Wolfram White Paper

CUDA Programming within *Mathematica*®

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 scientists wishing for a super computer at their disposal.

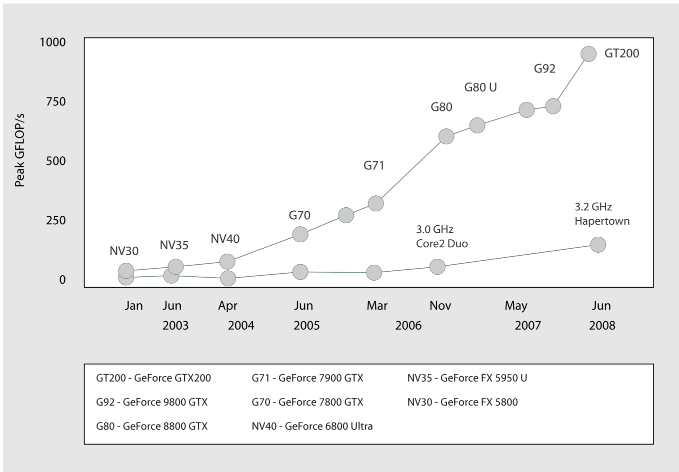
Yet while the user can achieve speedups, CUDA does have a steep learning curve—including learning the CUDA programming API, and understanding how to set up CUDA and compile CUDA programs. This learning curve has, in many cases, alienated many potential CUDA programmers.

Mathematica's *CUDALink* simplified the use of the GPU within *Mathematica* by introducing dozens of functions to tackle areas ranging from image processing to linear algebra. *CUDALink* also allows the user to load their own CUDA functions into the *Mathematica* kernel.

By utilizing *Mathematica*'s language, mirroring *Mathematica* 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 *Mathematica* and provide some applications for which it is suitable.

Motivations for *Mathematica 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.

Mathematica's CUDALink alleviates much of the burden required to use CUDA from within *Mathematica*. *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.

A Brief Introduction to *Mathematica*

In its over 20 years of existence, *Mathematica* has always been at the forefront of technical computing, providing a highly expressive language and inventing and implementing groundbreaking algorithms.

Mathematica is a development environment that combines a flexible programming language with a wide range of symbolic and numeric computational capabilities, production of high-quality visualizations, built-in application area packages, and a range of immediate deployment options. Combined with integration of dynamic libraries, automatic interface construction, and C code generation, *Mathematica* is the most sophisticated build-to-deploy environment on the market today.

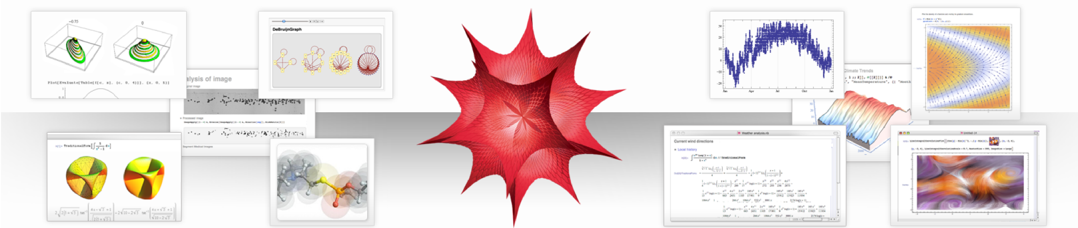
Mathematica 8 introduces the ability to program the GPU. For developers, this new integration means native access to *Mathematica*'s computing abilities—creating hybrid algorithms that combine the CPU and the GPU. Some of *Mathematica*'s features include:

Full-featured, unified development environment

Through its unique interface and integrated features for computation, development, and deployment, *Mathematica* provides a streamlined workflow. Wolfram Research also offers Wolfram *Workbench™*, a state-of-the-art integrated development engine based on the Eclipse platform.

Unified data representation

At the core of *Mathematica* is the foundational idea that everything—data, programs, formulas, graphics, documents—can be represented as symbolic entities, called *expressions*. This unified representation makes *Mathematica*'s language and functions extremely flexible, streamlined, and consistent.



Multiparadigm programming language

Mathematica provides its own highly declarative functional language, as well as several different programming paradigms, such as procedural and rule-based programming. Programmers can choose their own style for writing code with minimal effort. Along with comprehensive documentation and resources, *Mathematica*'s flexibility greatly reduces the cost of entry for new users.

Symbolic-numeric hybrid system

The principle behind *Mathematica* is full integration of symbolic and numeric computing capabilities. Through its full automation and preprocessing mechanisms, users reap the power of a hybrid computing system without needing knowledge of specific methodologies and algorithms.

Scientific and technical area coverage

Mathematica provides thousands of built-in functions and packages that cover a broad range of scientific and technical computing areas, such as statistics, control systems, data visualization, and image processing. All functions are carefully designed and tightly integrated with the core system.

High-performance computing

Mathematica has built-in support for multicore systems, utilizing all cores on the system for optimal performance. Many functions automatically utilize the power of multicore processors, and built-in parallel constructs make high-performance programming easy.

Data access and connectivity

Mathematica natively supports hundreds of formats for importing and exporting, as well as real-time access to data from WolframlAlpha®, Wolfram Research's computational knowledge engine™. It also provides APIs for accessing many programming languages and databases, such as C/C++, Java, .New, MySQL, and Oracle.

Platform-independent deployment options

Through its interactive notebooks, *Mathematica Player*™, and browser plugins, *Mathematica* provides a wide range of options for deployment. Built-in code generation functionality can be used to create standalone programs for independent distribution.

Scalability

Wolfram Research's grid*Mathematica* ™ allows *Mathematica* programs to be parallelized on many machines in cluster or grid configuration.

CUDA Integration in *Mathematica*

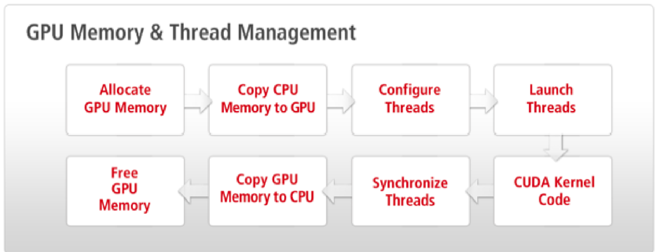
CUDALink offers a high-level interface to the GPU built on top of *Mathematica*'s development technologies. It allows users to execute code on their GPU with minimal effort. By fully integrating and automating the GPU's capabilities using *Mathematica,* users experience a more productive and efficient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

Automation of development project management

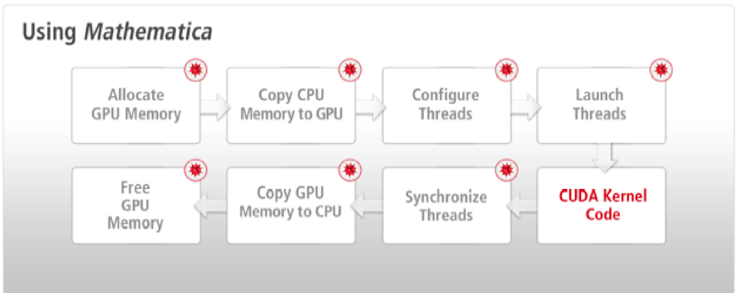
Unlike other development frameworks that require the user to manage project setup, platform dependencies, and device configuration, *CUDALink* makes the process transparent and automated.

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:

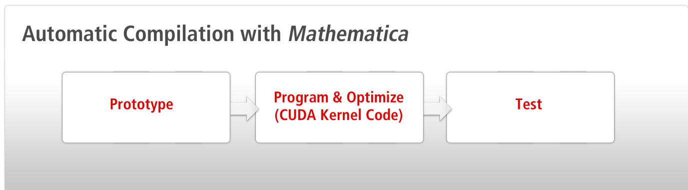


With *Mathematica*, memory and thread management is automatically handled for the user.



The *Mathematica* 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.

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



Integration with *Mathematica*'s built-in capabilities

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

With *Mathematica*'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 *Mathematica* 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 *Mathematica*'s *CUDALink* Applications.

Zero device configuration

Mathematica automatically finds, configures, and makes CUDA devices available to the users.

Multiple GPU Support

Through *Mathematica*'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 grid*Mathematica*.

Technologies Underlying *CUDALink*

GPU integration in *Mathematica* has only been possible due to advancement in system integration introduced in *Mathematica* 8. 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

Mathematica 8 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 *Mathematica* command (for native speed), or be integrated with an external application. Through the code generation mechanism, you can use *Mathematica* 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 call option in terms of the Black–Scholes equation is defined by:

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

with

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

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

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

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

```
d1 = (T (r + σ^2/2) + Log[S/X]) / (Sqrt[T] σ);
d2 = d1 - σ Sqrt[T];
BlackScholes = CDF[NormalDistribution[0, 1], d1] S -
  CDF[NormalDistribution[0, 1], d2] X ⅇ^(-r T)
```

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

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

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

The function can be used as any other *Mathematica* function. Here we call the above function:

```
cf[1.0, 1.0, 1.0, 1.0, 1.0]
```

```
0.678818
```

LibraryLink

LibraryLink allows you to load C functions as *Mathematica* functions. It is similar in purpose to *MathLink*, but, by running in the same process as the *Mathematica* 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 *Mathematica*'s symbolic capabilities, users can generate C programs within *Mathematica*. The following, for example, creates macros for common math constants:

```
<< SymbolicC`
```

These are all constants in the *Mathematica* system context. We use *Mathematica'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
#define Degree  0.017453292519943295
#define E  2.718281828459045
#define EulerGamma  0.5772156649015329
#define Glaisher  1.2824271291006226
#define GoldenRatio  1.618033988749895
#define Khinchin  2.6854520010653062
#define MachinePrecision  15.954589770191003
#define Pi  3.141592653589793
```

By representing the C program symbolically, you can manipulate it using standard *Mathematica* 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
#define degree  0.017453292519943295
#define e  2.718281828459045
#define eulergamma  0.5772156649015329
#define glaisher  1.2824271291006226
#define goldenratio  1.618033988749895
#define khinchin  2.6854520010653062
#define machineprecision  15.954589770191003
#define pi  3.141592653589793
```

C Compiler Invoking

Another *Mathematica* 8 innovation is the ability to call external C compilers from within *Mathematica*. 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"]
```

```
/home/abduld/.Mathematica/SystemFiles/LibraryResources/Linux/foo
```

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

```
Import["!" <> exe, "Text"]
```

```
Hello from CCompilerDriver.
```

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

Mathematica's *CUDALink*: Integrated GPU Programming

CUDALink is a built-in *Mathematica* application that provides a powerful interface for using CUDA within *Mathematica*. 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`"]
```

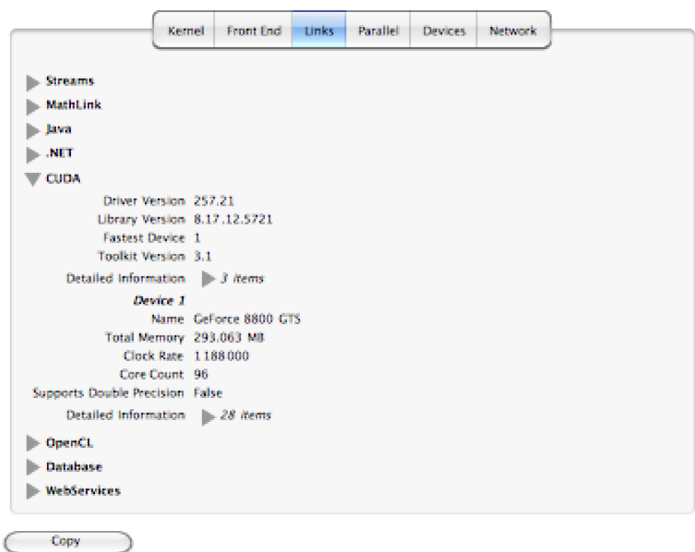
`CUDAQ` tells whether the current hardware and system configuration support *CUDALink*:

```
CUDAQ[]
```

```
True
```

`SystemInformation` gives information on the available GPU hardware:

```
SystemInformation[]
```



Example of a report generated by **`SystemInformation`**.

Integration with *Mathematica* Functions

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

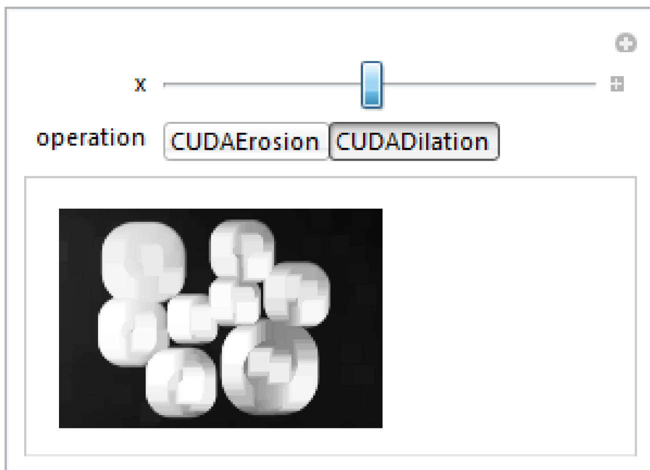
Manipulate: *Mathematica*'s automatic interface generator

Mathematica provides extensive built-in interface building functions. Users can customize controls using *Mathematica*'s highly declarative interface language.

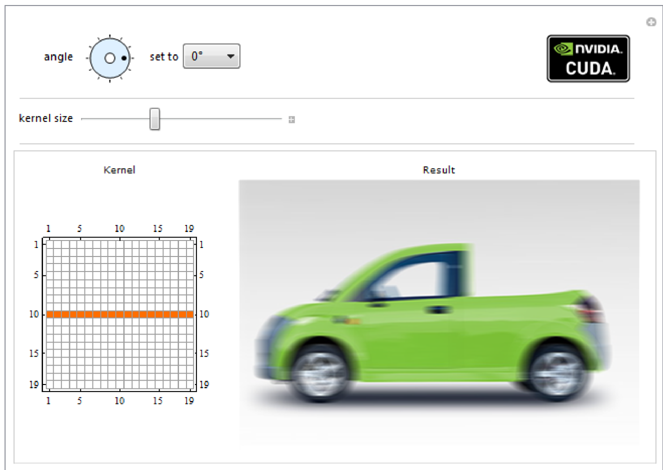
One fully automated interface generating function is **`Manipulate`**, which builds the interface by inspecting the possible values of variables. It then chooses the appropriate GUI widget based on the interpretation of the variable values.

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

```
Manipulate[operation[ , x],
 {x, 0, 9}, {operation, {CUDAErosion, 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

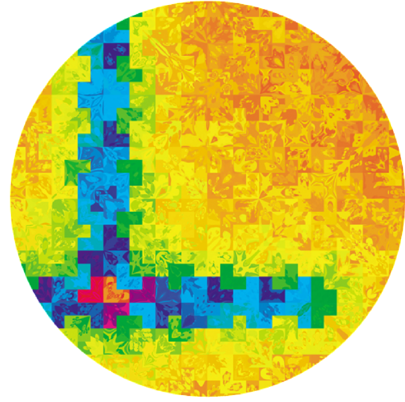
Mathematica natively supports hundreds of file formats and their subformats for importing and exporting. Supported formats include: common image formats (JPEG, PNG, TIFF, BMP, etc.), video formats (AVI, MOV, H264, etc.), audio formats (WAV, AU, AIFF, FLAC, etc.), medical imaging formats (DICOM), data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

Any supported data formats will be automatically converted to *Mathematica*'s unified data representation, or an *expression*, which can be used in all *Mathematica* 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 *Mathematica* expression. This can be directly used by *CUDALink* functions, such as **CUDAImageAdd**:



All outputs from *Mathematica* 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 *Mathematica* is straightforward. It begins with writing a CUDA kernel. Here, we will create a simple example that negates colors of a 3-channel image:

```
kernel = "
__global__ void cudaColorNegate(int
     *img, int *dim, int 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:

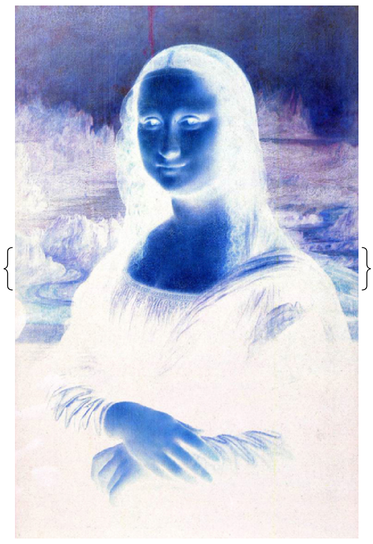
```
colorNegate = CUDAFunctionLoad[kernel, "cudaColorNegate",
   {{_Integer, "InputOutput"},
    {_Integer, "Input"}, _Integer}, {16, 16}];
```

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

```
img = 
```

```
;
```

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

{  }

System requirements

To utilize *Mathematica*'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.

Mathematica's *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;
spotPrices = RandomReal[{25.0, 35.0}, numberOfOptions];
strikePrices = RandomReal[{20.0, 40.0}, numberOfOptions];
expiration = RandomReal[{0.1, 10.0}, numberOfOptions];
interest = 0.08;
volatility = RandomReal[{0.10, 0.50}, numberOfOptions];
dividend = RandomReal[{0.2, 0.06}, numberOfOptions];
```

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

```
CUDAFinancialDerivative[{"AsianArithmetic", "Call"},
 { "StrikePrice" → strikePrices, "Expiration" → expiration},
 { "CurrentPrice" → spotPrices, "InterestRate" → interest,
  "Volatility" → volatility, "Dividend" → dividend}]
```

```
{8.34744, 1.18026, 9.53711, 5.39746, 2.2478, 4.94333, 0.859259,
 6.08291, 2.4044, 2.41929, 6.53313, 7.48516, 2.71696, 1.08229,
 7.50222, 0.790236, 0.816325, 1.28744, 0.953413, 0.131352,
 7.60693, 1.15648, 7.07213, 8.2441, 4.45964, 7.94849,
 2.22669, 1.17793, 10.1456, 0.263328, 4.12236, 4.99476}
```

Computing the price of Asian arithmetic call options corresponding to random data.

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 American 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:

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

with

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

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

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

This loads the above code into *Mathematica*:

```
CUDABlackScholes = CUDAFunctionLoad[code, "blackScholes",
   {{_Real}, {_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[{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 the call result:

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

This calls the function:

```
CUDABlackScholes[call, S, X, T, R, Q, V, numberOfOptions]
```

```
{CUDAMemory[<17988>, Double], CUDAMemory[<29045>, Double]}
```

This retrieves the CUDA memory back into *Mathematica*:

```
CUDAMemoryGet[call]
```

```
{-54.0707, -7.72175, -71.9744, -52.5466, -37.523,
 -54.7824, -66.2942, -38.6423, -44.1287, -57.2223,
 -95.4658, -89.4017, -7.29522, -100.874, -34.8353,
 -7.77607, -65.9444, -88.8973, -51.1116, -72.3198,
 -16.5077, -83.9489, -73.5695, -66.6177, -44.9486, -5.54228,
 -36.1501, -92.0541, -53.079, -97.6815, -34.6764, -38.1144}
```

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)
__global__ void binaryAssetOption(Real_t * call,
    Real_t * put, Real_t * S, Real_t * X, Real_t * T,
    Real_t * R, Real_t * D, Real_t * V, mint length) {
    int ii = threadIdx.x + blockIdx.x*blockDim.x;
    if (ii < length) {
        Real_t d1 = (log(S[ii]/X[ii]) + (R[ii] - D[ii] +
    0.5f * V[ii] * V[ii]) * T[ii])/(V[ii] * sqrt(T[ii]));
        call[ii] = S[ii] * exp(-D[ii] * T[ii]) * N(d1);
        put[ii] = S[ii] * exp(-D[ii]  * T[ii]) * N(-d1);
    }
}";
```

This loads the function into *Mathematica*:

```
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 = 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];
```

The call and put memory is 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[<17988>, Double], CUDAMemory[<29045>, Double]}
```

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

```
CUDAMemoryGet[call]
```

```
{11.4286, 17.9626, 9.18249, 6.12558, 16.0267, 31.328, 9.00912,
 18.1401, 15.0725, 22.843, 16.9324, 2.47273, 13.3358,
 12.6676, 27.3212, 21.1174, 3.89489, 21.0599, 24.4702,
 26.2096, 16.4893, 23.412, 17.8685, 5.92021, 16.5681,
 29.1562, 10.5462, 21.7121, 12.9029, 28.6162, 13.7667,
 29.118, 13.1107, 13.5246, 30.2435, 17.8356, 19.787, 20.3238,
 15.7308, 21.2625, 7.60982, 14.5222, 17.4748, 20.4389, 2.9495,
 6.98209, 18.9727, 33.6374, 14.5634, 14.1232, 12.1017,
 17.8766, 22.6046, 18.3085, 17.1108, 18.7836, 24.816, 13.8477,
 22.9212, 12.7896, 26.5338, 9.44261, 13.4497, 0.0381422}
```

```
CUDAMemoryGet[put]
```

```
{5.44932, 5.30771, 4.84476, 11.6373, 10.5256, 4.30614,
 15.0703, 9.17361, 7.14722, 5.68002, 7.53906, 21.5292, 12.769,
 7.99204, 1.85855, 3.09161, 13.1441, 5.38475, 6.38312, 7.681,
 8.36659, 2.28956, 7.79013, 8.42515, 6.09523, 3.58002,
 6.99927, 9.59092, 5.91341, 2.7295, 7.77723, 3.32061, 6.80901,
 6.62897, 4.22096, 8.75582, 4.88707, 10.5992, 5.17202,
 4.62305, 5.16322, 13.3896, 6.92575, 7.32805, 16.0042,
 13.4299, 5.21345, 2.79763, 11.8877, 6.21821, 13.8582,
 5.52745, 6.02657, 6.10139, 3.20472, 3.7745, 1.86951, 6.29216,
 2.6889, 7.95113, 7.44534, 15.6227, 7.80008, 20.6595}
```

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 pseudo- and quasi-random numbers. Here, we generate quasi-random numbers using the Halton sequence:

```
src = "
__device__ mint 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, unsigned int dim, unsigned int n) {
    const mint tx = threadIdx.x, bx =
    blockIdx.x, dx = blockDim.x;
    const mint index = tx + bx*dx;
    if (index >= n)
        return ;
    mint ii;
    double digit, rnd, idx, half;
    for (ii = 0,
    idx=index, rnd=0, digit=0; ii < dim; ii++) {
        half = 1.0/primes[ii];
        while (idx > DBL_EPSILON) {
            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 *Mathematica:*

```
CUDAHaltonSequence = CUDAFunctionLoad[src,
  "Halton", {{_Real, "Output"}, _Integer, _Integer}, 256]
```

```
CUDAFunction[<>, Halton, {{_Real, _, Output}, _Integer, _Integer}]
```

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]
```

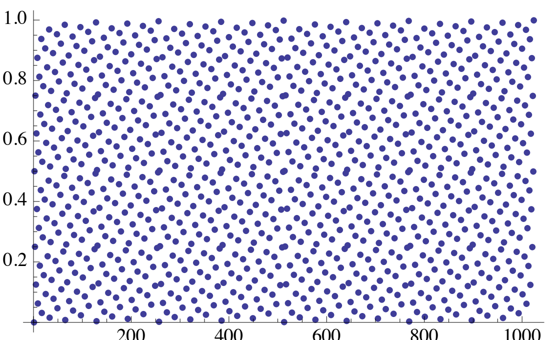
This calls the function:

```
CUDAHaltonSequence[mem, 1, 1024]
```

```
{CUDAMemory[<11521>, Double]}
```

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

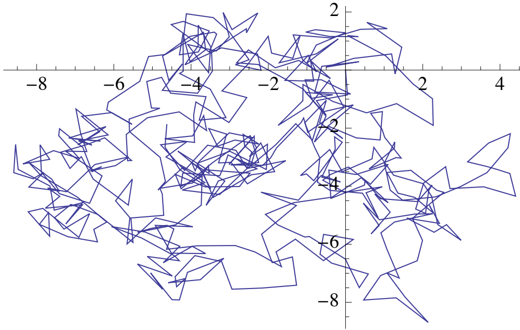
```
ListPlot[CUDAMemoryGet[mem]]
```



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 *Mathematica*'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 *Mathematica*:

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



Random walk simulation.

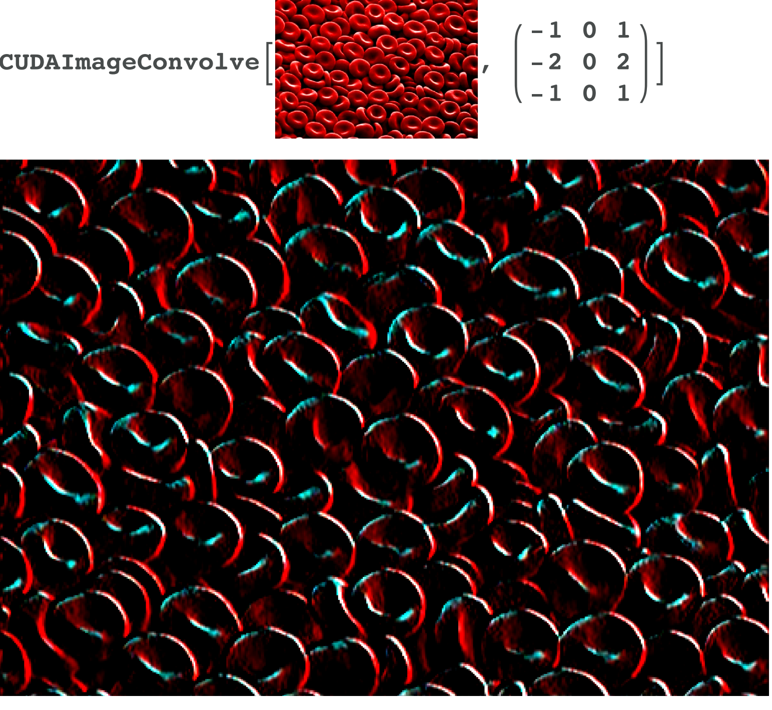
Image Processing

CUDALink's image processing capabilities can be classified into three categories. The first is convolution, which is optimized for CUDA. The second is morphology, which contains abilities such as erosion, dilation, opening, and closing. Finally, there are the binary operators. These are the image multiplication, division, subtraction, and addition. All operations work on either images or lists.

Image convolution

CUDALink's convolution is similar to *Mathematica*'s **ListConvolve** and **ImageConvolve** functions. It will operate on images, lists, or CUDA memory references, and it can use *Mathematica*'s built-in filters as the kernel.

```
CUDAImageConvolve[ , ( -1 0 1
                       -2 0 2
                       -1 0 1 )]
```

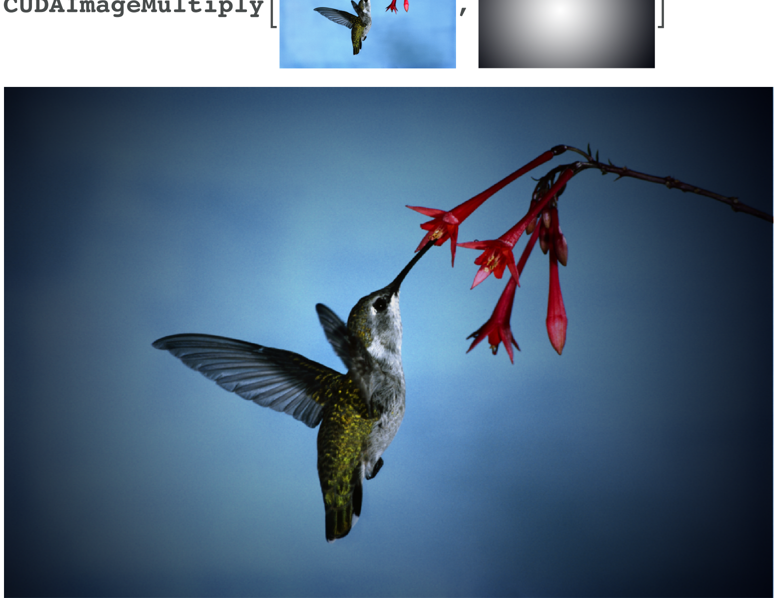


Convolving a microscopic image with a Sobel mask to detect edges.

Pixel operations

CUDALink supports simple pixel operations on one or two images, such as adding or multiplying pixel values from two images.

```
CUDAImageMultiply[ , ]
```



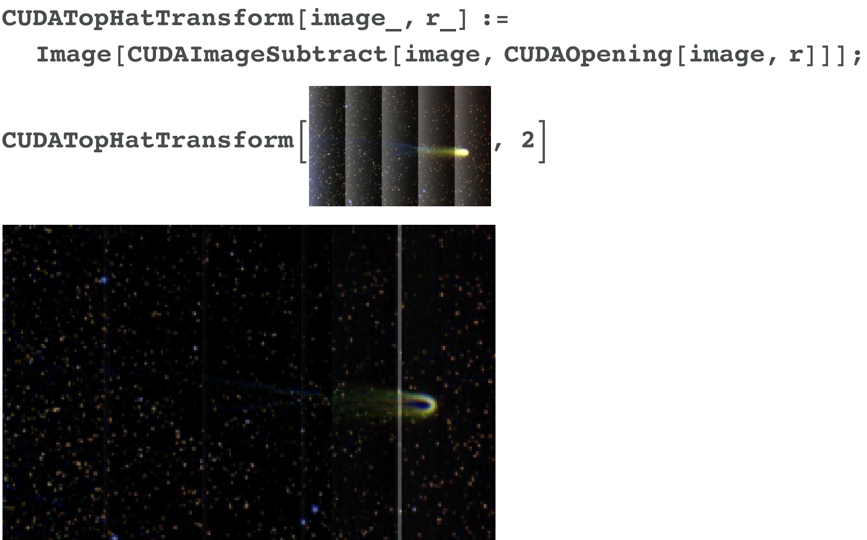
Multiplication of two images.

Morphological operations

CUDALink supports fundamental operations such as erosion, dilation, opening, and closing. **CUDAErosion**, **CUDADilation**, **CUDAOpening**, and **CUDAClosing** are equivalent to *Mathematica*'s built-in **Erosion**, **Dilation**, **Opening**, and **Closing** functions. More sophisticated operations can be built using these fundamental operations.

```
CUDATopHatTransform[image_, r_] :=
  Image[CUDAImageSubtract[image, CUDAOpening[image, r]]];

CUDATopHatTransform[ , 2]
```



Building non-elementary image processing operations using primitive ones.

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.

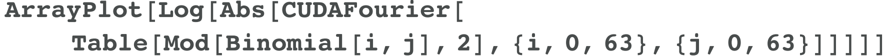
```
Nest[CUDADot[RandomReal[1, {100, 100}], #] &,
  RandomReal[1, {100}], 1000];
```

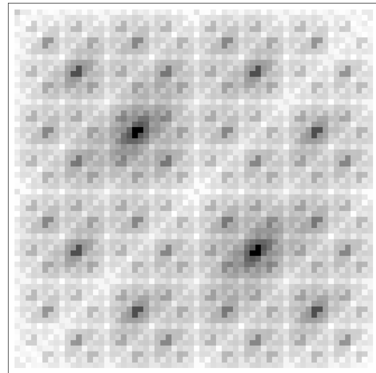
Perform Arnolidi-like linear algebra operation.

Fourier Analysis

The Fourier analysis capabilities of the *CUDALink* package 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}]]]]]
```

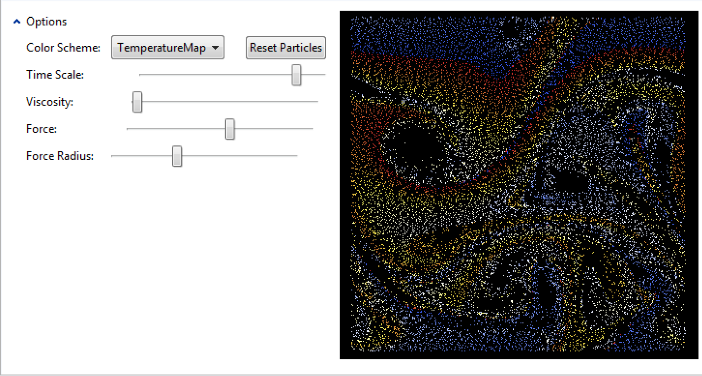




Find the logarithmic power spectrum of a dataset.

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:



Fluid simulation with multi-particles.

Volumetric Rendering

CUDALink includes functions to read and display volumetric data in 3D, with interactive interfaces for setting the transfer functions and other volume-rendering parameters.

Volumetric rendering of a medical image.

OpenCL Integration in *Mathematica*

Mathematica also includes the ability to use the GPU using OpenC via *OpenCLLink*. This 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)
__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 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]));
         p/Thetaower = 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 OpenCL function into *Mathematica*:

```
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 result:

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

This calls the function:

```
OpenCLOneTouchOption[call, put, S, X, T, R, Q, V, numberOfOptions]
```

```
{CUDAMemory[<6029>, Double], CUDAMemory[<15684>, 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., 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 *Mathematica*'s integrated platform design, all functionality is included without the need to buy and maintain multiple tools and add-on packages.

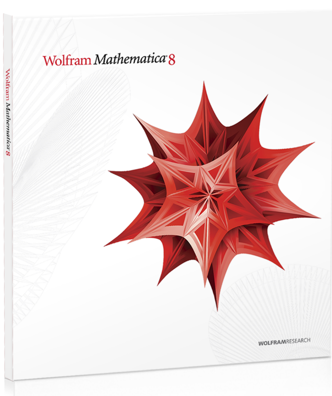
With its simplified development cycle, multicore computing, and built-in functions, *Mathematica*'s built-in *CUDALink* application provides a powerful high-level interface for GPU computing.

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/mathematica-purchase

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

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

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

Asia:
+81-(0)3-3518-2880
info@wolfram.co.jp

© 2011 Wolfram Research, Inc. *Mathematica* is a registered trademark and *Mathematica Player,* Wolfram *Workbench,* and grid*Mathematica* are trademarks of Wolfram Research, Inc. Wolfram|Alpha is a registered trademark and computational knowledge engine is a trademark of Wolfram Alpha LLC—A Wolfram Research Company. All other trademarks are the property of their respective owners. *Mathematica* is not associated with Mathematica Policy Research, Inc. or Mathtech, Inc. MKT3014 2173925 0111.hk