

---

WOLFRAM WHITE PAPER

---

# CUDA Programming with the Wolfram Language

**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. 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 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's CUDALink simplified the use of the GPU within the Wolfram Language 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 Wolfram System kernel.

By utilizing the Wolfram Language 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 the Wolfram Language 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 GPU 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 well, 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.

CUDALink alleviates much of the burden required to use CUDA from within the Wolfram Language. 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 the Wolfram Language

The Wolfram Language is a flexible programming language with a wide range of symbolic and numeric computational capabilities, high-quality visualizations, built-in application areas, and a range of immediate deployment options. Combined with integration of dynamic libraries, automatic interface construction, and C code generation, the Wolfram Language is the most sophisticated build-to-deploy environment on the market today.

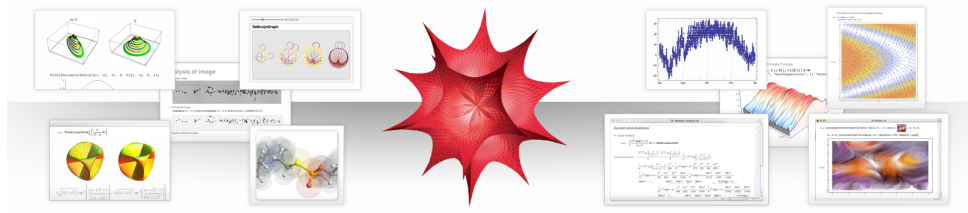
The Wolfram Language provides the ability to program the GPU. For developers, this integration means native access to the Wolfram Language's computing abilities—creating hybrid algorithms that combine the CPU and the GPU. Following are some key features of the Wolfram Language.

### Full-Featured, Unified Development Environment

Through its unique interface and integrated features for computation, development, and deployment, the Wolfram Language 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 the Wolfram Language is the foundational idea that everything—data, programs, formulas, graphics, documents—can be represented as symbolic entities, called *expressions*. This unified representation makes the Wolfram Language extremely flexible, streamlined, and consistent.



## Multiparadigm Programming Language

The Wolfram Language is a highly declarative functional language that also enables you to use 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, the Wolfram Language's flexibility greatly reduces the cost of entry for new users.

## Symbolic-Numeric Hybrid System

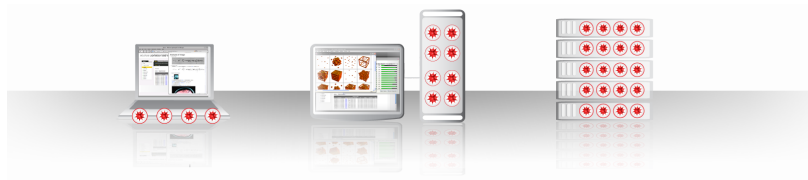
The principle behind the Wolfram Language is full integration of symbolic and numeric computing capabilities. Through its full automation and preprocessing mechanisms, users reap the power of a hybrid computing system without needing knowledge of specific methodologies and algorithms.

## Scientific and Technical Area Coverage

The Wolfram Language 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

The Wolfram Language 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

The Wolfram Language natively supports hundreds of formats for importing and exporting, as well as real-time access to data from the Wolfram Knowledgebase. It also provides APIs for accessing many programming languages and databases, such as C/C++, Java, .NET, MySQL, and Oracle.

## Platform-Independent Deployment Options

Through its interactive documents, Wolfram CDF Player, browser plugins, and cloud connectivity, the Wolfram Language 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 gridMathematica allows Wolfram Language programs to be parallelized on many machines in cluster or grid configuration.

## CUDA Integration in the Wolfram Language

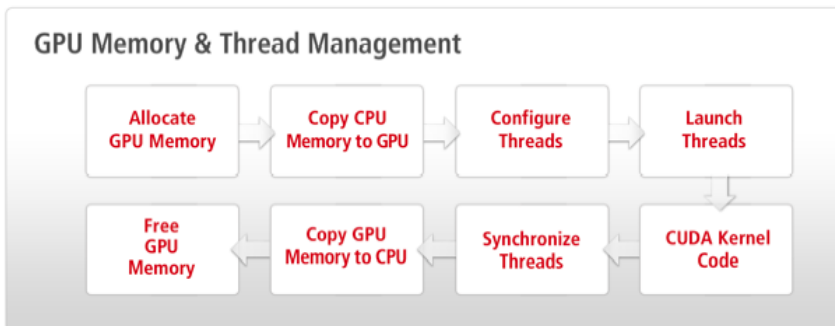
CUDALink offers a high-level interface to the GPU built on top of the Wolfram Language's development technologies. It allows users to execute code on their GPU with minimal effort. By fully integrating and automating the GPU's capabilities using the Wolfram Language, users experience a more productive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

### Automation of development project management

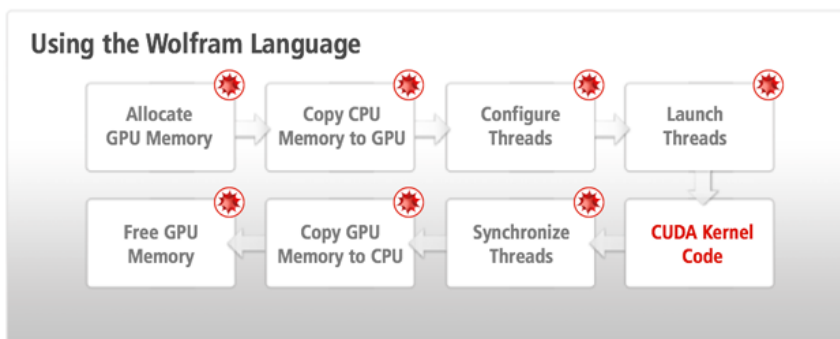
Unlike other development frameworks that require the user to manage project setup, platform dependencies, and device configuration, CUDALink makes the process transparent and automated.

### 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.

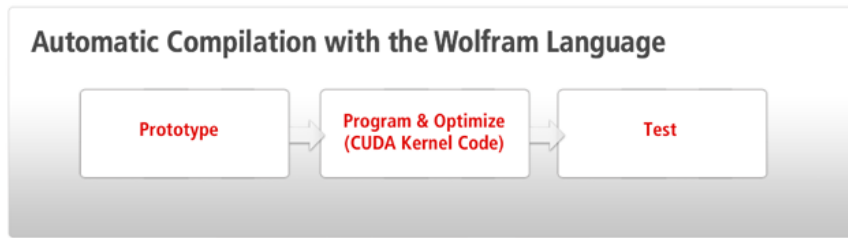


With the Wolfram Language, memory and thread management is automatically handled for the user.



The Wolfram Language's 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.

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



Integration with the Wolfram Language's built-in capabilities

CUDA integration provides full access to the Wolfram Language's built-in functions.

With the Wolfram Language'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 the Wolfram Language 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 The Wolfram Language's CUDALink Applications.

Zero device configuration

The Wolfram Language automatically finds, configures, and makes CUDA devices available to the users.

Multiple GPU support

Through the Wolfram Language'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`.

## Technologies Underlying CUDALink

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

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

To motivate the C code generation feature, we will use a simple mathematical function:

$$f(x) = \sin(x) + x^2 - \frac{1}{x + 1}$$

The function can be defined in the Wolfram Language as follows:

```
func = Sin[x] + x2 -  $\frac{1}{1+x}$ ;
```

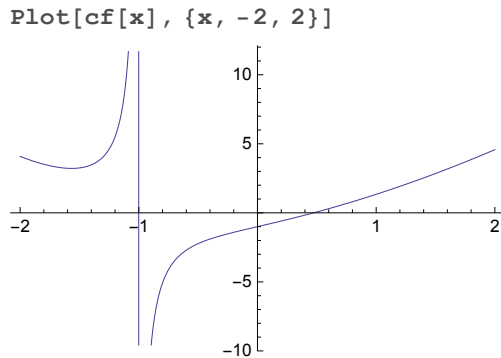
The following command generates the C code, compiles it, and links it back into the Wolfram Language to provide native speed:

```
cf = Compile[{{x, _Real}}, func, CompilationOptions →  
  {"InlineExternalDefinitions" → True}, CompilationTarget → "C"];
```

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

```
cf[1.0]  
1.34147
```

Here, we can call the function within Plot:



## LibraryLink

LibraryLink allows you to load C functions as Wolfram Language functions. It is similar in purpose to WSTP (Wolfram Symbolic Transfer Protocol) but, by running in the same process as the Wolfram System kernel, it avoids the memory transfer cost associated with WSTP. 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 the Wolfram Language's symbolic capabilities, users can generate C programs within the Wolfram Language. The following, for example, creates macros for common math constants:

```
<< SymbolicC`
```

These are all constants in the Wolfram System context. We use the Wolfram Language'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 Wolfram Language 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 Wolfram Language innovation is the ability to call external C compilers from within the Wolfram Language. The following compiles a simple C program into an executable:

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

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

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

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

# The Wolfram Language's CUDALink: Integrated GPU Programming

CUDALink is a built-in Wolfram Language application that provides a powerful interface for using CUDA within the Wolfram Language. 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[]
```

Kernel	Front End	Links	Parallel	Devices	Network
Streams					
MathLink					
Java					
.NET					
CUDA					
Driver Version				261.28	
Library Version				8.17.12.6128	
Fastest Device				1	
Toolkit Version				3.1	
Detailed Information				3 items	
Device 1					
Name				Quadro FX 3800M	
Total Memory				999.563 MB	
Clock Rate				1650000	
Core Count				128	
Supports Double Precision				False	
Detailed Information				28 items	
OpenCL					
Database					
WebServices					

Copy

Example of a report generated by `SystemInformation`.

## Integration with Wolfram Language Functions

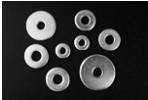
CUDALink integrates with existing Wolfram Language functions such as its import/export facilities, functional language, and interface building. This allows you to build deployable programs in the Wolfram Language with minimal disruption to the GPU task. This section showcases how you can build interfaces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

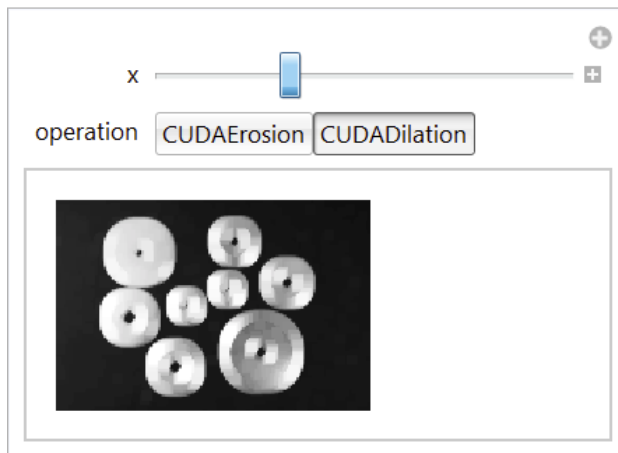
## Manipulate: The Wolfram Language's automatic interface generator

The Wolfram Language provides extensive built-in interface-building functions.

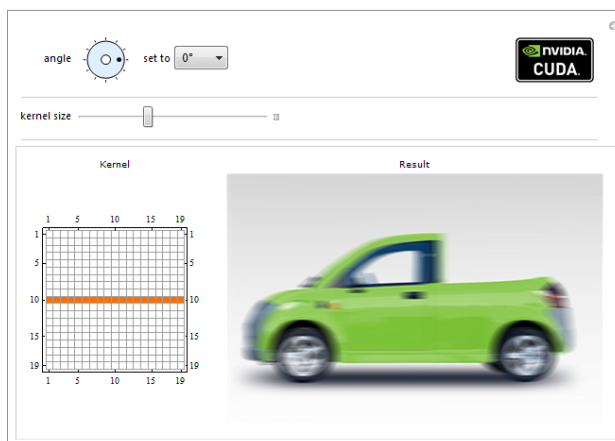
One fully automated interface-generating function is `Manipulate`, which builds the interface by inspecting the possible values of variables. It then chooses the appropriate GUI widget based on the interpretation of the variable values.

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 angles) and performs a convolution on the image on the right.



Example of a user interface built with `Manipulate`.

## Support for import and export

The Wolfram Language 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.

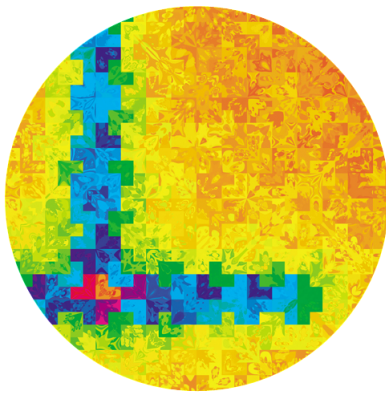
Any supported data formats will be automatically converted to the Wolfram Language's unified data representation, or an *expression*, which can be used in all Wolfram Language functions, including CUDALink functions.

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 Language expression. This can be directly used by CUDALink functions, such as `CUDAImageAdd`:

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



All outputs from Wolfram Language 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 preceding output into PNG format:

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

## CUDALink Programming

Programming the GPU in the Wolfram Language is straightforward. It begins with writing a CUDA kernel. Here, we will create a simple example that negates the 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. The Wolfram Language automatically compiles the kernel function and loads it as a Wolfram Language function. Now you can apply this new CUDA function to an image:

```
img = ;
```

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



## System Requirements

To utilize the Wolfram Language'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.

## The Wolfram Language'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.

### Complex Dynamics

CUDALink enables you to easily investigate computationally intensive complex dynamics structures. Following is an example for generating a Julia set.

The Julia set is a generalization of the Mandelbrot set. This implements the CUDA kernel:

```
code = "  
__global__ void julia_kernel(Real_t * set,  
    int width, int height, Real_t cx, Real_t cy) {  
    int xIndex = threadIdx.x + blockIdx.x*blockDim.x;  
    int yIndex = threadIdx.y + blockIdx.y*blockDim.y;  
    int ii;  
  
    Real_t x = ZOOM_LEVEL*(width/2 - xIndex);  
    Real_t y = ZOOM_LEVEL*(height/2 - yIndex);  
    Real_t tmp;  
    Real_t c;  
  
    if (xIndex < width && yIndex < height) {  
        for (ii = 0; ii <  
            MAX_ITERATIONS && x*x + y*y < BAILOUT; ii++) {  
            tmp = x*x - y*y + cx;  
            y = 2*x*y + cy;  
            x = tmp;  
        }  
        c =  
        logf(static_cast<Real_t>(0.1) + sqrtf(x*x + y*y));  
        set[xIndex + yIndex*width] = c;  
    }  
}  
";
```

The width and height are set. Since the set is computed, the memory need not be set—that is, only memory allocation is needed:

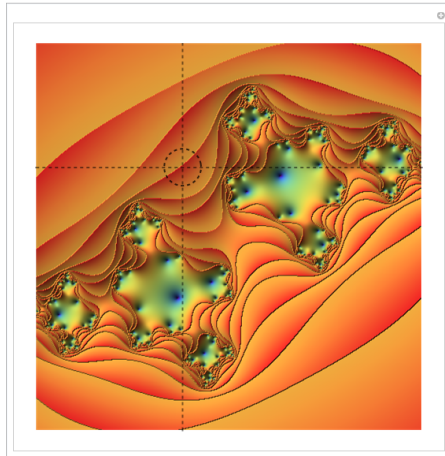
```
{width, height} = {512, 512};  
jset = CUDAMemoryAllocate[Real, {height, width}];
```

This loads `CUDAFunction`. Macros are used to allow the compiler to optimize the code—doing things like loop unrolling:

```
JuliaCalculate = CUDAFunctionLoad[code, "julia_kernel",  
  {{_Real, _, "Output"}, _Integer, _Integer, _Real, _Real},  
  {16, 16}, "Defines" → {"MAX_ITERATIONS" → 10,  
    "ZOOM_LEVEL" → "0.0050", "BAILOUT" → "4.0"}];
```

This creates an interface using `Manipulate` and `ReliefPlot` where you can adjust the value of `c` interactively:

```
Manipulate[  
  JuliaCalculate[jset, width,  
    height, c[[1]], c[[2]], {width, height}];  
  ReliefPlot[Reverse@CUDAMemoryGet[jset], ColorFunction → "Rainbow",  
    DataRange → {{-2.0, 2.0}, {-2.0, 2.0}}, ImageSize → 512,  
    Frame → None, Epilog → {Opacity[.5], Dashed, Thick, Line[  
      {{{c[[1]], -2}, {c[[1]], 2}}, {{-2, c[[2]]}, {2, c[[2]]}}]}],  
    {{c, {0, 1}}, {-2, -2}, {2, 2}, Locator, Appearance →  
      Graphics[{Thick, Dashed, Opacity[.75], Circle[]}, ImageSize → 50]}]
```



Interactive computation and rendering of a Julia set.

## Random Number Generators

One of the difficult problems when parallelizing algorithms is generating good random numbers. CUDALink offers many examples of how to generate both pseudo- and quasi-random numbers. Here, we generate quasi-random 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 the Wolfram Language:

```
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:

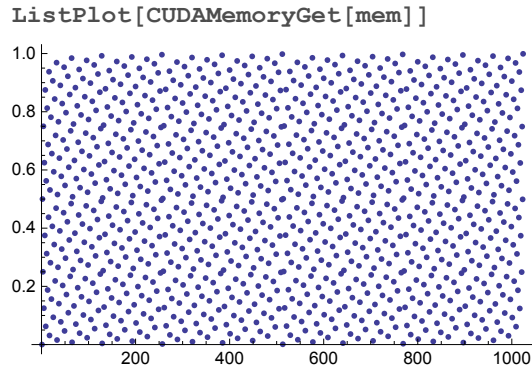
```
mem = CUDAMemoryAllocate[Real, {1024}]  
CUDAMemory[<11521>, Double]
```

This calls the function:

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



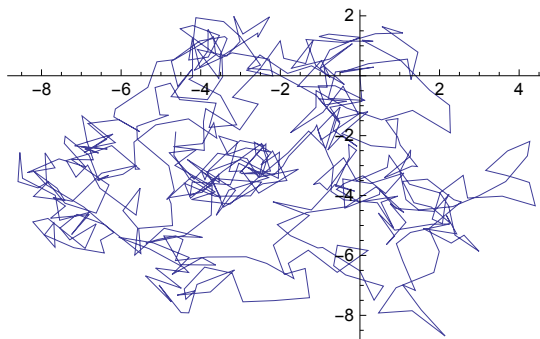
You can use the Wolfram Language's extensive visualization support to visualize the result. Here we plot the data:



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. With this approach, users can maximize the Wolfram Language'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 the Wolfram Language:

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




Random walk simulation.

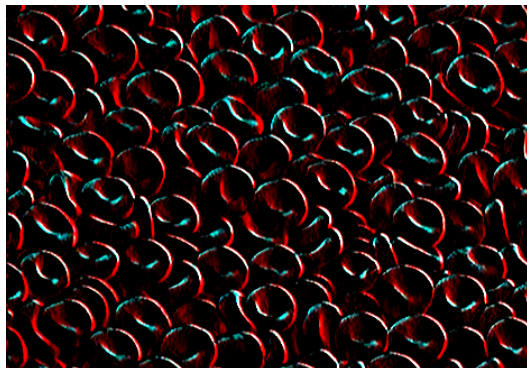
## Image Processing

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 image multiplication, division, subtraction, and addition. All operations work on either images or lists.

## Image convolution

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

```
CUDAImageConvolve[,  $\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$ ]
```

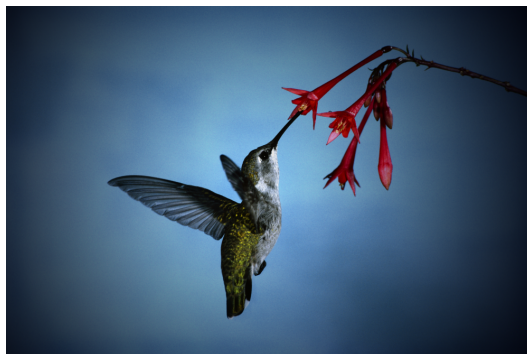


Convolving a microscopic image with a Sobel mask to detect edges.

## 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 the Wolfram Language's built-in `Erosion`, `Dilation`, `Opening`, and `Closing` functions. More sophisticated operations can be built using these fundamental operations.

## Linear Algebra

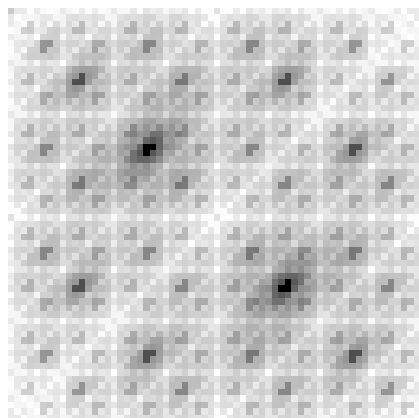
You can perform various linear algebra functions with CUDALink, such as matrix-matrix and matrix-vector multiplication, finding minimum and maximum elements, and transposing matrices:

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

## 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:

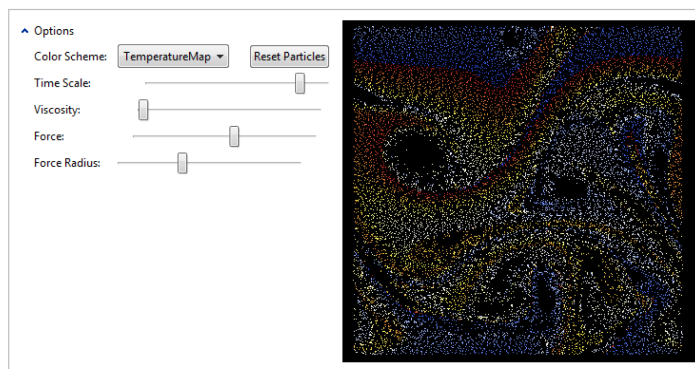
```
ArrayPlot[Log[Abs[CUDAFourier[Table[  
  Mod[Binomial[i, j], 2], {i, 0, 63}, {j, 0, 63}]]]], Frame -> False]
```



Finding 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.

## 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}
```

## OpenCL Integration in the Wolfram Language

The Wolfram Language 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)
#ifdef 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 the Wolfram Language:

```
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}
```

## OpenCL Application: Many-Body Physical Systems

The N-body simulation is a classic Newtonian problem. This implements it in OpenCL:

```
srcf = FileNameJoin[{$OpenCLLinkPath, "SupportFiles", "NBody.cl"}];
```

This loads OpenCLFunction:

```
NBody = OpenCLFunctionLoad[{srcf}, "nbody_sim",
  {"Float[4]", _, "Input"}, {"Float[4]", _, "Input"},
  _Integer, "Float", "Float", {"Local", "Float"},
  {"Float[4]", _, "Output"}, {"Float[4]", _, "Output"}}, 256]
OpenCLFunction[<>, nbody_sim,
  {{Float[4], _, Input}, {Float[4], _, Input}, _Integer, Float, Float,
  {Local, Float}, {Float[4], _, Output}, {Float[4], _, Output}}]
```

The number of particles, time step, and epsilon distance are chosen:

```
numParticles = 1024;
deltaT = 0.05;
epsSqrt = 50.0;
```

This sets the input and output memories:

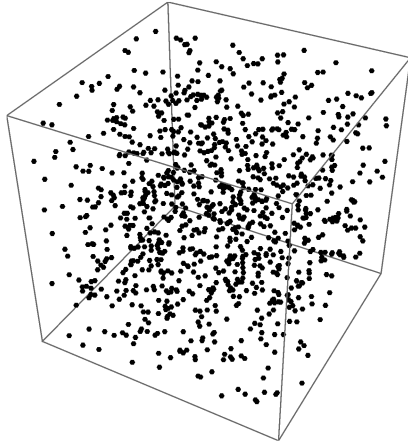
```
pos =
  OpenCLMemoryLoad[RandomReal[512, {numParticles, 4}], "Float[4]"];
vel = OpenCLMemoryLoad[RandomReal[1, {numParticles, 4}], "Float[4]"];
newPos = OpenCLMemoryAllocate["Float[4]", {numParticles, 4}];
newVel = OpenCLMemoryAllocate["Float[4]", {numParticles, 4}];
```

This calls the `NBody` function:

```
NBody[pos, vel, numParticles,  
      deltaT, epsSqrt, 256 * 4, newPos, newVel, 1024];  
NBody[newPos, newVel, numParticles, deltaT,  
      epsSqrt, 256 * 4, pos, vel, 1024];
```

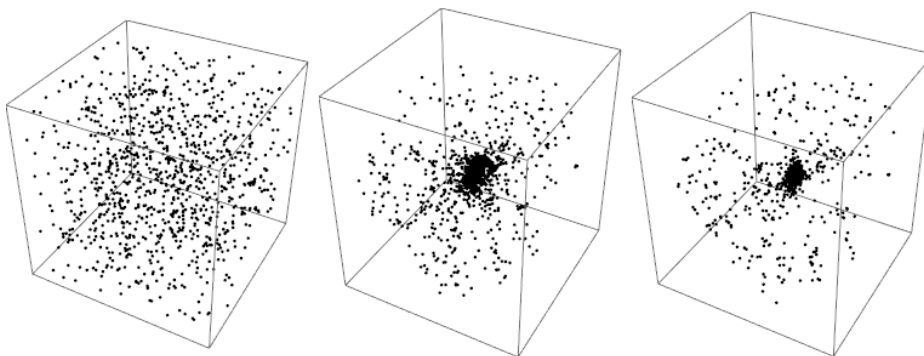
This plots the body points:

```
Graphics3D[Point[Take[#, 3] & /@OpenCLMemoryGet[pos]]]
```



This shows the result as a `Dynamic`:

```
Graphics3D[Point[  
  Dynamic[Refresh[  
    NBody[pos, vel, numParticles,  
          deltaT, epsSqrt, 256 * 4, newPos, newVel, 1024];  
    NBody[newPos, newVel, numParticles, deltaT,  
          epsSqrt, 256 * 4, pos, vel, 1024];  
    Take[#, 3] & /@OpenCLMemoryGet[pos], UpdateInterval -> 0]]]]
```



Real-time animation of the N-body simulation.

## Summary

Due to the Wolfram Language'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, the Wolfram Language'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/how-to-buy](http://www.wolfram.com/mathematica/how-to-buy)

## Recommended Next Steps



**Try the Wolfram Language in Mathematica for free:**

[www.wolfram.com/mathematica/trial](http://www.wolfram.com/mathematica/trial)

**Schedule a technical demo:**

[www.wolfram.com/mathematica-demo](http://www.wolfram.com/mathematica-demo)

**Learn more about CUDA programming in the Wolfram Language:**

US and Canada:

1-800-WOLFRAM (965-3726)

[info@wolfram.com](mailto:info@wolfram.com)

Europe:

+44-(0)1993-883400

[info@wolfram.co.uk](mailto:info@wolfram.co.uk)

Outside US and Canada (except Europe and Asia):

+1-217-398-0700

[info@wolfram.com](mailto:info@wolfram.com)

Asia:

+81-(0)3-3518-2880

[info@wolfram.co.jp](mailto:info@wolfram.co.jp)

© Wolfram Research, Inc. Trademarks: Mathematica, Wolfram Language, Wolfram Workbench, Wolfram Knowledgebase, Wolfram CDF Player, gridMathematica, Wolfram Symbolic Transfer Protocol, Wolfram System. MKT 3025 TCS-261 0204.BW