
    CDeclare["mint", "stride"],
    CAssign["stride", COperator[Times, {"gridDim.x", "blockDim.x"}]],
    CDeclare["Real_t", "_start"],
    CDeclare["Real_t", "_end"],
    CAssign["_start", ToSymbolicC[Evaluate[wrt[[2]]]]],
    CAssign["_end", ToSymbolicC[Evaluate[wrt[[3]]]]],
    CDeclare["Real_t", "foldv"],
    CDeclare["__shared__ Real_t", CArray["corrMat", dim*dim]],
    CIf[COperator[Less, {"threadIdx.x", dim*dim}],
     CAssign[CArray["corrMat", "threadIdx.x"], CArray["gCorrMat", "threadIdx.x"]]
    ],
    CDeclare["__shared__ Real_t", CArray["_buffer1", blockSize*dim]],
    CDeclare["__shared__ Real_t", CArray["_buffer2", blockSize*dim]],
    CDeclare["mint", "ii"],
    CFor[CAssign["ii", "index"], COperator[Less, {"ii", "numSamples"}], CAssign["ii", COperator[Plus, {"ii", "stride"}]],
     CBlock[{
       CCall[name <> "_simulate", Join[vars, {"_start", "_end", CAddress["rngState"], "foldv",
           "corrMat", CAddress[CArray["_buffer1", COperator[Times, {dim, CMember["threadIdx", "x"]}]]],
           CAddress[CArray["_buffer2", COperator[Times, {dim, CMember["threadIdx", "x"]}]]]}]],
       CAssign[CArray["res", "ii"], "foldv"]
      }]
    ]
   }]
 ]
```

```
generateStochasticIntegralKernel[name_, drift_, diffusion_,
  valueFun_, solveFor_, wrt_, vars_, fold_, nSteps_, method_, blockSize_] :=
 Module[{fixedFold, foldCode, simCode, samplingKernel, res},
  fixedFold = (First@fold) /. {#1 → a, #2 → b};
  foldCode = generateFoldFunction[name, fixedFold, Join[{a, b}, vars]];
  simCode = generateSimulationFunction[name, drift, diffusion, valueFun, solveFor, wrt, vars, nSteps, method];
  samplingKernel = generateSamplingFunction[name, wrt, vars, Length@drift, blockSize];
  res = {
    CInclude["CUDARandomVariate.cu"],
    foldCode,
    simCode,
    samplingKernel
   };
  res
 ]

generateStochasticIntegralFunction[name_, drift_, diffusion_, valueFun_, solveFor_, wrt_, vars_, fold_, nSteps_, method_] :=
 Module[{code, cudaFun, res},
  code = ToCCodeString[generateStochasticIntegralKernel[name, drift,
      diffusion, valueFun, solveFor, wrt, vars, fold, nSteps, method, 64], "Indent" → Automatic];
  cudaFun = CUDAFunctionLoad[code, name <> "_sampler", Join[_Real & /@ vars,
      {{_Real, _, "Input"}, _Integer, _Integer, {_Real, _, "Output"}}], 64,
    "ShellOutputFunction" → Print, "IncludeDirectories" → {NotebookDirectory[]}, "UnmangleCode" → False];
  res = {cudaFun, SymbolName /@ vars}
 ]

CUDAStochasticIntegrate[sde_, inputs_, corrMat_, nSamples_: 16 384, seed_: Automatic] :=
 Module[{cholCorrMat, realSeed, dCorr, dRes, hRes},
  cholCorrMat = CholeskyDecomposition[corrMat];
  realSeed = If[seed === Automatic, RandomInteger[{-2 147 483 647, 2 147 483 647}], seed];
  dCorr = CUDAMemoryLoad[Flatten[cholCorrMat]];
  dRes = CUDAMemoryAllocate[Real, nSamples];
  (First@sde)[Sequence @@ inputs, dCorr, realSeed, nSamples, dRes];
  hRes = CUDAMemoryGet[dRes];
  CUDAMemoryUnload /@ {dCorr, dRes};
  hRes
 ]
```

WFP Style Sheet

Mouseoverlay

VaR Plot Colors

VaR Plotting Styles

VaR Stock Prices, Names and Symbols

```
stocksUS = {"AAPL", "GOOG", "CTSH", "GRMN", "AMZN", "DLB", "RIMM", "COH", "INFY", "DECK", "BIDU", "SYK", "NVDA", "EBIX", "DV",
    "FCX", "AFAM", "GILD", "FSLR", "MIDD", "ORCL", "QSII", "MSFT", "NFLX", "WDC", "LULU", "PG", "ARO", "AOB", "GE"};
stocksF = {"F:UTDI", "TSX:RIM", "F:SWV", "F:NDA", "F:SZG", "F:SDF", "F:MAN", "F:DBK", "F:DAI", "ASX:WOW", "F:QCE",
    "F:VIA", "F:BMW", "F:ALV", "F:SAP", "F:SIE", "F:WDI", "F:CBK", "F:IFX", "ASX:BHP", "F:LHA", "F:ADS", "F:EOAN"};
stocksI = {"^AORD", "^ATX", "^HSI", "^N225", "^BFX", "^BSESN", "^FCHI", "^BXM", "^VIX", "^VXN",
    "FTSE100", "FTSE250", "FTSE350", "^DJI", "SP100", "SP400", "SP500", "SP600",
    "^OSEAX", "^GDAXI", "^OMXSPI", "^NZ50", "^KS11", "^JKSE", "^MXX", "^MERV", "^BVSP"};
commodities = {"GLD", "SLV", "JO", "UNG", "USO", "DBA"};
bonds = {"TUZ", "FIVZ", "TENZ", "ZROZ", "STPZ", "TIPZ", "AGG", "BND", "CSJ", "AGZ", "BAB"};
```

```
retUS = FinancialData[#, "Return", DatePlus[-720]] & /@ stocksUS;
retF = FinancialData[#, "Return", DatePlus[-720]] & /@ stocksF;
retI = FinancialData[#, "Return", DatePlus[-720]] & /@ stocksI;
retC = FinancialData[#, "Return", DatePlus[-720]] & /@ commodities;
retB = FinancialData[#, "Return", DatePlus[-720]] & /@ bonds;

priceUS = FinancialData[#, "Price", DatePlus[-720]] & /@ stocksUS;
priceF = FinancialData[#, "Price", DatePlus[-720]] & /@ stocksF;
priceI = FinancialData[#, "Price", DatePlus[-720]] & /@ stocksI;
priceC = FinancialData[#, "Price", DatePlus[-720]] & /@ commodities;
priceB = FinancialData[#, "Price", DatePlus[-720]] & /@ bonds;

nameUS = FinancialData[#, "Name"] & /@ stocksUS;
nameF = FinancialData[#, "Name"] & /@ stocksF;
nameI = FinancialData[#, "Name"] & /@ stocksI;
nameC = FinancialData[#, "Name"] & /@ commodities;
nameB = FinancialData[#, "Name"] & /@ bonds;

symbUS = FinancialData[#, "Symbol"] & /@ stocksUS;
symbF = FinancialData[#, "Symbol"] & /@ stocksF;
symbI = FinancialData[#, "Symbol"] & /@ stocksI;
symbC = FinancialData[#, "Symbol"] & /@ commodities;
symbB = FinancialData[#, "Symbol"] & /@ bonds;
```

Yield-Curve Fitting

```
NelsonSiegelCurve[m_, β0_, β1_, β2_, τ_] := Module[{mt = m / τ, ratio}, ratio = (1 - e^-mt) / mt; β0 + β1 ratio + β2 (ratio - e^-mt)];

intdata = WolframAlpha["yield curve", {{"TreasuryYieldCurve:EconomicData", 2}, "ComputableData"},
  PodStates -> {"TreasuryYieldCurve:EconomicData__More"}]


intRates = Module[{tr = Transpose[intdata]},
   Transpose[{Which[! StringFreeQ[#, "month"], N[ToExpression[StringCases[#, DigitCharacter ..][[1]]] * 30],
        ! StringFreeQ[#, "year"], ToExpression[StringCases[#, DigitCharacter ..][[1]]] * 365] & /@ First[tr],
     tr[[2]]}]];

NSFit = FindFit[intRates, NelsonSiegelCurve[m, b0, b1, b2, t], {b0, b1, b2, t}, m, Method → NMinimize];
```

```
{{1-month treasury bill, 0.08}, {3-month treasury bill, 0.09}, {6-month treasury bill, 0.15},
 {1-year treasury bill, 0.19}, {2-year treasury note, 0.29}, {3-year treasury note, 0.38}, {5-year treasury note, 0.74},
 {7-year treasury note, 1.19}, {10-year treasury note, 1.76}, {20-year treasury bond, 2.5}, {30-year treasury bond, 2.91}}
```

Yield-Curve Fitting Plot Style

Yield-Curve Fitting Arrow

```
arrow = ImageResize[ , 350];
```

CUDA Link

```
Needs["CUDALink`"]
```

CUDA FD Plot Style

```
listplot3dsty = Sequence[ImageSize → 250,
   PlotStyle → Directive[wfpblue[.6], Specularity[White, 20]], BaseStyle → {7, FontFamily → wfpfont}];
```

Graph Theory and Finance

```
data = Join[Take[#[[All, 2]], -400] & /@ retI, Take[#[[All, 2]], -400] & /@ retC, Take[#[[All, 2]], -400] & /@ retB];
```

```
portcor = Correlation[Transpose[data]];

portm[θ_] := ReplacePart[portcor, {i_, i_} → 0] /. {x_ /; x > θ → 1, x_ /; x < θ → 0}

bigport = Join[symbI, symbC, symbB];
bigportnames = Join[nameI, nameC, nameB];

graphsty = Sequence[VertexSize → Large, GraphLayout → {"PackingLayout" → "ClosestPacking"}, ImagePadding → 20, VertexStyle →
    Join[Thread[Rule[Range[Length[symbI]], wfpblue[0]]], Thread[Rule[Length[symbI] + Range[Length[symbC]], wfpblue[.5]]],
     Thread[Rule[Length[symbI] + Length[symbC] + Range[Length[symbB]], wfpblue[1]]]
    ]];
graphlabels = Row[{Graphics[{EdgeForm[Black], wfpblue[0], Disk[]}, ImageSize → 10, ImageMargins → {{0, 0}, {3, 0}}],
    Style["Equity Indexes", 11, FontFamily → "Verdana"], "    ",
    Graphics[{EdgeForm[Black], wfpblue[.5], Disk[]}, ImageSize → 10, ImageMargins → {{0, 0}, {3, 0}}],
    Style["Commodities", 11, FontFamily → "Verdana"], "    ", Graphics[{EdgeForm[Black], wfpblue[1], Disk[]},
     ImageSize → 10, ImageMargins → {{0, 0}, {3, 0}}], Style["Bonds", 11, FontFamily → "Verdana"]}, Spacer[3]];
```

Quote Pane

Quote Pane Outputs

```
quotePane["I think there's no question Wolfram Finance Platform is the superior platform for this
   kind of project and development of this project.", "Bernard Gress, Financial Economist, Fannie Mae"]
quotePane["The different technologies that Wolfram Finance Platform has now that allow you to go from something
   very sophisticated, like the notebook interface, all the way to the web interface is very powerful.",
 "Philip Zecher, Chief Risk Officer, EQA Partners, LP"]
```