
WOLFRAM WHITE PAPER

CUDA Programming Using Wolfram *Finance Platform*[™]

WOLFRAM

Introduction

CUDA, short for Common Unified Device Architecture, is a C-like programming language developed by NVIDIA to facilitate general computation on the Graphical Processing Unit (GPU). CUDA allows users to design programs around the many-core hardware architecture of the GPU. And, by using many cores, carefully designed CUDA programs can achieve speedups (1000x in some cases) over a similar CPU implementation. Coupled with the investment price and power required to GFLOP (billion floating-point operations per second), the GPU has quickly become an ideal platform for both high-performance clusters and analysts wishing for a supercomputer at their disposal.

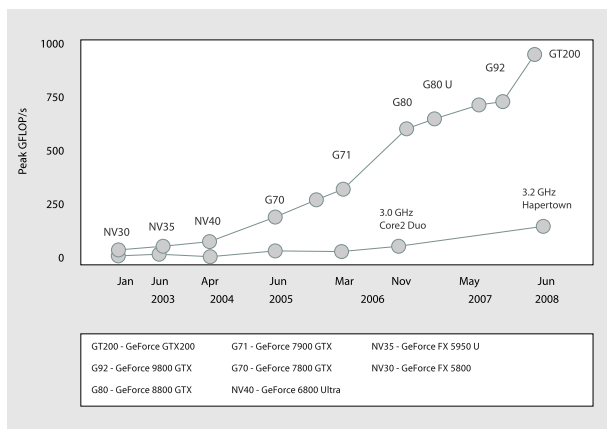
Yet while the user can achieve speedups, CUDA does have a steep learning curve—including learning the CUDA programming API and understanding how to set up CUDA and compile CUDA programs. This learning curve has, in many cases, alienated many potential CUDA programmers.

Wolfram *Finance Platform's* *CUDALink* simplified the use of the GPU by introducing dozens of functions to tackle areas ranging from linear algebra to option pricing using Monte Carlo methods. *CUDALink* also allows the user to load their own CUDA functions into the kernel.

By utilizing Wolfram *Finance Language's* language, mirroring function syntax, and integrating with existing programs and development tools, *CUDALink* offers an easy way to use CUDA. In this document we describe the benefits of CUDA integration in Wolfram *Finance Platform* and provide some applications for which it is suitable.

Motivations for *CUDALink*

CUDA is a C-like language designed to write general programs around the NVIDIA GPU hardware. By programming the GPU, users can get performance unrivaled by a CPU for a similar investment. The following graph shows the performance of the GPU compared to the CPU:



Today, GPUs priced at just \$500 can achieve performance of 2 TFLOP (trillion operations per second). The GPU also competes with the CPU in terms of power consumption, using a fraction of the power compared to the CPU for the same GFLOP performance.

Because GPUs are off-the-shelf hardware, can fit into a standard desktop, have low power consumption, and perform exceptionally, they are very attractive to users. Yet a steep learning curve has always been a hindrance for users wanting to use CUDA in their applications.

Wolfram *Finance Platform's* *CUDALink* alleviates much of the burden required to use CUDA. *CUDALink* allows users to query system hardware, use the GPU for dozens of functions, and define new CUDA functions to be run on the GPU.

How *CUDALink* Makes GPU Programming Easy

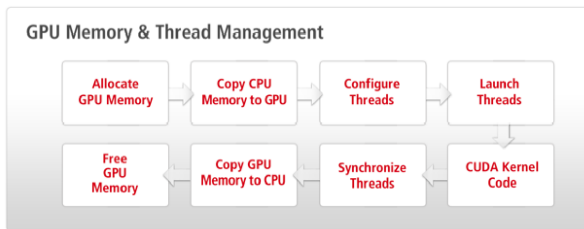
CUDALink offers a high-level interface to the GPU built on top of Wolfram *Finance Platform*'s development technologies. It allows users to execute code on their GPU with minimal effort. By fully integrating and automating the GPU's capabilities using Wolfram *Finance Platform*, users experience a more productive and efficient development cycle.

Automation of development project management

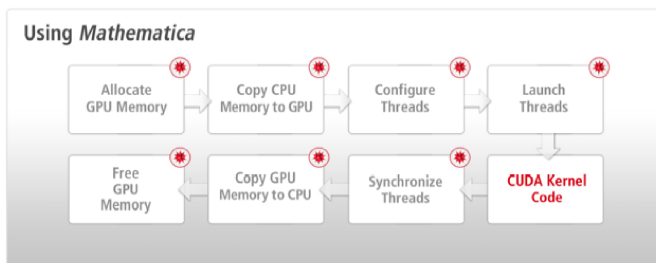
Unlike other development frameworks that require the user to manage project setup, platform dependencies, and device configuration, *CUDALink* makes the process transparent and automated.

Automated GPU memory and thread management

A CUDA program written from scratch delegates memory and thread management to the programmer. This bookkeeping is required in lieu of the need to write the CUDA program.

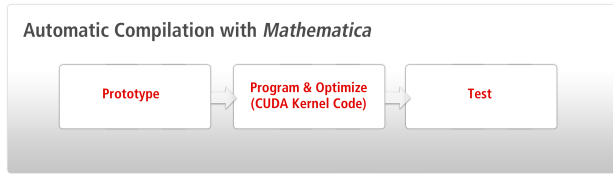


With Wolfram *Finance Platform*, memory and thread management is automatically handled for the user.



The Wolfram *Finance Platform* memory manager handles memory transfers intelligently in the background. Memory, for example, is not copied to the GPU until computation is needed and is flushed out when the GPU memory gets full.

Wolfram *Finance Platform*'s CUDA support streamlines the whole programming process. This allows GPU programmers to follow a more interactive style of programming:



Integration with Wolfram *Finance Platform*'s built-in capabilities

CUDA integration provides full access to Wolfram *Finance Platform*'s native language and built-in functions.

With Wolfram *Finance Platform*'s comprehensive symbolic and numerical functions, built-in application area support, and graphical interface-building functions, users can write hybrid algorithms that use the CPU and GPU, depending on the efficiency of each algorithm.

Ready-to-use applications

CUDA integration in Wolfram *Finance Platform* provides several ready-to-use CUDA functions that cover a broad range of topics such as mathematics, image processing, financial engineering, and more. Examples will be given in the section *CUDALink* Applications.

Zero device configuration

Wolfram *Finance Platform* automatically finds, configures, and makes CUDA devices available to the users.

Multiple GPU support

Through Wolfram *Finance Platform*'s built-in parallel programming support, users can launch CUDA programs on different GPUs. Users can also scale the setup across machines and networks using *gridMathematica*®.

GPU Programming with *CUDALink*

CUDALink provides a powerful interface for using CUDA within Wolfram *Finance Platform*. Through *CUDALink*, users get carefully tuned linear algebra, Fourier transform, financial derivative, and image processing algorithms. Users can also write their own *CUDALink* modules with little effort.

Accessing system information

CUDALink supplies functions that query the system's GPU hardware. To use *CUDALink* operations, users have to first load the *CUDALink* application:

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

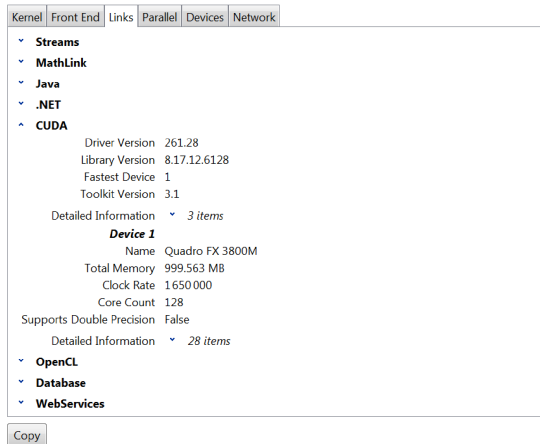
CUDAQ tells whether the current hardware and system configuration support *CUDALink*:

```
CUDAQ [ ]
```

```
True
```

SystemInformation gives information on the available GPU hardware:

```
SystemInformation [ ]
```



Example of a report generated by SystemInformation.

Integration with Wolfram *Finance Platform* functions

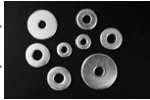
CUDALink integrates with existing Wolfram *Finance Platform* functions such as its import/export facilities, functional language, and interface building. This allows you to build deployable programs in Wolfram *Finance Platform* with minimal disruption to the GPU task. This section showcases how you can build interfaces as well as use the import/export capabilities in Wolfram *Finance Platform*.

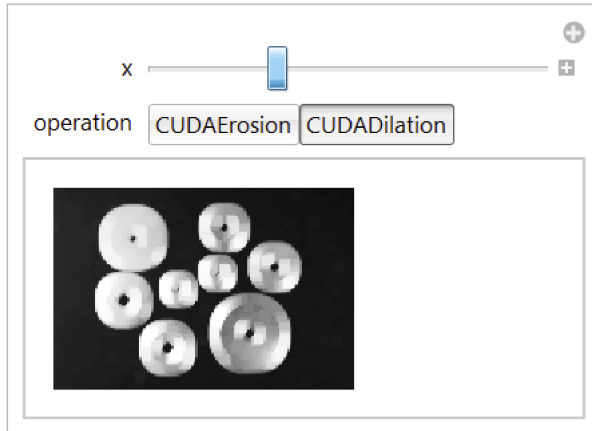
Manipulate: Wolfram *Finance Platform*'s automatic interface generator

Wolfram *Finance Platform* provides extensive built-in interface-building functions. Users can customize controls using its highly declarative interface language.

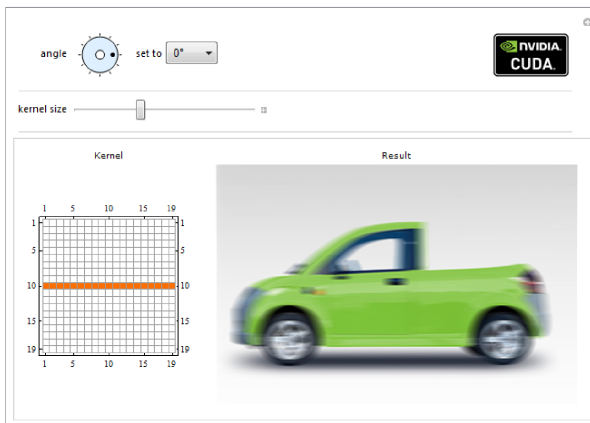
One fully automated interface-generating function is `Manipulate`, which builds the interface by inspecting the possible values of variables. It then chooses the appropriate GUI widget based on the interpretation of the variable values.

Here, we build an interface that performs a morphological operation on an image with varying radii:

```
Manipulate[operation[, x],  
{x, 0, 9}, {operation, {CUAERosion, CUDADilation}}]
```



Using the same technique, you can build more complicated interfaces. This allows users to choose different Gaussian kernel sizes (and their angle) and performs a convolution on the image on the right:



Example of a user interface built with Manipulate.

Support for import and export

Wolfram *Finance Platform* natively supports hundreds of file formats and their subformats for importing and exporting. Supported formats include: common image formats (JPEG, PNG, TIFF, BMP, etc.), video formats (AVI, MOV, H264, etc.), audio formats (WAV, AU, AIFF, FLAC, etc.), medical imaging formats (DICOM), data formats (Excel, CSV, MAT, etc.), and various raw formats for further processing.

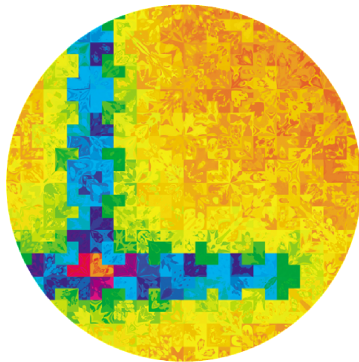
Any supported data formats will be automatically converted to Wolfram *Finance Platform*'s unified data representation, or an *expression*, which can be used in all Wolfram *Finance Platform* functions, including *CUDALink* functions.

Users can also get data from the web or Wolfram-curated datasets. The following code imports an image from a given URL:

```
image = Import[  
  "http://gallery.wolfram.com/2d/popup/00_contourMosaic.pop.jpg"];
```

The function `Import` automatically recognizes the file format and converts it into a Wolfram *Finance Platform* expression. This can be directly used by *CUDALink* functions, such as `CUDAImageAdd`:

```
output = CUDAImageAdd[image, 
```



All outputs from Wolfram *Finance Platform* functions, including the ones from *CUDALink* functions, are also expressions and can be easily exported to one of the supported formats using the `Export` function. For example, the following code exports the above output into PNG format:

```
Export["masked.png", output]  
masked.png
```

CUDALink programming

Programming the GPU in Wolfram *Finance Platform* is straightforward. It begins with writing a CUDA kernel. Here, we will create a simple example that negates colors of a three-channel image:

```
kernel = "  
__global__ void cudaColorNegate(mint  
  *img, mint *dim, mint channels) {  
  int width = dim[0], height = dim[1];  
  int xIndex = threadIdx.x + blockIdx.x * blockDim.x;  
  int yIndex = threadIdx.y + blockIdx.y * blockDim.y;  
  int index = channels * (xIndex + yIndex*width);  
  if (xIndex < width && yIndex < height) {  
    for (int c = 0; c < channels; c++)  
      img[index + c] = 255 - img[index + c];}}";
```

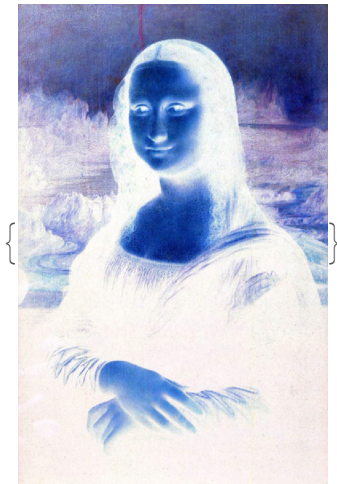

Pass that string to the built-in function `CUDAFunctionLoad`, along with the kernel function name and the argument specification. The last argument denotes the CUDA block size:

```
colorNegate = CUDAFunctionLoad[kernel, "cudaColorNegate",  
  {{_Integer, "InputOutput"},  
   {_Integer, "Input"}, _Integer}, {16, 16}];
```

Several things are happening at this stage. Wolfram *Finance Platform* automatically compiles the kernel function and loads it as a Wolfram *Finance Platform* function. Now you can apply this new CUDA function to an image:

```
img = ;
```

```
colorNegate[img, ImageDimensions[img], ImageChannels[img]]
```



System requirements

To use Wolfram *Finance Platform*'s *CUDALink*, the following is required:

- Operating System: Windows, Linux, and Mac OS X 10.6.3+, both 32- and 64-bit architecture.
- NVIDIA CUDA-enabled products.
- For CUDA programming, a *CUDALink*-supported C compiler is required.

CUDALink Applications

In addition to support for user-defined CUDA functions and automatic compilation, *CUDALink* includes several ready-to-use functions ranging from image processing to financial option valuation.

Financial engineering

CUDALink's options pricing function uses the binomial or Monte Carlo method, depending on the type of option selected. Computing options on the GPU can be dozens of times faster than using the CPU.

This generates some random input data:

```
numberOfOptions = 32;  
S = RandomReal[{25.0, 35.0}, numberOfOptions];  
X = RandomReal[{20.0, 40.0}, numberOfOptions];  
T = RandomReal[{0.1, 10.0}, numberOfOptions];  
R = RandomReal[{0.03, 0.07}, numberOfOptions];  
Q = RandomReal[{0.01, 0.04}, numberOfOptions];  
V = RandomReal[{0.10, 0.50}, numberOfOptions];
```

This computes the Asian arithmetic call option with the above data:

```
CUDAFinancialDerivative[{"AsianArithmetic", "Call"},  
  {"StrikePrice" → X, "Expiration" → T}, {"CurrentPrice" → S,  
  "InterestRate" → R, "Volatility" → V, "Dividend" → Q}]  
{5.15341, 7.45667, 0.0895437, 8.6807, 0.00231528, 0.629001,  
  3.82846, 6.21568, 9.5776, 6.85937, 3.43729, 0.000196773,  
  9.39402, 3.23904, 6.70806, 0.916565, 3.19008, 1.95835,  
  1.92784, 1.14869, 0.845791, 1.69902, 3.97125, 9.50727, 6.29223,  
  2.47507, 4.85501, 2.95033, 0.969712, 4.32193, 9.78511, 6.11466}
```

Black-Scholes

For options with no built-in implementation in *CUDALink*, users can load their own. Here, we will show how to load the Black-Scholes model for calculating the vanilla European option and in the next section we will show how to load code to compute the binary call and put of an asset-or-nothing option.

Recall from above that the call option in the Black-Scholes model is defined by:

$$C(S, t) = N(d_1) S e^{-q(T-t)} - N(d_2) X e^{-r(T-t)}$$

with

$$d_1 = \frac{(T-t)\left(r - q + \frac{\sigma^2}{2}\right) + \text{Log}\left(\frac{S}{X}\right)}{\sqrt{T-t} \sigma}$$
$$d_2 = d_1 - \sigma \sqrt{T-t}$$

$N(p)$ is the cumulative distribution function of the normal distribution.

The following CUDA code computes the call option when $t = 0$:

```
code = "  
#define N(x)      (erf((x)/sqrt(2.0))/2+0.5)  
__device__ Real_t iBlackSholes(Real_t & S, Real_t &  
    X, Real_t & T, Real_t & R, Real_t & Q, Real_t & V) {  
    Real_t d1 = (log(S/X) + (R - Q + V*V/2)*T)/(V*sqrt(T));  
    Real_t d2 = d1 - V*sqrt(T);  
    Real_t call = S*exp(-Q*T)*N(d1) - X*exp(-R*T)*N(d2);  
    return call > 0 ? call : 0;  
}  
__global__ void blackScholes(Real_t *  
    call, Real_t * S, Real_t * X, Real_t * T, Real_t  
    * R, Real_t * Q, Real_t * V, mint length) {  
    int ii = threadIdx.x + blockIdx.x*blockDim.x;  
    if (ii < length) {  
        call[ii] =  
        iBlackSholes(S[ii], X[ii], T[ii], R[ii], Q[ii], V[ii]);  
    }  
}";
```

This loads the above code into Wolfram *Finance Platform*:

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

Here we generate some random input data for the model. We are only computing 32 options:

```
numberOfOptions = 32;  
S = RandomReal[{25.0, 35.0}, numberOfOptions];  
X = RandomReal[{20.0, 40.0}, numberOfOptions];  
T = RandomReal[{0.1, 10.0}, numberOfOptions];  
R = RandomReal[{0.03, 0.07}, numberOfOptions];  
Q = RandomReal[{0.01, 0.04}, numberOfOptions];  
V = RandomReal[{0.10, 0.50}, numberOfOptions];
```

This allocates memory for the call result:

```
call = CUDAMemoryAllocate[Real, numberOfOptions];
```

This calls the function:

```
CUDABlackScholes[call, S, X, T, R, Q, V, numberOfOptions]  
{CUDAMemory[<10008>, Float]}
```

This retrieves the CUDA memory back into Wolfram *Finance Platform*:

```
CUDAMemoryGet[call]  
{11.129, 4.38379, 6.52147, 5.49432, 11.5947, 11.9229, 8.12619, 9.09809,  
15.2792, 3.28516, 5.08137, 8.62249, 10.0763, 4.54254, 7.91241, 1.03392,  
1.50925, 2.75238, 3.28798, 7.01667, 8.48418, 4.26581, 5.73602, 10.906,  
4.10028, 9.03023, 6.01691, 8.70756, 1.06836, 2.56222, 10.6502, 9.37264}
```

Binary option

Using the same Black-Scholes model, we can calculate both the asset-or-nothing call and put for the binary/digital option model:

```
code = "  
#define N(x)      (erf((x)/sqrt(2.0))/2+0.5)  
__device__ void iBinaryAssetOption(Real_t  
    & call, Real_t & put, Real_t & S, Real_t & X,  
    Real_t & T, Real_t & R, Real_t & Q, Real_t & V) {  
    Real_t d1 = (log(S/X) + (R - Q + V*V/2)*T)/(V*sqrt(T));  
    call = S * exp(-Q * T) * N(d1);  
    put = S * exp(-Q * T) * N(-d1);  
}  
__global__ void binaryAssetOption(Real_t * call,  
    Real_t * put, Real_t * S, Real_t * X, Real_t * T,  
    Real_t * R, Real_t * Q, Real_t * V, mint length) {  
    int ii = threadIdx.x + blockIdx.x*blockDim.x;  
    if (ii < length) {  
        iBinaryAssetOption(call[ii],  
            put[ii], S[ii], X[ii], T[ii], R[ii], Q[ii], V[ii]);  
    }  
};
```

This loads the function into Wolfram *Finance Platform*:

```
CUDABinaryAssetOption = CUDAFunctionLoad[code, "binaryAssetOption",  
    {{_Real, "Output"}, {_Real, "Output"}, {_Real, "Input"},  
    {_Real, "Input"}, {_Real, "Input"}, {_Real, "Input"},  
    {_Real, "Input"}, {_Real, "Input"}, _Integer, 128];
```

This creates some random data for the strike price, expiration, etc.:

```
numberOfOptions = 32;  
S = RandomReal[{25.0, 35.0}, numberOfOptions];  
X = RandomReal[{20.0, 40.0}, numberOfOptions];  
T = RandomReal[{0.1, 10.0}, numberOfOptions];  
R = RandomReal[{0.03, 0.07}, numberOfOptions];  
Q = RandomReal[{0.01, 0.04}, numberOfOptions];  
V = RandomReal[{0.10, 0.50}, numberOfOptions];
```

The call memory and put memory are allocated:

```
call = CUDAMemoryAllocate[Real, numberOfOptions];  
put = CUDAMemoryAllocate[Real, numberOfOptions];
```

This calls the function, returning the call and put memory handles:

```
CUDABinaryAssetOption[call, put, S, X, T, R, Q, V, numberOfOptions]  
{CUDAMemory[<17941>, Float], CUDAMemory[<17645>, Float]}
```

Both the call memory and put memory can be retrieved using `CUDAMemoryGet`:

CUDAMemoryGet [call]

```
{10.081, 19.8343, 24.916, 15.8739, 16.1053, 20.9529, 21.1403, 20.1766,  
11.6614, 15.0135, 19.413, 14.3981, 22.8407, 21.4579, 13.5464, 16.4192,  
17.2243, 16.9844, 21.422, 19.9588, 14.1371, 29.3712, 20.5797, 14.0867,  
17.4331, 16.4028, 10.359, 24.0818, 10.6085, 18.5223, 12.7078, 26.5564}
```

CUDAMemoryGet [put]

```
{18.0516, 4.14473, 0.770478, 11.5232, 6.08763, 8.3997,  
4.83071, 5.32312, 8.5637, 6.14408, 11.5159, 9.26106, 9.74336,  
4.77558, 15.315, 10.1546, 12.2944, 7.02279, 2.64413, 12.0883,  
10.3472, 0.702504, 11.0442, 13.5412, 12.4254, 8.10019,  
12.5802, 2.04422, 17.2133, 6.06415, 12.2815, 8.12901}
```

Random number generators

One of the difficult problems when parallelizing algorithms is generating good random numbers. *CUDALink* offers many examples on how to generate both pseudorandom and quasirandom numbers. Here, we generate quasirandom numbers using the Halton sequence:

```
src = "  
__device__ int primes[] = {  
    2, 3, 5, 7, 11, 13, 17, 19, 23, 29,  
    31, 37, 41, 43, 47, 53, 59, 61, 67, 71,  
    73, 79, 83, 89, 97,101,103,107,109,113,  
    127,131,137,139,149,151,157,163,167,173,  
    179,181,191,193,197,199,211,223,227,229};  
__global__ void Halton(Real_t * out, int dim, int n) {  
    const int tx = threadIdx.x, bx = blockIdx.x, dx = blockDim.x;  
    const int index = tx + bx*dx;  
  
    if (index >= n)  
        return ;  
  
    Real_t digit, rnd, idx, half;  
    for (int ii = 0,  
        idx=index, rnd=0, digit=0; ii < dim; ii++) {  
        half = 1.0/primes[ii];  
        while (idx > 0.0001) {  
            digit = ((mint)idx)%primes[ii];  
            rnd += half*digit;  
            idx = (idx - digit)/primes[ii];  
            half /= primes[ii];  
        }  
        out[index*dim + ii] = rnd;  
    }  
}  
";
```

This loads the CUDA source into Wolfram *Finance Platform*:

```
CUDAHaltonSequence = CUDAFunctionLoad[src, "Halton",  
  {[_Real, "Output"], "Integer32", "Integer32"}, 256 ]  
CUDAFunction[<>, Halton, {[_Real, Output], Integer32, Integer32}]
```

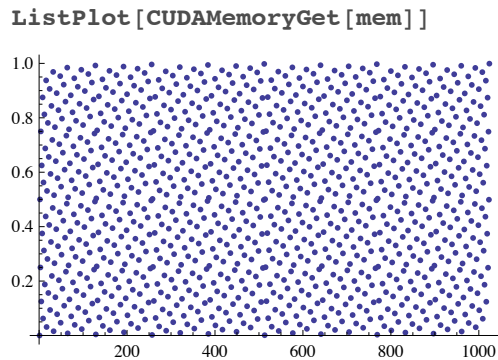
This allocates 1024 real elements. Real elements are interpreted to be the highest floating precision on the machine:

```
mem = CUDAMemoryAllocate[Real, {1024}]  
CUDAMemory[<11521>, Double]
```

This calls the function:

```
CUDAHaltonSequence[mem, 1, 1024]  
{CUDAMemory[<11521>, Double]}
```

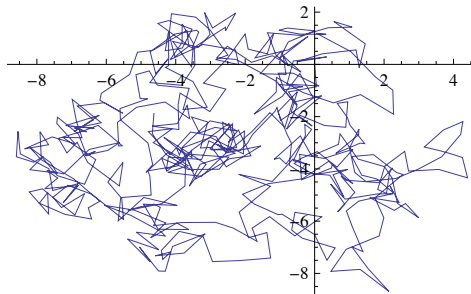
You can use Wolfram *Finance Platform's* extensive visualization support to visualize the result. Here we plot the data:



Some random number generators and distributions are not naturally parallelizable. In those cases, users can adopt a hybrid GPU programming approach—utilizing the CPU for some tasks and the GPU for others. Using this approach, users can use Wolfram *Finance Platform's* extensive statistics capabilities to generate or derive distributions from their data.

Here, we simulate a random walk by generating numbers on the CPU, performing a reduction (using `CUDAFoldList`) on the GPU, and plotting the result using Wolfram *Finance Platform*:

```
ListLinePlot[  
  Thread[List[CUDAFoldList[Plus, 0, RandomReal[{-1, 1}, 500]],  
    CUDAFoldList[Plus, 0, RandomReal[{-1, 1}, 500]]]]]
```



Linear algebra

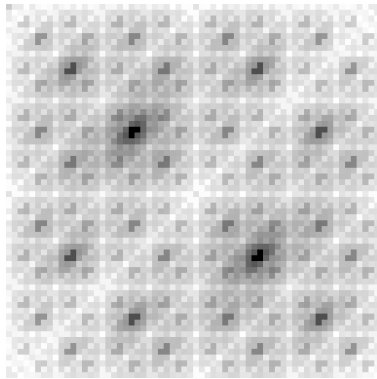
You can perform various linear algebra functions with *CUDALink*, such as matrix–matrix and matrix–vector multiplication, finding minimum and maximum elements, and transposing matrices:

```
Nest [CUDADot [RandomReal [1, {100, 100}], #] &,
      RandomReal [1, {100}], 1000];
```

Fourier analysis

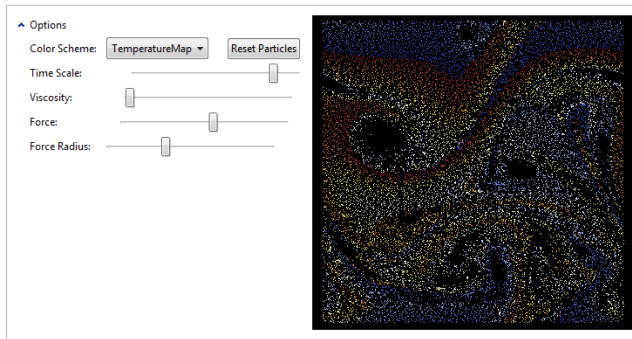
Find the logarithmic power spectrum of a dataset using the Fourier analysis capabilities of the *CUDALink* package, which include forward and inverse Fourier transforms that can operate on a list of 1D, 2D, or 3D real or complex numbers:

```
ArrayPlot [Log [Abs [CUDAFourier [Table [
      Mod [Binomial [i, j], 2], {i, 0, 63}, {j, 0, 63}]]]], Frame → False]
```



PDE solving

This computational fluid dynamics example is included with *CUDALink*. This solves the Navier–Stokes equations for a million particles using the finite–element method:



OpenCL Integration

Wolfram *Finance Platform* also includes the ability to use the GPU using OpenC via *OpenCLLink*. This is a vendor-neutral way to use the GPU and works both on NVIDIA and non-NVIDIA hardware. *OpenCLLink* and *CUDALink* offer the same syntax, and the following demonstrates how to compute the one-touch option:

```
code = "  
#define N(x)      (erf((x)/sqrt(2.0))/2+0.5)  
#ifndef USING_DOUBLE_PRECISIONQ  
#pragma OPENCL EXTENSION cl_khr_fp64 : enable  
#endif /* USING_DOUBLE_PRECISIONQ */  
__kernel void onetouch(__global Real_t * call, __global  
Real_t * put, __global Real_t * S, __global Real_t *  
X, __global Real_t * T, __global Real_t * R, __global  
Real_t * D, __global Real_t * V, mint length) {  
Real_t tmp, d1, d5, power;  
int ii = get_global_id(0);  
if (ii < length) {  
    d1 = (log(S[ii]/X[ii]) + (R[ii] - D[ii] + 0.5f  
* V[ii] * V[ii]) * T[ii]) / (V[ii] * sqrt(T[ii]));  
    d5 = (log(S[ii]/X[ii]) - (R[ii] - D[ii] + 0.5f *  
V[ii] * V[ii]) * T[ii]) / (V[ii] * sqrt(T[ii]));  
    power = pow(X[ii]/S[ii], 2*R[ii]/(V[ii]*V[ii]));  
    call[ii] = S[ii] < X[ii]  
? power * N(d5) + (S[ii]/X[ii])*N(d1) : 1.0;  
    put[ii] = S[ii] > X [ii] ? power * N(-d5)  
+ (S[ii]/X[ii])*N(-d1) : 1.0;  
}  
};
```

This loads the OpenCL function into Wolfram *Finance Platform*:

```
OpenCLOneTouchOption = OpenCLFunctionLoad[code, "onetouch",  
{_Real, "Output"}, {_Real, "Output"}, {_Real, "Input"},  
{_Real, "Input"}, {_Real, "Input"}, {_Real, "Input"},  
{_Real, "Input"}, {_Real, "Input"}, _Integer}, 128];
```

This generates random input data:

```
numberOfOptions = 64;  
S = RandomReal[{20.0, 40.0}, numberOfOptions];  
X = RandomReal[{20.0, 40.0}, numberOfOptions];  
T = RandomReal[{0.1, 10.0}, numberOfOptions];  
R = RandomReal[{0.02, 0.1}, numberOfOptions];  
Q = RandomReal[{0.0, 0.08}, numberOfOptions];  
V = RandomReal[{0.1, 0.4}, numberOfOptions];
```

This allocates memory for both the call and put results:

```
call = OpenCLMemoryAllocate[Real, numberOfOptions];  
put = OpenCLMemoryAllocate[Real, numberOfOptions];
```

This calls the function:

```
OpenCLOneTouchOption[call, put, S, X, T, R, Q, V, numberOfOptions]  
{OpenCLMemory[<19661>, Double], OpenCLMemory[<28475>, Double]}
```


This retrieves the result for the call option (the put option can be retrieved similarly):

OpenCLMemoryGet [call]

```
{1., 0.398116, 1., 1., 1.00703, 0.909275, 1., 1., 1., 0.541701, 0.631649,  
1., 0.702748, 1., 1., 1., 0.626888, 1., 1., 0.827843, 0.452237,  
0.998761, 0.813008, 1., 1., 0.96773, 0.795428, 1., 1.79325, 1.,  
1., 1., 1., 0.547425, 0.968162, 1., 1., 0.907489, 1., 1.90031,  
0.316174, 1., 0.998824, 0.383825, 1., 0.804287, 0.977305,  
1., 1., 0.855764, 1., 0.952568, 0.573249, 0.239455, 0.635454,  
0.917078, 0.624179, 1., 0.679681, 1., 1., 0.968929, 0.712148}
```

Summary

Due to Wolfram *Finance Platform*'s integrated platform design, all functionality is included without the need to buy and maintain multiple tools and add-on packages.

With its simplified development cycle, multicore computing, and built-in functions, Wolfram *Finance Platform*'s built-in *CUDALink* application provides a powerful high-level interface for GPU computing.

Notes

Technologies underlying *CUDALink*

GPU integration in Wolfram *Finance Platform* has only been possible due to advancements in system integration introduced in recent versions of the software. Features such as C code generation, SymbolicC manipulation, dynamic library loading, and C compiler invocation are all used internally by *CUDALink* to enable fast and easy access to the GPU.

C code generation

Wolfram *Finance Platform* introduces the ability to export expressions written using `Compile` to a C file. The C file can then be compiled and run either as a Wolfram *Finance Platform* command (for native speed), or be integrated with an external application. Through the code-generation mechanism, you can use Wolfram *Finance Platform* for both prototype and native-speed deployment.

To motivate the C code generation feature, we will solve the call option using the Black-Scholes equation. The European call option in terms of the Black-Scholes equation is defined by:

$$C(S, t) = N(d_1) S e^{-q(T-t)} - N(d_2) X e^{-r(T-t)}$$

with

$$d_1 = \frac{(T-t)\left(r - q + \frac{\sigma^2}{2}\right) + \text{Log}\left(\frac{S}{X}\right)}{\sqrt{T-t} \sigma}$$
$$d_2 = d_1 - \sigma \sqrt{T-t}$$

$N(p)$ is the cumulative distribution function of the normal distribution.

Here, we define the equation for $t = 0$:

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

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

$$\text{BlackScholes} = \text{CDF}[\text{NormalDistribution}[0, 1], d_1] S e^{-qT} - \text{CDF}[\text{NormalDistribution}[0, 1], d_2] X e^{-rT}$$

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

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

The following command generates the C code, compiles it, and links it back into Wolfram *Finance Platform* to provide native speed:

```
cf = Compile[{{S, _Real}, {X, _Real}, {σ, _Real},
             {T, _Real}, {r, _Real}, {q, _Real}}, BlackScholes,
             CompilationOptions -> {"InlineExternalDefinitions" -> True},
             CompilationTarget -> "C"];
```

The function can be used like any other Wolfram *Finance Platform* function. Here we call the above function:

```
cf[50.0, 50.0, 0.4, 1.0, 0.05, 0.02]
8.39968
```

LibraryLink

LibraryLink allows you to load C functions as Wolfram *Finance Platform* functions. It is similar in purpose to *MathLink*, but by running in the same process as the Wolfram *Finance Platform* kernel, it avoids the memory transfer cost associated with *MathLink*. This loads a C function from a library; the function adds one to a given integer:

```
addOne = LibraryFunctionLoad["demo", "demo_I_I", {Integer}, Integer]
LibraryFunction[<>, demo_I_I, {Integer}, Integer]
```

The library function is run with the same syntax as any other function:

```
addOne[3]
4
```

CUDALink and *OpenCLLink* are examples of *LibraryLink*'s usage.

Symbolic C code

Using Wolfram *Finance Platform's* symbolic capabilities, users can generate C programs within Wolfram *Finance Platform*. The following, for example, creates macros for common math constants:

```
<< SymbolicC`
```

These are all constants in the Wolfram *Finance Platform* system context. We use Wolfram *Finance Platform's* `CDefine` to declare a C macro:

```
s = Map[CDefine[ToString[#], N[#]] &, Map[ToExpression,
      Select[Names["System`*"], MemberQ[Attributes[#], Constant] &]]]
{CDefine[Catalan, 0.915966],
 CDefine[Degree, 0.0174533], CDefine[E, 2.71828],
 CDefine[EulerGamma, 0.577216], CDefine[Glaisher, 1.28243],
 CDefine[GoldenRatio, 1.61803], CDefine[Khinchin, 2.68545],
 CDefine[MachinePrecision, 15.9546], CDefine[Pi, 3.14159]}
```

The symbolic expression can be converted to C using the `ToCCodeString` function:

```
ToCCodeString[s]
"#define Catalan 0.915965594177219\n#define Degree
 0.017453292519943295\n#define E 2.718281828459045\n#define
EulerGamma 0.5772156649015329\n#define
Glaisher 1.2824271291006226\n#define
GoldenRatio 1.618033988749895\n#define Khinchin
 2.6854520010653062\n#define MachinePrecision
 15.954589770191003\n#define Pi 3.141592653589793\n"
```

By representing the C program symbolically, you can manipulate it using standard Wolfram *Finance Platform* techniques. Here, we convert all the macro names to lowercase:

```
ReplaceAll[s, CDefine[name_, val_] → CDefine[ToLowerCase[name], val]]
{CDefine[catalan, 0.915966],
 CDefine[degree, 0.0174533], CDefine[e, 2.71828],
 CDefine[eulergamma, 0.577216], CDefine[glaisher, 1.28243],
 CDefine[goldenratio, 1.61803], CDefine[khinchin, 2.68545],
 CDefine[machineprecision, 15.9546], CDefine[pi, 3.14159]}
```

Again, the code can be converted to C code using `ToCCodeString`:

```
ToCCodeString[%]
"#define catalan 0.915965594177219\n#define degree
 0.017453292519943295\n#define e 2.718281828459045\n#define
eulergamma 0.5772156649015329\n#define
glaisher 1.2824271291006226\n#define
goldenratio 1.618033988749895\n#define khinchin
 2.6854520010653062\n#define machineprecision
 15.954589770191003\n#define pi 3.141592653589793\n"
```

C compiler invoking

Another recent Wolfram *Finance Platform* innovation is the ability to call external C compilers from within Wolfram *Finance Platform*. The following compiles a simple C program into an executable:

```
<< CCompilerDriver`  
  
exe = CreateExecutable["  
#include <stdio.h>  
int main(void) {  
    printf("Hello from CCompilerDriver.");  
    return 0;  
}", "foo"];
```

Using the above syntax, you can create executables using any Wolfram *Finance Platform*-supported C compiler (Visual Studio, GCC, Intel CC, etc.) in a compiler-independent fashion. The above command can be executed within Wolfram *Finance Platform*:

```
Import["!" <> exe, "Text"]  
  
Hello from CCompilerDriver.
```

By using the Wolfram *Finance Platform* enhancements mentioned earlier in this section, *CUDALink* and *OpenCLLink* facilitate fast and simple access to the GPU.

Pricing and Licensing Information

Wolfram Research offers many flexible licensing options for both organizations and individuals. You can choose a convenient, cost-effective plan for your workgroup, department, directorate, university, or just yourself, including network licensing for groups.

Visit us online for more information:

www.wolfram.com/finance-platform/contact-us

Recommended Next Steps

Watch videos about Wolfram *Finance Platform*

www.wolfram.com/broadcast/video.php?channel=249

Request a free trial or schedule a technical demo

www.wolfram.com/finance-platform/contact-us

Learn more about Wolfram *Finance Platform*

US and Canada

1-800-WOLFRAM (965-3726)

info@wolfram.com

Europe

+44-(0)1993-883400

info@wolfram.co.uk

Outside US and Canada (except Europe and Asia)

+1-217-398-0700

info@wolfram.com

Asia

+81-(0)3-3518-2880

info@wolfram.co.jp

© 2013 Wolfram Research, Inc. *Mathematica* and *gridMathematica* are registered trademarks and Wolfram *Finance Platform* is a trademark of Wolfram Research, Inc. All other trademarks are the property of their respective owners. *Mathematica* is not associated with Mathematica Policy Research, Inc.