Mathematica for GPU Programming

Ulises Cervantes-Pimentel
Senior Kernel Developer, Scientific Visualization Group, Wolfram Research, Inc.

Abdul Dakkak
Kernel Developer, Scientific Visualization Group, Wolfram Research, Inc.

Mathematica is widely used in scientific, engineering, mathematical fields and education. In this session, new tools for general GPU programming in the next release of *Mathematica* are presented. These tools build on top of *Mathematica*'s technology which provides a simple, yet powerful, interface to the large base of compiling tools. Applications of CUDA and OpenCL from within *Mathematica* will be presented. These examples will provide a general overview of the powerful development environment for GPU programming that *Mathematica* can offer not just for researchers but for anybody with basic knowledge of *Mathematica* and GPU programming.

Introduction

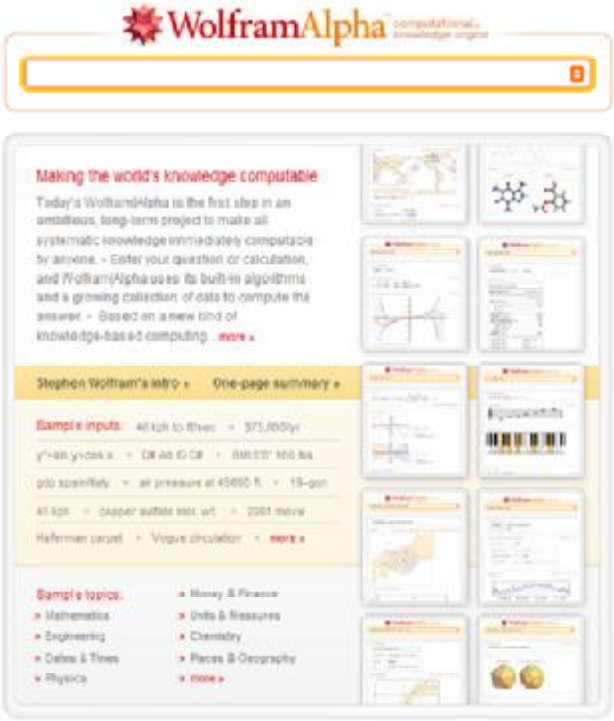
Thousands of organizations, including all of the Fortune 50 companies and top 200 universities worldwide, use *Mathematica* to help them maintain their innovative edge.

Mathematica offers an intuitive environment—even featuring built-in ready-to-use examples for common application areas, such as image processing, medical imaging, statistics, and finance—that makes CUDA programming a breeze, even if you've never used *Mathematica* before.

Considering CUDA's advanced technology, you may expect its programming to be enormously complicated. Enter *Mathematica*, the easiest way to program for CUDA and unlock GPU performance potential. Unlike programming in C or developing CUDA wrapper code, now you don't have to be a programming wizard to use CUDA.

And if you have used *Mathematica*, you'll be amazed by the massive boost in computational power, as well as application performance, enhancing speed by factors easily exceeding 100.

Wolfram|Alpha was **Prototyped**, **Developed** and **Deployed** entirely using *Mathematica* Technologies [web]



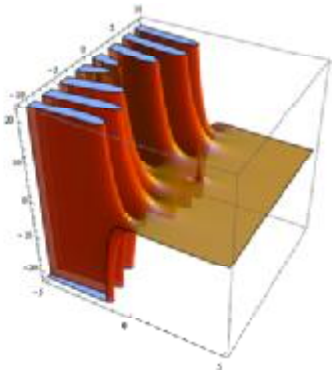
Slide 1 of 15

Overview

- A Brief Introduction to *Mathematica*
  - *Mathematica* features highlights
  - Built-in high-performance computing
  - Documentation and Learning
  - Examples
- Benefits of CUDA Integration in *Mathematica*
  - Simplified Development Cycle
  - Performance Improvement
- *Mathematica CUDALink*: Integrated GPU Programming
  - System Requirements
  - Getting Started with *CUDALink*
  - Accessing System Information
  - Retargetable Code Generation
  - Integration with *Mathematica* Functions
  - OpenCL Compatibility
- *Mathematica CUDALink* Applications
  - Perlin Noise
  - Image Processing
  - Video Processing
  - Linear Algebra
  - Fourier Analysis
- Pricing and Licensing Information
- Summary
- *Mathematica* Technology Conference

Slide 2 of 15

A Brief Introduction to *Mathematica*



Mathematica has been one of the most powerful languages for technical computing for more than 20 years. Wolfram Research introduces fully integrated GPU programming capabilities in *Mathematica*, which brings a whole new meaning to high-performance computing. For CUDA developers, the new integration means unlimited access to *Mathematica*'s vast computing abilities.

Mathematica is a sophisticated 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. With direct integration of dynamic libraries, instant interface construction, and automatic C code generation and linking, *Mathematica* provides the most sophisticated build-to-deploy environment in the market.

Slide 3 of 15

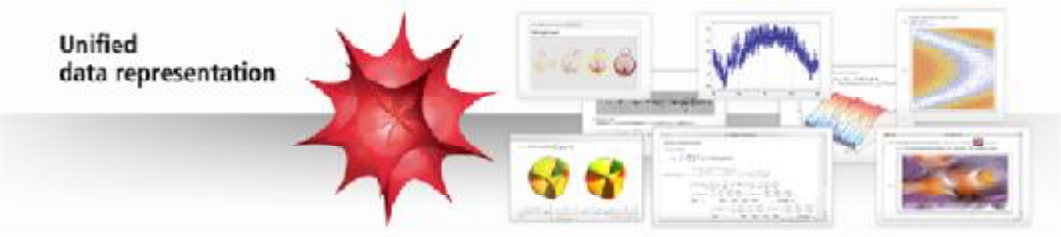
Some highlights of *Mathematica*'s features include:

Full-featured, unified development environment

Through its unique interface and integrated features for computation, development, and deployment, *Mathematica* provides a streamlined workflow.

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.



Multi-paradigm 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.

Unique symbolic-numeric hybrid system

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

Extensive 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 system, which enables users to break the barriers between specialized areas and explore new possibilities.

Easy data access and connectivity

Mathematica natively supports hundreds of formats for importing and exporting, as well as unlimited access to data from Wolfram|Alpha®, Wolfram Research's computational knowledge engine™. It also provides APIs for accessing common database and programming languages, such as SQL, C/C++, and .NET.

Platform-dependent deployment options

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

Slide 4 of 15

Built-in high-performance computing

Mathematica fully supports multi-threaded, multicore processors without extra packages. Many functions automatically utilize the power of multicore processors, and built-in generic parallel functions make high-performance programming a simple process.

gridMathematica increases the power of Mathematica by adding extra computation kernels and automated network - distribution tools. Extending Mathematica' s built-in parallelization capabilities, gridMathematica runs more tasks in parallel, over more CPUs, for faster execution.

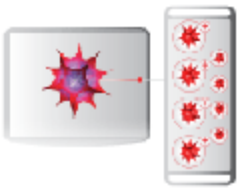
Whether you distribute tasks over local or remote CPUs or both, process coordination and management is completely automated. Appropriate parallel tasks run faster with no need for code changes. Choose the grid solution that' s best for you:

grid*Mathematica* [more...]

Scaling up your parallel *Mathematica* installation. The same code scaled to a larger grid.

grid*Mathematica* Local = *Mathematica* + 4 subkernels

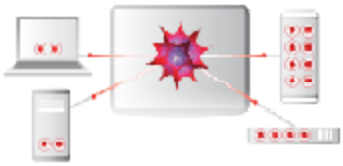
Targeted for development machines



grid*Mathematica* Server = *Mathematica* + 16 x subkernels

Targeted for department cluster

Works with existing batch schedulers (Windows CCS, Windows HPC, Platform LSF, Altair PBS Pro, Sun GridEngine, ...)



Slide 5 of 15

Documentation and Learning

Documentation

- Over 300 guide pages or "concept maps" [root guide page] [sample guide page]
- Over 4000 reference pages with some 50,000+ carefully developed examples [sample ref page]
- Over 600 tutorials, including updated sections from the *Mathematica* book [sample tutorial]
- Many how-to task oriented documents [howto]
- Over 150,000 cross links between document elements
- On-web and in-product documentation [reference.wolfram.com]
- In-product and on-web linguistic-aware search system

Other

- Free live seminars [seminar list]
- Paid live courses [course list]
- Over 4500 online demonstrations [demonstrations]
- Large number of reference books
- Many more online resources [web resources]

Mathematica Examples

```
$Line = 0;
```

Manipulate Anything

Automatically create an interface for manipulating any expression:

```
Clear[a, b, x, n]
Manipulate[n!, {n, 1, 1000, 1}]
Manipulate[Expand[(a + b)^n], {n, 1, 100, 1}]
Manipulate[ContourPlot3D[x^2 + y^2 + a z^3 == 1, {x, -2, 2},
  {y, -2, 2}, {z, -2, 2}, Mesh → None], {a, -2, 2}]
```

Manipulate any number of variables with discrete or continuous domains:

```
Manipulate[Plot[Sin[a x + b], {x, 0, 6}], {{a, 2, "Frequency"}, 1, 4},
 {{b, 0, "Phase"}, 0, 10}]
Manipulate[Plot[f[x - x0], {x, 0, 2 Pi}], {f, {Sin, Cos, Tan, Cot}},
 {x0, 0, 2 Pi}]
```

Specify custom controls:

```
Manipulate[Graphics[Line[{{0, 0}, p}], PlotRange → 2],
 {{p, {1, 1}}, Locator}]
A = ( -1.1  0.9
      -1.4  0.3 );
Manipulate[ParametricPlot[Evaluate[MatrixExp[A t, #] & /@ pt],
  {t, 0, 10}, PlotRange → 5],
 {{pt, {{2, 0}, {0, 1}, {-3, 0}}}, Locator, LocatorAutoCreate → True},
 SaveDefinitions → True
]
```

Slide 7 of 15

Dynamic and Control Objects

Dynamic variables maintain dependencies and update dynamically:

```
e = 1;
e
e = 2.0;
e
Dynamic[e]
e = 3
e = Integrate[1/(1 - x^3), x]
e = Plot[Sin[x], {x, 0, 2 Pi}]
e = .
```

Controls repeatedly set (or control) variables:

```
Slider[.5]
Slider[Dynamic[e]]
e = .5
{Slider[Dynamic[e]], Dynamic[e]}
```

You can localize this dependence by introducing dynamic module:

```
DynamicModule[{e}, {Slider[Dynamic[e]], Dynamic[e]}]
DynamicModule[{e}, {Slider[Dynamic[e], {1, 10}],
  Dynamic[Plot[Sin[e x], {x, 0, 2 Pi}]]}]
```

Apart from this sliders, checkboxes etc are just like any other expression:

```
Expand[(1 + Slider[Dynamic[e]])^3]
```

Also just about anything can be made dynamic:

```
{Slider[Dynamic[n], {5, 200}],
 Style["Wolfram", FontSize → Dynamic[n]]}
```

This looks slightly long, but compare to other ways of doing it [web]:

```
makeHand[fl_, bl_, fw_, bw_] :=
  Polygon[{{-bw, -bl}, {bw, -bl}, {fw, fl}, {0, fl + 8 fw},
     {-fw, fl}} / 9];
hourHand = makeHand[5, 5 / 3, .1, .3];
minuteHand = makeHand[7, 7 / 3, .1, .3];
secondHand = {Red, EdgeForm[Black],
   makeHand[7, 7 / 3, .1 / 2, .3 / 2]};
Graphics[{
  {Thickness[.03], Circle[]},
  {Thickness[.003],
   Table[Line[{.9 {Cos[a], Sin[a]}, .95 {Cos[a], Sin[a]}}],
    {a, 0, 2 π, 2 π / 60}]},
  {Thickness[.01],
   Table[Line[{.9 {Cos[a], Sin[a]}, .95 {Cos[a], Sin[a]}}],
    {a, 0, 2 π, 2 π / 12}]},
  Style[Table[Text[i, .77 {Cos[-i π / 6 + π / 2], Sin[-i π / 6 + π / 2]}],
    {i, 1, 12}], FontFamily → "Helvetica", FontSize → 24],
  Rotate[hourHand,
   Dynamic[Refresh[-6 Mod[AbsoluteTime[] / 360, 60] °,
     UpdateInterval → 60]], {0, 0}],
  Rotate[minuteHand,
   Dynamic[Refresh[-6 Mod[AbsoluteTime[] / 60, 60] °,
     UpdateInterval → 1]], {0, 0}],
  Rotate[secondHand,
   Dynamic[Refresh[-6 Round[Mod[AbsoluteTime[], 60]] °,
     UpdateInterval → .25]], {0, 0}]
 }, ImageSize → Small]
```

Viewing and Annotation

Use different viewers to package information compactly:

```
TabView[Table[Plot[BesselJ[n, x], {x, 0, 10}], {n, 5}]]
```

```
TabView[
 Table[TraditionalForm@BesselJ[n, x] →
   Plot[BesselJ[n, x], {x, 0, 10}], {n, 5}]]
```

```
SlideView[Table[Plot[BesselJ[n, x], {x, 0, 10}], {n, 5}]]
```

Use mouseover etc to provide alternate information:

```
Mouseover[a, b]
```

```
data = RandomReal[1, {100, 2}];
Mouseover[
 ListPlot[data],
 ListLinePlot[data[[Last@FindShortestTour[data]]], Mesh → All]
]
```

Use tooltip to provide additional information:

```
Tooltip[a, b]
```

```
Grid@
 Table[Tooltip[ParametricPlot[{Sin[n t], Sin[m t]}, {t, 0, 2 Pi},
    ImageSize → 70, Frame → True, FrameTicks → None, Axes → False],
   {Sin[n t], Sin[m t]}], {m, 3}, {n, 3}]
```

Use monitor to temporarily provide a view of information:

```
Monitor[Do[i, {i, 10^6}], i]
```

```
Monitor[
 NDSolve[{∂t,t u[t, x] == ∂x,x u[t, x] + Sin[u[t, x]] , u[0, x] == ℯ^-x^2,
   u^(1,0)[0, x] == 0, u[t, -10] == u[t, 10]}, u, {t, 0, 10},
  {x, -10, 10}, StepMonitor :↦ (sol = u[t, x]; time = t)],
 Plot[sol, {x, -10, 10}, PlotRange → {0, 8}, PlotLabel → time]]
```

Slide 9 of 15

Data Handling and Data Sources – Overview

- Import/Export framework that support addressing of elements or parts of files
- Large number of formats and area of coverage (2D/3D graphics, sound, medical, chemical etc) [more...]
- Documentation for each supported format (history, elements, examples etc) [sample format]
- Convenience methods for URLs, compression, etc...
- Support for computable data (mathematical, physical, chemical, financial etc) [more...]
- Support for properties to extract potentially smaller part of much larger data sources
- Continuously updated data sources

Import and Export of Files [more...]

Use the "Elements" to decide what part to import: [file]

```
Import["http://exampledata.wolfram.com/mersenne.xhtml",
 "Elements"]
Import["http://exampledata.wolfram.com/mersenne.xhtml",
 "Data"]
ListLogPlot[%, Joined → True]
```

Can also import from URLs:

```
Import[
 "http://www.eia.doe.gov/pub/international/iealf/table12.xls",
 "Elements"]
```

Import row 250 from sheet 1:

```
data =
 Import[
  "http://www.eia.doe.gov/pub/international/iealf/table12.xls",
  {"Data", 1, 250}]
DateListPlot[Take[data, {4, -1}], {1980}, Filling -> Bottom,
 Joined -> True]
```

Use options to control other aspects of the process:

```
{Import["ExampleData/aspirin.mol"],
 Import["ExampleData/aspirin.mol", "Rendering" → "Wireframe"],
 Import["ExampleData/aspirin.mol", "Rendering" → "Spacefilling"]}
```

Slide 10 of 15

Benefits of CUDA Integration in *Mathematica*

For users who want to tap into the power of GPU computing, CUDA integration in *Mathematica* provides benefits in both development and performance. The full integration and automation of *Mathematica*'s CUDA capability means a more productive and efficient development cycle. In addition, it brings unprecedented levels of performance improvements without extra development time and cost.

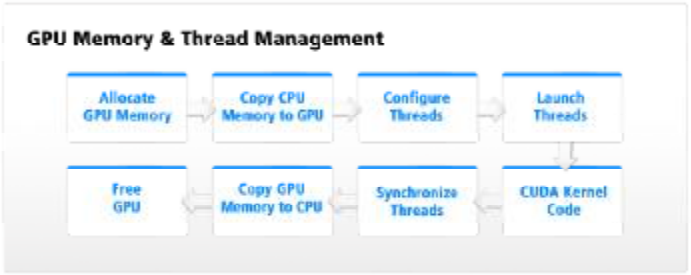
Simplified Development Cycle

Automation of development project management

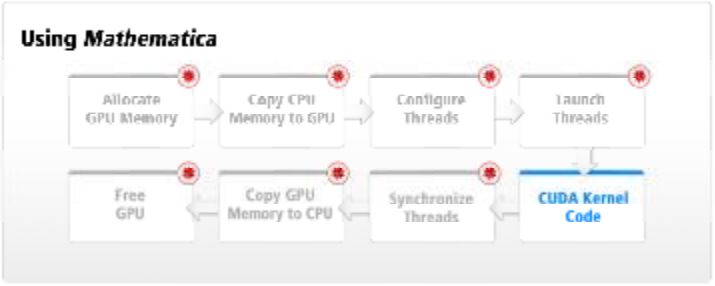
Like many other development frameworks, programming with CUDA require programmers to manage project setup, platform dependencies, and device configuration. CUDA integration in *Mathematica* makes the process completely transparent and fully automated.

Automated GPU memory and thread management

In a typical CUDA program, programmers write memory and thread management code manually, in addition to a CUDA kernel function:



With *Mathematica*, memory and thread management for the GPU is automatic:

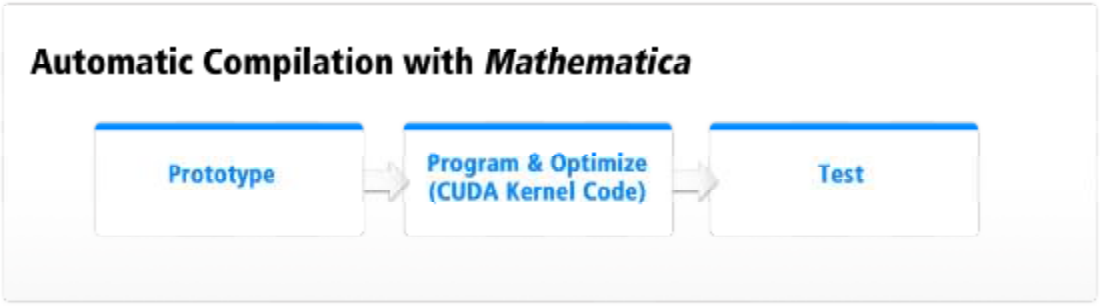


For advanced applications, full control of how and when the memory needs to be copied between

the host and GPU devices is provided.

Automatic compilation and binding of dynamic libraries

Mathematica's CUDA support streamlines the whole programming process, allowing for faster code turnaround



Full integration with *Mathematica*'s built-in capabilities

CUDA integration provides full access to *Mathematica*'s native language and built-in functions. It also provides free exchange of data between *Mathematica* and users' CUDA programs.

With *Mathematica*'s comprehensive symbolic and numerical functions, built-in application area support, and graphical interface building functions, users can not only combine the power of *Mathematica* and GPU computing, but also spend more time on developing and optimizing core CUDA kernel algorithms.

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 later in the slides.

Slide 11 of 15

Mathematica' s CUDALink: Integrated GPU Programming

CUDALink is a built-in *Mathematica* package that provides a simple and powerful interface for using CUDA within *Mathematica*'s streamlined workflow.

CUDALink provides you with carefully tuned linear algebra, discrete Fourier transform, and image processing algorithms. You can also write your own *CUDALink* modules with minimal effort. Using *CUDALink* from within *Mathematica* gives you access to *Mathematica*'s features including visualization, import/export, and programming capabilities.

The *CUDALink* package included with *Mathematica* at no additional cost offers:

- Compilation of CUDA programs
- Multiple-GPU support
- Support for single and double arithmetic precision operations
- Access to *Mathematica*'s flexible programming language, automatic interface builders, and full-featured development environment
- Access to *Mathematica*'s computable data, import/export capabilities, visualization features, and more
- Ready-to-use functionality with zero configuration in areas such as image processing, FFT, and linear algebra

Slide 12 of 15

Getting started with *CUDALink*

```
$Line = 0;
```

Programming the GPU in *Mathematica* is straightforward. It begins with loading the *CUDALink* package into *Mathematica*:

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

The following function verifies that the system has CUDA support:

```
CUDAQ[]
True
```

Cellular Automaton

Rule 30 Cellular Automaton does not gain much from CUDA until the column count becomes very large, since the next row is dependent on the previous. None the less, one can write a simple Rule 30 Cellular Automaton as a CUDA function

First, write a CUDA kernel function as a string, and assign it to a variable:

```
code = "
#define BLOCKDIM    256

__global__ void rule30ca_kernel(int * prevRow, int *
    nextRow, int width) {
    __shared__ int smem[BLOCKDIM+2];
    int tx = threadIdx.x, bx = blockIdx.x;
    int index = tx + bx*BLOCKDIM;

    smem[tx+1] = index < width ? prevRow[index] : 0;
    if (tx == 0)
        smem[0] = index > 0 ? prevRow[index-1] : 0;
    else if (tx == BLOCKDIM-1)
        smem[BLOCKDIM+1] = index < width-1 ?
    prevRow[index+1] : 0;

    __syncthreads();

    if (index < width)
        nextRow[index] = smem[tx] ^ (smem[tx+1] |
    smem[tx+2]);
}";
```

Pass that string to a built-in function **CUDAFunctionLoad**, along with the kernel function name and the argument specification. The last argument denotes the dimension of threads per block to be launched.

```
rule30 = CUDAFunctionLoad[code, "rule30ca_kernel",
  {{_Integer, _, "Input"}, {_Integer, _, "Output"}, _Integer}, 256]
```

Now you can apply this new CUDA function to any Array.

```
prevRow = ConstantArray[0, 256];
prevRow[[128]] = 1;
nextRow = ConstantArray[0, 256];
ca = {prevRow};
Do[
  res = rule30[prevRow, nextRow, 256];
  prevRow = First[res];
  AppendTo[ca, prevRow],
  {128}
 ];
ArrayPlot[ca]
```

Slide 13 of 15

Color Negate

Now, we will create a simple example that negates colors of a 3-channel image.

First, write a CUDA kernel function as a string, and assign it to a variable:

```
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 a built-in function **CUDAFunctionLoad**, along with the kernel function name and the argument specification. The last argument denotes the dimension of threads per block to be launched.

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

Several things are happening at this stage. *Mathematica* automatically compiles the kernel function as a dynamic library. There is no need for users to add system interface or memory management code. After compilation, the function is automatically bound to *Mathematica* and is ready to be called.

Now you can apply this new CUDA function to any image format that *Mathematica* can handle.

```
i =  ;
```

```
colorNegate[i, ImageDimensions[i], ImageChannels[i]]
```

Julia Set

◄ | ►

Accessing System Information

CUDALink supplies several functions that make it easy to acquire detailed system information for GPU programming.

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

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

This display all the information for CUDA capable devices in the local system

```
CUDAInformation[]
CUDAInformation[1, "Compute Capabilities"]
CUDAInformation[1, "Core Count"]
```

CUDAInformation generates a detailed report on supported CUDA devices. The returned data from **CUDAInformation** is a valid *Mathematica* input form, which means that it can be used to optimize CUDA kernel code programmatically. Several other functions are also provided that return in-depth information about *Mathematica*, operating systems, hardware, and C/C++ compilers that are currently used by *CUDALink*.

Device 1 | Device 2 | Device 3 | Device 4

Name	**GeForce GTX 295**
Clock Rate	**1 296 000**
Compute Capabilities	**1.3**
GPU Overlap	**1**
Maximum Block Dimensions	**{512, 512, 64}**
Maximum Grid Dimensions	**{65 535, 65 535, 1}**
Maximum Threads Per Block	**512**
Maximum Shared Memory Per Block	**16 384**
Total Constant Memory	**65 536**
Warp Size	**32**
Maximum Pitch	**2 147 483 647**
Maximum Registers Per Block	**16 384**
Texture Alignment	**256**
Multiprocessor Count	**30**
Core Count	**240**
Execution Timeout	**1**
Integerated	**False**
Can Map Host Memory	**False**
Compute Mode	**Default**
Texture1D Width	**8192**
Texture2D Width	**65 536**
Texture2D Height	**32 768**
Texture3D Width	**2048**
Texture3D Height	**2048**
Texture3D Depth	**2048**
Texture2D Array Width	**8192**
Texture2D Array Height	**8192**
Texture2D Array Slices	**512**
Surface Alignment	**256**
Concurrent Kernels	**False**
ECC Enabled	**False**
Total Memory	**911 736 832**

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

Retargetable Code Generation

CUDALink provides you with the ability to perform on-the-fly compilation and execution, or to compile executables or libraries to be used later. Code can also be compiled into an executable or an external library for out-of-*Mathematica* use.

Mathematica provides SymbolicC which provides a hierarchical view of C code as *Mathematica*'s own language. This makes it well suited to creating, manipulating, and optimizing C code. In conjunction with this capability, users can generate CUDA kernel code for several different targets, for greater portability, less platform dependency, and better code optimization.

Several built-in functions perform code generation, depending on the target:

SymbolicC

`CUDACodeGenerate` takes a CUDA kernel function or program and generates SymbolicC output. The SymbolicC output can then be used to render CUDA code that calls the correct functions to bind the CUDA code to *Mathematica*.

`CUDASymbolicCGenerate` produces the abstract syntax treee output for CUDA kernel as the wrapper code.

Dynamic library

`CUDALibraryGenerate` generates CUDA interface code and compiles it into a library that can be loaded into *Mathematica*.

String

`CUDACodeStringGenerate` generates CUDA interface code in string form which can be exported to other development platforms.

Slide 14 of 15

Integration with *Mathematica* Functions

When using *CUDALink*, all *Mathematica*'s features including visualization, import/export, and programming capabilities are at disposal. Combining *Mathematica*'s full-featured development environment and CUDA integration, you can focus on innovating your algorithms in CUDA kernels, rather than spending time on repetitive tasks, such as interface building.

Manipulate: *Mathematica*'s automatic interface generator

Mathematica provides extensive built-in interface functions including both standard and advanced controls. Users can also customize controls using *Mathematica*'s highly declarative interface language. Furthermore, *Mathematica* provides a fully automated interface generating function called **Manipulate**.

By specifying possible ranges for variables, **Manipulate** automatically chooses appropriate controls and creates a user interface around it.

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

Example of a user interface built with **Manipulate.**

Slide 15 of 15

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.

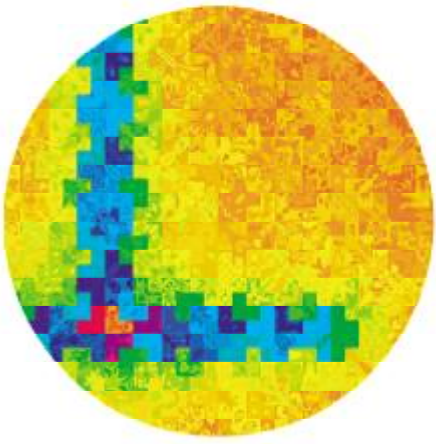
Not only does *Mathematica* provide access to local resources, but any URL can be used to access data online. The following code imports an image from a given URL:

```
$Line = 0;
```

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

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

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



The following code exports CUDA output into PNG format:

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

```
masked.png
```

Mathematica's *CUDALink* Applications

In addition to support for user-defined CUDA functions and automatic compilation, *CUDALink* includes several ready-to-use functions that support image processing, Fourier analysis, financial derivatives, and linear algebra.

Image Processing

CUDALink offers many image processing functions that have been carefully tuned for the GPU. These include pixel operations such as image arithmetic and composition; morphological operators such as such as erosion, dilation, opening, and closing; and image convolution and filtering. All of these operations work on either images or arrays of real and integer numbers.

Image convolution

CUDALink's convolution is similar to *Mathematica*'s **ListConvolve** and **ImageConvolve** functions. Here we operate on an image:

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.

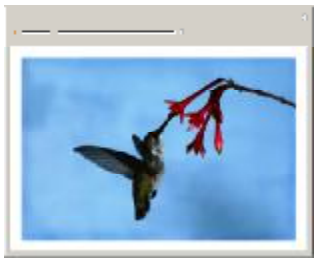
CUDAImageMultiply[ , ]

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 *Mathematica*'s built-in **Erosion**, **Dilation**, **Opening**, and **Closing** functions. More sophisticated morphological operations can be built using these fundamental operations.

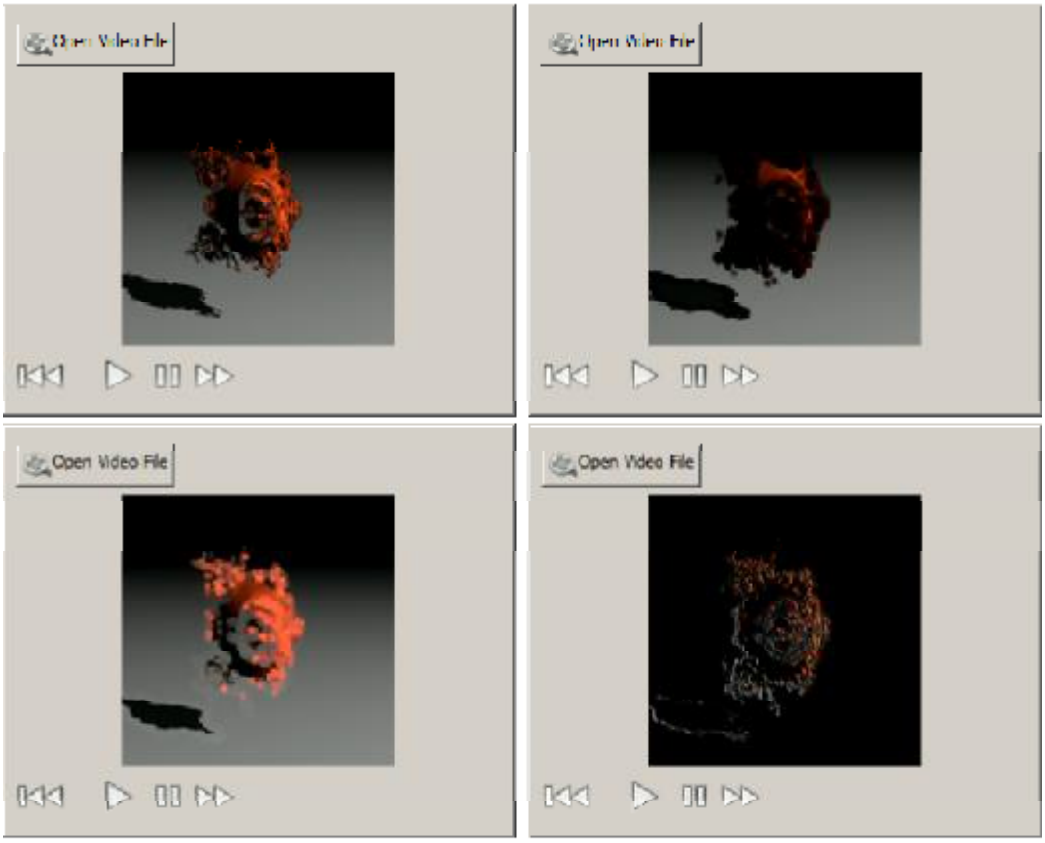
```
Manipulate[CUDAErosion[ <image>, ii], {ii, 0, 10}]
```



Video Processing

CUDALink's built-in image processing functions can also be applied to videos to perform real-time filtering. Many common formats such as H.264, QuickTime, and DivX are supported. With GPU computing power, *CUDALink*'s video processing function can easily handle full high-resolution video (1080p) filtering in 30 frames per second.

```
Needs["GPUExamples`"]
movieFile =
  FileNameJoin[{NotebookDirectory[], "Data", "mb.avi"}];
Grid[
 {{VideoProcessing["InputFile" -> movieFile,
    ImageSize → {640, 360} / 2],
   VideoProcessing["InputFile" -> movieFile,
    "ProcessingFunction" → (CUDAErosion[#, 5] &),
    ImageSize → {640, 360} / 2]},
  {VideoProcessing["InputFile" -> movieFile,
    "ProcessingFunction" → (CUDADilation[#, 5] &),
    ImageSize → {640, 360} / 2],
   VideoProcessing["InputFile" -> movieFile,
    "ProcessingFunction" →
     (CUDAImageConvolve[#, {{-1, 0, 1}, {-2, 0, 2},
         {-1, 0, 1}}] &), ImageSize → {640, 360} / 2]}}]
```



Processing multiple video streams with different filters in real time.

Linear Algebra

You can perform various linear algebra functions with the *CUDALink*. Examples include vector addition, products, and other operations, finding minimum or maximum elements, or transposing rows and columns of an image.

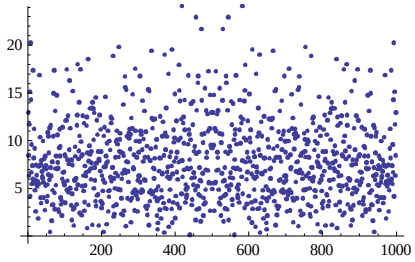
```
CUDADot[RandomReal[1, {10, 10}], RandomReal[1, {10, 10}]] //
 MatrixForm
```

```
2.86741  2.27721  1.33477  2.06823   2.4862   1.98769  2.45262  1.86956
 2.8771   2.5447   1.6475  2.27753  2.41796  2.37178  2.79259  2.13496
 3.0303  2.90008  1.51209  2.49778  2.87887  2.56315  3.18617  2.30868
3.11159  3.09919  2.09021  2.34827  2.65171  2.53904  3.14003   1.6707
 2.8931  2.15938  1.53122  1.85838  2.59512  2.41974   2.6245  2.02024
2.66476  2.61839  1.68847  1.99216   2.1434  2.46542   2.7474  1.27924
3.59753  2.93054  2.15184  2.59905  2.99768   2.7365  3.04215  1.97228
2.51176  1.97733  1.00389  2.06084  2.39118  1.75387  2.32628  2.17611
3.40237  2.95214  2.16906  2.26805  2.78934  2.84702  3.15229  1.73759
2.64359  1.71209  1.44451  1.35907  2.11079  1.68986  1.83644  1.53745
```

Fourier Analysis

The Fourier analysis capabilities of the *CUDALink* application include forward and inverse discrete Fourier transforms.

```
CUDAFourier[RandomReal[1, 1000]] // Abs // ListPlot
```



Multiple GPUs

Using *Mathematica*'s built in parallel capabilities, or using gridMathematica, multiple GPUs can be used to perform an operation:

```
LaunchKernels[]
{KernelObject[1, local], KernelObject[2, local],
 KernelObject[3, local], KernelObject[4, local]}
Needs["CUDALink`"]
```

This loads CUDALink for all kernels

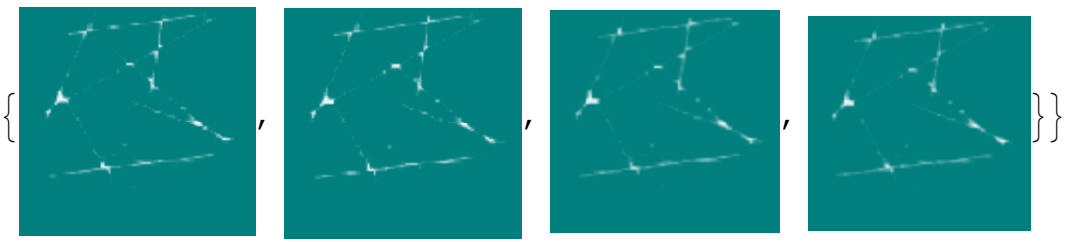
```
ParallelNeeds["CUDALink`"]
```

This sets the $CUDADevice for all kernels

```
ParallelEvaluate[$CUDADevice = $KernelID] // AbsoluteTiming
{0.0146490, {1, 2, 3, 4}}
```

This evaluates CUDAErosion on all devices on the system

```
ParallelEvaluate[CUDAErosion[ , 2]] // AbsoluteTiming
```



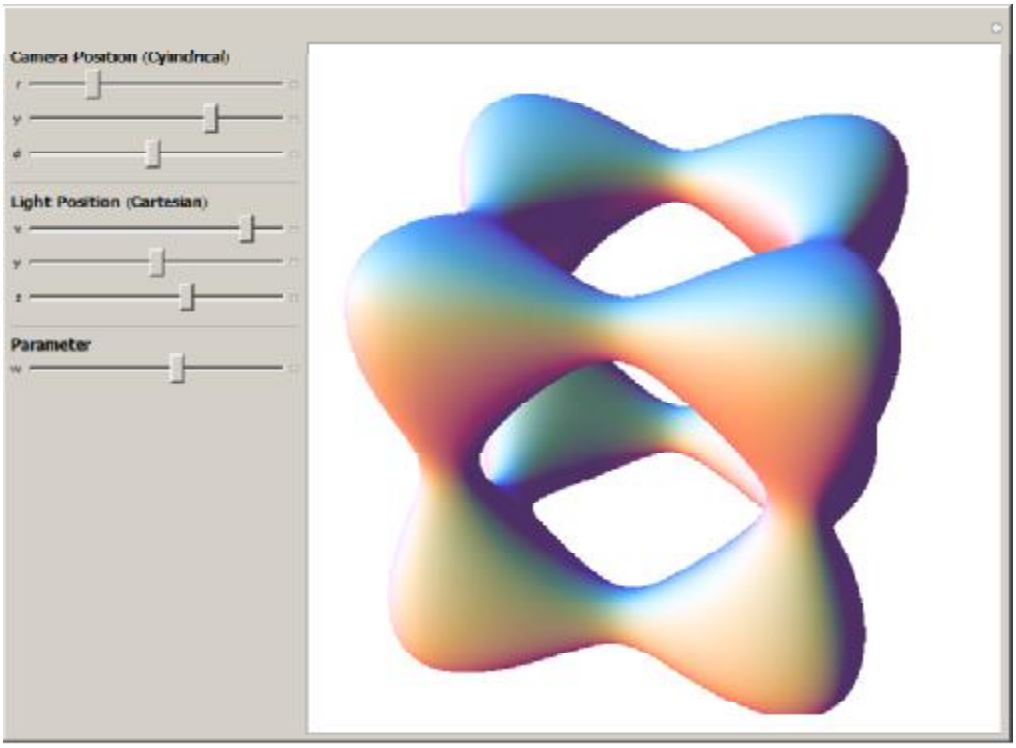
Here, there is a 2x speedup, since most of the time is spent copying data

```
4 * First[CUDAErosion[ , 2] // AbsoluteTiming]
0.027345
```

OpenCL Compatibility

In addition to *CUDALink*, *Mathematica* supports OpenCL with the built-in package *OpenCLLink*, which provides the same benefits and functionality of GPU programming as *CUDALink* over OpenCL architecture.

```
<< OpenCLLink`
tangle = (x^2 - 5) * x^2 + (y^2 - 5) * y^2 + (z^2 - 5) * z^2 + w;
OpenCLImplicitRender3D[tangle, {x, y, z, w}, 7.5, "Precision" → 0.01,
 "SliderParameters" → {0.0, 20.0, 11.8}, "Shadows" → False]
```



◄ | ►

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:
http://www.wolfram.com/products/mathematica/purchase.html

Summary

Thanks to *Mathematica*'s integrated platform design, all functionality is included without the need to buy, learn, use, and maintain multiple tools and add-on packages.

With its simplified development cycle, automatic memory management, multicore computing, and built-in functions for many applications—plus full integration with all of *Mathematica*'s other computation, development, and deployment capabilities—*Mathematica*'s built-in *CUDALink* package provides a powerful interface for GPU computing.

Initialization

Mathematica Technology Conference

October 13 - 15, Champaign, Illinois, USA

Join us for the 2010 Wolfram Technology Conference, which brings together leading experts to discuss how Wolfram technologies are shaping technical computing today and in the future.A forum for users from all fields, the event provides a unique opportunity to learn from experts, and each other, about how to work more efficiently using the latest Wolfram tools and resources.

- Included with registration are CUDALink and OpenCLLink training sessions
- Price to register is $695 for Standard, and $495 for Educators
- We have one-day registrations for $75. We are also offering the Developer and Author Summit on Tuesday, also for $75.
- Register through October 9, however, you will not be turned away if registered late..