Heterogeneous Computing in Mathematica® 8
Introduction

The past few years have seen an increase in the number of cores on both the CPU and GPU. Despite this, developers have not been able to fully utilize the parallel frameworks. As developers began to target the CPU and GPU frameworks, they realized that some algorithms map nicely onto the CPU while others work best with the GPU. As a result, much research has been done to develop algorithms that exploit both the CPU’s and GPU’s capabilities. The research culminated in a heterogeneous school of thought, with both the CPU and GPU collaborating to maximize the accuracy and speed of the user’s program. In such methodology, the GPU is utilized in parts where it excels, as is the CPU. Yet while heterogeneous algorithms are ideal, no system has exposed built-in access to both the CPU and the GPU with a concise and easy-to-use syntax until Mathematica 8 which makes heterogeneous computing a reality.

By providing an environment where programs can be run on either the CPU or GPU, Mathematica is the only computational system that realizes the heterogeneous message. Coupled with Mathematica’s language, comprehensive import/export support, symbolic computation, extensive field coverage, state-of-the-art visualization features, platform neutrality, and ease of use, Mathematica is ideal for heterogeneous algorithm development.

This white paper is divided as follows: first, we briefly discuss what Mathematica is and why you should use it for heterogeneous computing, then we look at some of Mathematica’s multicore and GPU features, develop a dozen applications along the way, and finally discuss why Mathematica has an advantage over other systems.

A Brief Introduction to Mathematica

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 perform computations on the GPU and thus facilitates heterogeneous computing. 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:

Free-form linguistic input

Mathematica’s free-form linguistic input is the ability to interpret English text as Mathematica code. It is a breakthrough in usability, making development intuitive and simple.

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.

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.

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

FullFeatured, 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 environment based on the Eclipse platform.

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.
Platform-independent deployment options
Through its interactive notebooks, Wolfram CDF 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 gridMathematica™ allows Mathematica programs to be parallelized on many machines in cluster or grid configuration. Also available is webMathematica™ which allows Mathematica programs to be run on a web server.

Motivations for Heterogeneous Computing in Mathematica
Over the past few years, multicore systems have transitioned from being found only on specialty devices to commodity devices. With CPUs routinely being 2-, 4-, or 8-cores, software has not been able to exploit the now-common multicore features.

The GPU has also transitioned from being used only for graphics and gaming to a device capable of performing general computation. The GPU has quickly surpassed the CPU in terms of performance, with a modest GPU able to perform over 20 times more computations per second than a CPU. This is because of the GPU architecture, which reduces cache and RAM in favor of arithmetic logical units (ALU).

Unlike Moore’s law for the CPU—which states that the number of transistors that can be placed inexpensively on an integrated circuit doubles approximately every 18 months—GPUs have been quadrupling the number of transistors every 18 months. With an architecture that facilitates the addition of more ALUs, the GPU has quickly surpassed the CPU in terms of raw computational power, making the GPU ideal for certain forms of computation.

Yet as algorithms were implemented for the GPU, it became apparent that the lack of cache and control makes some algorithms either difficult or inefficient on the GPU. Hence the popularization of heterogeneous computing, which uses the CPU in cases where the GPU is inefficient and the GPU where speedups are desired.

Mathematica 7 introduced multicore programming in Mathematica, and Mathematica 8 made many algorithms run on multicore systems in addition to introducing GPU programming. By providing an interface for running programs on both the CPU and the GPU, Mathematica embodies the heterogeneous computing philosophy, allowing users to choose either the CPU or GPU based on the program’s performance.

CPU Multicore Integration in Mathematica
Most of the built-in functions, like linear algebra, image processing, and wavelets, already make use of all cores on the system. Mathematica also exposes to the user a few ways to accelerate programs using multiple cores. Those functions can be further extended to run on
a cluster of machines with the use of grid Mathematica, as well as be used in conjunction with the GPU capabilities in Mathematica 8.

Compile

Mathematica’s Compile command takes a sequence of inputs and expressions, and, based on the input usage, performs type inferencing to construct a byte code representation of the program. The byte code can be targeted to Mathematica’s internal virtual machine (WVM) or C. Compile is used internally by many Mathematica functions and, when it makes sense, some expressions are automatically compiled for the user—when constructing a list using a function, for example, the function is automatically created to speed up the construction.

<table>
<thead>
<tr>
<th>Compile Command</th>
<th>Compile Instructions</th>
<th>Compile Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compile [(x), Sin[x] + x^2 - 1/(x^2)]</td>
<td>Instruction[(Sin, x, x)] Instruction[(Power, x, 2, x)] Instruction[(Plus, x, x)] Instruction[(Div, x, x)] Instruction[(Plus, x, x)]</td>
<td>Compile[(x), Sin[x] + x^2 - 1/(x^2)] CompilationTarget-&gt;&quot;C&quot;</td>
</tr>
</tbody>
</table>

Mathematica’s Compile command translates Mathematica expressions into a sequence of instructions. The compiler performs type inferencing, type coercion, common subexpression elimination, and dead code removal. Since Compile translates expressions into byte code close to machine instructions, it works with machine precision and tracks when overflows and underflows occur. The Mathematica byte code is a machine-independent representation of both arithmetic and instructions as well as list manipulation instructions. The list instructions represent common function such as Map and Fold. The instructions are simple, either unary, binary, or ternary instructions that are inert, allowing the user to further manipulate the symbolically.

Compile works only on a subset of the Mathematica language, and although it sacrifices some Mathematica features, it does afford the user significant speed improvements.

To use the Compile command, the user passes in the input variables along with a sequence of Mathematica expressions. Compile will automatically perform code optimization—the following code, for example, performs common subexpression elimination evaluating $x^2$ once.

```mathematica

cf = Compile[{x}, Sin[x] + x^2 - 1/x^2]

CompiledFunction[{x}, Block[{Compile$2},
  Compile$2 = x^2; Sin[x] + Compile$2 - 1/Compile$2], -CompiledCode-]
```

The CompiledFunction returned behaves the same as any Mathematica function. It can be visualized as follows.

```mathematica
Plot[cf[x], {x, -\[Pi], \[Pi]}]
```

The Compile statement generates byte code from the Mathematica program input by the user. The byte code can be printed to understand what code is being generated.
Needs["CCodeGenerator"]
CompilePrint[cf]

1 argument
5 Real registers
Underflow checking off
Overflow checking off
Integer overflow checking on
RuntimeAttributes -> {}

R0 = A1
Result = R2

1 R1 = Square[ R0]
2 R2 = Sin[ R0]
3 R3 = Reciprocal[ R1]
4 R4 = - R3
5 R2 = R2 + R1 + R4
6 Return

The byte code generated can be used to target the Wolfram Virtual Machine (WVM), C, LLVM IR, CUDA, OpenCL, or JVM with both WVM and C built into Mathematica 8.

Compilation Target
By default, Compile targets the WVM, but if CompilationTarget->"C" is set, then Compile will generate a C version of the Mathematica program, compile it, and run it. This allows Mathematica to run at the same speed as C, without going through low-level C programming.

cf = Compile[{x}, Sin[x] + x^2 - \frac{1}{x^2}, CompilationTarget -> "C"]

CompiledFunction[{x}, Block[{Compile`$3},
Compile`$3 = x^2; Sin[x] + Compile`$3 - \frac{1}{Compile`$3}, -CompiledCode-]

Since Mathematica is a highly expressive language, programs that take dozens or hundreds of lines in C code can be written in two or three lines in Mathematica and have similar performance. The following, for example, uses Compile to generate a multi-threaded C program to compute the Mandelbrot set. The code is executed from within Mathematica.

compileMandelbrot = Compile[{{c, _Complex}},
Boole[Norm[FixedPoint[x \rightarrow x^2 + c &, 0, 1000, SameTest -> (Norm[#] > 4 &)]] > 4],
CompilationTarget -> "C", RuntimeAttributes -> Listable ];

The following sets the evaluation points for the Mandelbrot function.

evaluationPoints = Table[a + I b, {b, -1, 1, 0.005}, {a, -1.5, 0.5, 0.005}];
And finally, we invoke the function with the evaluation points and plot the result.

```mathematica
ArrayPlot[compileMandelbrot[evaluationPoints]]
```

Written in C, the above would have taken many hundreds of lines of code, required project setup, and might not have been portable across systems.

**Automatic vectorization**

*Mathematica* 8 added the ability to automatically parallelize Compile statements. Making a CompiledFunction run in parallel is simple—the user only has to pass the option `RuntimeAttributes->"Listable"`. From there, `Compile` will run in as many threads as there are on the system.

As an example, we implement a basic ray tracer. The ray tracer takes a list of sphere centers, sphere radii, sphere colors, and a ray origin position. It then shines a parallel ray from the ray origin and records which sphere intersects the ray and associates a color with the intersection. We will compile the code into "C" by passing the `CompilationTarget->"C"` option, as we will use `RuntimeAttributes->"Listable"` to make the code run in parallel.

```mathematica
raySpheresIntersectionColor = Compile[
  {{centers, _Real, 2},
   {radii, _Real, 1}, {colors, _Real, 2}, {x, _Real}, {y, _Real}},
  Module[
    ray = {x, y, 0.0}, v, rad, center, res = {0., 0., 0.}],
  Do[
    rad = radii[[ii]]^2;
    center = centers[[ii]];
    v = Norm[ray - center]^2;
    If[v < rad,
      res = colors[[ii]] (rad - v)
      rad - v];
    ], {ii, Length[centers]}
  ];
  res
], CompilationTarget -> "C", RuntimeAttributes -> Listable
]
```

Here, we define a *Mathematica* function that calls the above compiled function.

```mathematica
rayTraceCompile[centers_, radii_, colors_, x_, y_] :=
Module[{xx, yy},
  xx = Transpose[ConstantArray[x, Length[y]]];
  yy = ConstantArray[y, Length[x]];
  raySpheresIntersectionColor[centers, radii, colors, xx, yy]
]
```
This constructs the scene, generating 100 spheres in random positions and creating the ray origins. We use the *Mathematica* symbol `ColorData` to get colors that follow the "DarkBands" color scheme.

\[
\begin{align*}
\text{width} &= \text{height} = 300; \\
\text{nx} &= \text{ny} = 300; \\
\text{numSpheres} &= 100; \\
\text{centers} &= \text{Table}\left[\left\{\text{RandomReal}\left[\left\{-\frac{\text{width}}{2}, \frac{\text{width}}{2}\right\}\right], \right. \\
&\text{RandomReal}\left[\left\{-\frac{\text{height}}{2}, \frac{\text{height}}{2}\right\}\right], \text{RandomReal}[100]\right\}, \{\text{numSpheres}\}\right]; \\
\text{radii} &= \text{RandomReal}\left[\left\{5, 30\right\}, \text{numSpheres}\right]; \\
\text{colors} &= \text{ReplaceAll}\left[\text{ColorData}\left[\text{"DarkBands"}ight]@\text{Range}\left[0, 1, 1/\text{numSpheres}\right], \text{RGBColor} \rightarrow \text{List}\right]; \\
\text{x} &= \text{width} * (\text{N}[\text{Range}[0, \text{nx}] / \text{nx} - .5]); \\
\text{y} &= \text{height} * (\text{N}[\text{Range}[0, \text{ny}] / \text{ny} - .5]);
\end{align*}
\]

This runs the ray tracer and visualizes the result.

\[
\text{imgc} = \text{rayTraceCompile}[\text{centers}, \text{radii}, \text{colors}, \text{x}, \text{y}]; \\
\text{Image}[\text{imgc}]
\]

Again, the speed of this program is comparable to a program written in C, but unlike the C version, our code is around 20 lines long.

**Parallel Computing**

Built into the *Mathematica* language since Version 7 is the capability for multicore computing. *Mathematica*‘s parallel tools allow users to make use of all cores on their system, developing more sophisticated projects that execute at a fraction of the time. *Mathematica*‘s parallel infrastructure is also set up to allow seamless scaling to networks, clusters, and grids.

The most basic parallel program you can write is a Monte Carlo integrator for approximating \(\frac{\pi}{4}\). This is done by generating uniform random points in the \([0,1] \times [0,1]\) region. The proportion of points with a norm less than 1 approximates \(\frac{\pi}{4}\). The following shows uniform points in the \([0,1] \times [0,1]\) region with red points having a norm less than 1. As can be seen, those points approximately cover a quarter of a unit circle.

\[
\text{pts} = \text{RandomReal}[1, \{10000, 2\}]; \\
\text{Graphics}\left[\left\{\text{AbsolutePointSize}[2], \left\{\text{Red}, \text{Point}[\text{Select}[\text{pts}, \text{Norm}[\#] \leq 1 \&]]\right\}, \\
\text{\{Black, Point[Select[pts, Norm[#] > 1 \&]]\}\right\}\right]\]
\]
The *Mathematica* language has extensive support for list processing. This makes writing the above Monte Carlo $\pi$-approximation trivial.

```mathematica
n = 1000000;
Mean[Table[If[Norm[RandomReal[1, 2]] <= 1.0, 0.0], {n}]]
0.784838
```

*Mathematica* 7 introduced parallel primitives such as `ParallelMap` and `ParallelTable`. The parallelization has been further simplified in *Mathematica* 8 by introducing `Parallelize`, which performs automatic parallelization. So, to make the above program run on multiple cores, the user just has to place `Parallelize` around `Table`, and it is automatically run in parallel.

```mathematica
n = 1000000;
Mean[
  Parallelize[
    Table[If[Norm[RandomReal[1, 2]] <= 1.0, 0.0], {n}]
  ]
]
0.784934
```

Parallel tools allow fine-grained control on how to split the parallel tasks and how many parallel kernels to run. A queuing mechanism is also available to saturate the CPU usage and achieve the best speedup.

**gridMathematica**

`gridMathematica` extends *Mathematica*’s parallel functionality to run on homogeneous and heterogeneous networks and clusters. By installing the `gridMathematica` server on a machine, the machine will broadcast its availability to the master machine.
Similar to *Mathematica’s* parallel tools, grid*Mathematica* allows for fine-grained control over the scheduling of tasks, giving priority to certain kernels. grid*Mathematica* also interfaces with existing grid management software from Microsoft, Sun, LFS, PBS, etc. It also contains its own management software called Wolfram Lightweight Grid™ Manager.

Programs written using *Mathematica’s* parallel tools are valid grid*Mathematica* programs, and thus *Mathematica* is easily scalable from single multicore systems to clusters.

**GPU Integration in Mathematica**

CUDA*Link* and OpenCL*Link* offer a high-level interface to the GPU built on top of *Mathematica’s* language and development technologies. They allow users to execute code on their GPU with minimal effort. By fully integrating and automating the GPU’s capabilities using *Mathematica*, hiding unnecessary complexity associated with GPU programming, and lowering the learning curve for GPU computation, users experience a more productive and efficient development cycle.

**CUDA*Link* and OpenCL*Link* Support**

CUDA*Link* is supported on all CUDA-enabled NVIDIA hardware. *Mathematica* is bundled with the CUDA Toolkit. This makes the NVIDIA driver and a supported C compiler the only requirements for CUDA*Link* programming.

OpenCL*Link* supports both NVIDIA and AMD GPUs, with only the NVIDIA driver being needed for NVIDIA GPUs. Both the AMD video driver and AMD APP SDK are needed for AMD. OpenCL*Link* also supports CPU implementations of OpenCL provided by either AMD or Intel.

GPUs that have CUDA or OpenCL support are supported by CUDA*Link* or OpenCL*Link* with *Mathematica* automatically determining the precision of the card and executing the appropriate function based on the maximal floating point precision. CUDA*Link* and OpenCL*Link* are supported on all platforms supported by *Mathematica* 8—Windows, Mac OS X, and Linux, both 32- and 64-bit.

**Making GPU Programming Easy**

By removing repetitive and low-level GPU related tasks, *Mathematica* makes GPU programming simple.
Automation of development project management
Unlike other development frameworks that require the user to manage project setup, platform dependencies, and device configuration, *Mathematica* makes the process of GPU programming transparent and automated.

Automated GPU memory and thread management
A GPU program written from scratch delegates memory and thread management to the programmer. This bookkeeping is required in addition to the need to write the GPU kernel.

With *Mathematica*, memory and thread management are 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.

As a result of hiding the bookkeeping and repetitive tasks found in GPU programming, *Mathematica* streamlines the whole programming process, allowing for simpler code, shorter development cycle, and better performance.

Integration with *Mathematica*'s built-in capabilities
*Mathematica*'s GPU integration provides full access to *Mathematica*'s native language and built-in functions. Users can write hybrid algorithms that use the CPU and GPU depending on the efficiency of each algorithm.

Ready-to-use applications
*Mathematica* provides several ready-to-use GPU functions that cover a broad range of topics such as computational mathematics, image processing, financial engineering, and more.

Zero device configuration
*Mathematica* automatically finds, configures, and makes GPU devices available to users. *Mathematica* will automatically detect and configure the required tools to make it simple to use the GPU.
Through *Mathematica*’s built-in parallel programming support, users can launch GPU programs on different devices. Users can also scale the setup across machines and networks using gridMathematica.

**Web deployment**

*Mathematica* can be deployed onto a web server using webMathematica. This means that GPU computation can be initiated from devices that do not have a GPU. This is ideal for the classroom, where the instructor may want to control the development environment, or mobile devices, which do not have GPUs.

**Mathematica’s OpenCLLink: Integrated GPU Programming**

OpenCLLink is a built-in *Mathematica* application that provides an interface for using OpenCL within *Mathematica*. Through OpenCLLink, users can execute OpenCL programs from *Mathematica* with little effort. Because *Mathematica* caches program compilation and makes intelligent choices about GPU memory transfer, users experience better execution speed versus handwritten OpenCL programs.

**Setting up OpenCLLink**

OpenCLLink supplies functions that query the system’s GPU hardware. To use OpenCLLink operations, users have to first load the OpenCLLink application.

```mathematica
Needs["OpenCLLink"]
```

OpenCLQ tells whether the current hardware and system configuration support OpenCLLink:

```mathematica
OpenCLQ[]
```

True

OpenCLInformation gives information on the available OpenCL hardware. Here, we query information about the first platform and device installed on the system.

```mathematica
OpenCLInformation[1, 1]
```

```
{Type -> GPU, Name -> Tesla C2050 / C2070, Version -> OpenCL 1.0 CUDA,
 Extensions -> {cl_khr_byte_addressable_store, cl_khr_icd, cl_khr_gl_sharing,
 cl_khr_d3d9_sharing, cl_khr_d3d10_sharing, cl_khr_d3d11_sharing,
 cl_nv_compiler_options, cl_nv_device_attribute_query, cl_nv pragma unroll,
 cl_khr_global_int32_base_atomics, cl_khr_global_int32_extended_atomics,
 cl_khr_local_int32_base_atomics, cl_khr_local_int32_extended_atomics, cl_khr_fp64},
 Driver Version -> 270.81, Vendor -> NVIDIA Corporation, Profile -> FULL_PROFILE,
 Vendor ID -> 4318, Compute Units -> 14, Core Count -> 448,
 Maximum Work Item Dimensions -> 3, Maximum Work Item Sizes -> {1024, 1024, 64},
 Maximum Work Group Size -> 1024, Preferred Vector Width Character -> 1,
 Preferred Vector Width Short -> 1, Preferred Vector Width Integer -> 1,
 Preferred Vector Width Long -> 1, Preferred Vector Width Float -> 1,
 Preferred Vector Width Double -> 1, Maximum Clock Frequency -> 1147,
 Address Bits -> 32, Maximum Memory Allocation Size -> 695.919200, Image Support -> True,
 Maximum Read Image Arguments -> 128, Maximum Write Image Arguments -> 8,
 Maximum Image2D Width -> 4096, Maximum Image2D Height -> 32768,
 Maximum Image3D Width -> 2048, Maximum Image3D Height -> 2048,
 Maximum Image3D Depth -> 2048, Maximum Samplers -> 16, Maximum Parameter Size -> 4352,
 Memory Base Address Align -> 4096, Memory Data Type Align Size -> 128,
 Floating Point Precision Configuration -> {Denorms, Infinity, NaNs,
 Round to Nearest, Round to Infinity, Round to Zero, IEEE754-2008 Pused MAD},
 Global Memory Cache Type -> Read Write, Global Memory Cache Line Size -> 128,
 Global Memory Cache Size -> 229376, Global Memory Size -> 2780364800,
 Maximum Constant Buffer Size -> 65536, Maximum Constant Arguments -> 9,
 Local Memory Type -> Local, Local Memory Size -> 49152, Error Correction Support -> True,
 Profiling Timer Resolution -> 1000, Endian Little -> True, Available -> True,
 Compiler Available -> True, Execution Capabilities -> {Kernel Execution},
 Command Queue Properties -> {Out of Order Execution, Profiling Enabled}]
```

**Example of a report generated by OpenCLInformation.**
OpenCLLink Programming

Programming the GPU in Mathematica is straightforward. It begins with writing an OpenCL program. The following OpenCL program negates colors of a multichannel image.

```c
__kernel void opencIColorNegate(__global
mint *img, __global mint *dim, mint channels) {
int width = dim[0], height = dim[1];
int xIndex = get_global_id(0), yIndex = get_global_id(1);
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];
}
}
```

The source code is passed to OpenCLFunctionLoad and the user gets a Mathematica function as output.

```math
OpenCLColorNegate = OpenCLFunctionLoad[src, "opencIColorNegate",
{(_Integer, "InputOutput"), (_Integer, "Input"), _Integer}, {16, 16}]
OpenCLFunction[<> , cudaMemcpy, cudaColorNegate,
{(_Integer, InputOutput), (_Integer, Input), _Integer}]
```

Now you can apply this new OpenCL function to an image.

```math
img = ;
OpenCLColorNegate[img, ImageDimensions[img], ImageChannels[img]]
```

OpenCLFunctionLoad follows a simple syntax, where the first argument is the OpenCL source, the second argument is the function name to be invoked, the third argument is a list of function parameter types, and the final argument is the work group size (block dimension).

```c
src = "__kernel void kernel (global mint * in, mint len) {...}"
OpenCLFunctionLoad[src, "kernel", {{_Integer}, _Integer}, blockDim]
```

Several things need to happen behind the scenes at this stage to load the OpenCL code into Mathematica efficiently.
First, we need to check to see if the arguments to \texttt{OpenCLFunctionLoad} are valid. Since this is the most common source of errors, we have to catch it early. Second, we compile the OpenCL function and cache it. The output from this step is an \texttt{OpenCLFunction} that behaves like any \texttt{Mathematica} function.

When an \texttt{OpenCLFunction} is invoked, we perform input checking to make sure it matches the input specification. We next load the data onto the GPU (memory copies are performed in a lazy fashion to minimize memory copy overhead). We then invoke a synchronization to make sure all computation has been performed, and get the results from the GPU.

\textbf{OpenCLLink Applications in Mathematica}

In this section we discuss some applications that run on the GPU using \texttt{OpenCLLink}. In \texttt{Mathematica} you can perform sophisticated heterogeneous computation easily by leveraging a variety of built-in features.

\textbf{Black–Scholes Equation}

The Black–Scholes equation is the basis of computational finance. It states that the call of a European style option can be modeled by a formula implemented by the following OpenCL program.

```c
#define N(x) (erf((x)/sqrt(2.0))/2+0.5)

__kernel void blackScholes(__global Real_t * call, __global Real_t * S,
                         __global Real_t * X, __global Real_t * T, __global Real_t *
                         R, __global Real_t * Q, __global Real_t * V, mint length) {
    int ii = get_global_id(0);
    if (ii < length) {
        Real_t d1 =
            (log(S[ii]/X[ii])+(R[ii]-Q[ii]+(pow(V[ii],(Real_t)2.0)/2)*T[ii]))/
            (V[ii]*sqrt(T[ii]));
        Real_t d2 = d1 - V[ii]*sqrt(T[ii]);
        call[ii] =
            S[ii]*exp(-Q[ii]*T[ii])*N(d1) - X[ii]*exp(-R[ii]*T[ii])*N(d2);
    }
}
```

\texttt{Real_t} is a type defined by \texttt{Mathematica} that maps to the highest precision of the OpenCL device. This ensures that users are getting the best accuracy when computing.
The following loads the above OpenCL code into *Mathematica*.

```mathematica
OpenCLBlackScholes = OpenCLFunctionLoad[code, "blackScholes", 
{(_Real), (_Real, "Input"), (_Real, "Input"), (_Real, "Input"), 
(_Real, "Input"), (_Real, "Input"), (_Real, "Input"), _Integer}, 128]
OpenCLFunction[<>], blackScholes, 
{(_Real), (_Real, Input), (_Real, Input), (_Real, Input), 
(_Real, Input), (_Real, Input), (_Real, Input), _Integer}]
```

This gets the stock price for the S&P 500 from the beginning of 2010 to March 2011. This data is curated by Wolfram Research and accessible via a web connection.

```mathematica
data = Transpose[FinancialData["SP500", {{2010, 0, 0}, {2011, 4, 0}}]];
```

This uses the S&P 500 data for the spot price and the dates for the expiration values. The rest of the data is randomly generated.

```mathematica
Needs["Calendar"]
numberOfOptions = Length[data[[1]]];
call = ConstantArray[0.0, numberOfOptions];
S = data[[2]];
X = 1.1*data[[2]];
T = (DaysBetween[H, {2011, 4, 20}] & /@ data[[1]]) / 365.;
R = RandomReal[{0.03, 0.07}, numberOfOptions];
Q = RandomReal[{0.01, 0.04}, numberOfOptions];
V = RandomReal[{0.10, 0.50}, numberOfOptions];
```

The following runs the computation on the OpenCL device.

```mathematica
res = OpenCLBlackScholes[call, S, X, T, R, Q, V, numberOfOptions];
```

We visualize the result as a Kagi chart. This visualization function is part of *Mathematica*’s comprehensive support for visualization and charting, which includes, plotting surfaces, computing bar and pie charts, and interacting with trading charts.

```mathematica
KagiChart[Transpose[{data[[1]], res[[1]]}]]
```

![Kagi Chart](chart_image)

**Computing with Data from an Excel File**

*Mathematica* supports many import and exports formats. One such format is Excel 2007 (XLSX), which we use in this example as source for our data when calculating 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;
_embedded
_opencl
#include <CL/cl.h>

__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 Mathematica in single precision mode.

OpenCLOneTouchOption =
OpenCLFunctionLoad[code, "onetouch", {{_Real, "Output"},
    {{_Real, "Output"}, {{_Real, "Input"}, {{_Real, "Input"},
    {{_Real, "Input"}, {{_Real, "Input"}, {{_Real, "Input"},
    {{_Real, "Input"}, _Integer, 128, "TargetPrecision" -> "Single"}}}}}}];

This imports the data from an Excel file and stores it in a Mathematica table.

rawData = First[Import["dataset.xlsx", "Data"]];
Do[data[First[row]] = Drop[row, 1], {row, Transpose[rawData]]];
numberOfOptions = Length[data["Spot Price"]];

This allocates memory for both the call and put result. We allocate the data as a "float".

call = OpenCLMemoryAllocate["Float", numberOfOptions];
put = OpenCLMemoryAllocate["Float", numberOfOptions];

This calls the function.

OpenCLOneTouchOption[call, put, data["Spot Price"],
    data["Strike Price"], data["Expiration"], data["Interest"],
    data["Dividend"], data["Volatility"], numberOfOptions]
{OpenCLMemory[<29650>, Float], OpenCLMemory[<29636>, Float]}

This retrieves the result for the call option.

OpenCLMemoryGet[call]
{1., 1., 0.93962, 1.63052, 0.895195, 1., 1., 1., 1., 0.940062,
    0.719849, 0.844359, 0.95156, 1., 1., 1., 0.933998, 1., 1.00517, 1.,
    0.113093, 0.34359, 1., 1., 1., 1., 1., 0.865092, 0.913892, 6.13967,
    1., 1.38213, 1., 0.939484, 1., 1., 1., 1.0914, 1., 1.0485, 0.889279,
    0.991108, 0.35102, 0.689786, 1., 1., 1., 0.982171, 3.07142, 1., 1.,
    0.999357, 0.850779, 1., 2.53988, 1., 1., 1., 1., 1., 0.852829, 1., 1.}

Conway's Game of Life

Conway's Game of Life is an example of a simple two-dimensional cellular automaton. From
simple rules that look only at the eight neighbors, it gives rise to complicated patterns. Here
is a basic OpenCL program that implements the Game of Life.

Heterogeneous Computing in Mathematica 8 | 15
src = "
_kernel void gol_kernel(__global mint *
    prev, __global mint * nxt, mint width, mint height) {
int xIndex = get_global_id(0), yIndex = get_global_id(1);
int index = xIndex + yIndex*width;
int ii, jj, curr, neighbors;
if (xIndex < width && yIndex < height) {
curr = prev[index];
neighbors = 0;
for (ii = -1, neighbors = -curr; ii <= 1; ii++) {
    if (xIndex + ii >= 0 && xIndex + ii < width) {
        for (jj = -1; jj <= 1; jj++) {
            if (yIndex + jj >= 0 && yIndex + jj < height)
                neighbors += prev[xIndex + ii + (yIndex + jj)*width];
        }
    }
}
if (curr == 1)
nxt[index] = (neighbors == 2 || neighbors == 3) ? 1 : 0;
else
    nxt[index] = (neighbors == 3) ? 1 : 0;
}
"

This loads the function using OpenCLFunctionLoad. We set the work group size to 16x16.

OpenCLGameOfLife = OpenCLFunctionLoad[src, "gol_kernel",
    {(_Integer, "Input"), (_Integer, "Output"), _Integer, _Integer}, {16, 16}]
OpenCLFunction[<>, gol_kernel,
    {(_Integer, Input), (_Integer, Output), _Integer, _Integer}]

We set the initial state using random choice, making 70% of the 512x512 initial state set to zero while the others are set to 1. We set the output state to all zeros.

initialState = RandomChoice[{0.7, 0.3} → {0, 1}, {512, 512}];
outputState = ConstantArray[0, {512, 512}];

This uses Dynamic to animate the result at 60 frames per second.

Dynamic[
    Refresh[
        initialState =
            First[OpenCLGameOfLife[initialState, outputState, 512, 512]];
        ArrayPlot[initialState, ImageSize → Medium],
        UpdateInterval → 1/60
    ]
]
"

Many-Body Physical Systems
The N-body simulation is a classic Newtonian problem. The OpenCL implementation is included as part of the Mathematica distribution.

srcf = FileNameJoin[{$OpenCLLinkPath, "SupportFiles", "NBody.cl"}];
This loads OpenCLFunction. Note that you can pass the vector type "float4" into the OpenCL program and *Mathematica* handles the conversion.

```mathematica
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.

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

This sets the input and output memories.

```mathematica
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.

```mathematica
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.

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

This animates the result using *Dynamic*.

```mathematica
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.*
OpenCL on the Web with webMathematica

Wolfram Research also offers webMathematica, which allows you to deploy Mathematica programs on the web by embedding them in JavaServer Pages (JSP). This allows heterogeneous computation to be performed on the server from within a client’s web browser.

There are many possible applications for this. The above shows a teaching module developed to enable students to program an OpenCL kernel without being exposed to either Mathematica syntax or host side programming. When a user clicks submit, the OpenCL kernel is compiled, an OpenCLFunction is generated, and the function is applied to an image. The computed image is then displayed on the screen for the user.

Aside from the academic applications, in some cases it is desirable to have a powerful workstation where users can invoke OpenCL computation from within the browser or mobile devices—invoking a financial computation using the latest stock data from a smart phone, for example. webMathematica is a solution for such scenarios.
**Mathematica's CUDALink Applications**

In addition to OpenCL support, *Mathematica* provides CUDA support. By providing the same usage syntax as *OpenCLLink*, *Mathematica* is unique in enabling easy porting of CUDA applications to OpenCL and vice versa. In this section, we show some of the built-in functionality in *CUDALink* as well as how to use *CUDALink* to program the GPU from within *Mathematica*.

**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 pixel operators. These are the image multiplication, division, subtraction, and addition.

To use *CUDALink*, users must first load it.

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

*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}, 
{0, 0, 0}, 
{-1, 0, 1}}
```

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

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

```
CUDAImageMultiply[...], 
...
```

Multiplication of two images.
Finally, morphology operations are supported. Here we use the `Manipulate` function, which makes creating GUI interfaces simple.

```mathematica
Manipulate[CUDAerosion[ , radius], {radius, 0, 9}]
```

Construction of an interface that performs a morphological operation on an image with varying radii.

**Fast Fourier Transforms and Linear Algebra**

Users can perform Fourier transforms and various linear algebra operations with `CUDALink`. Methods such as matrix-matrix multiplication, matrix-vector multiplication, finding minimum and maximum elements, and transposing matrices are all accelerated to use the GPU.

Here, we multiply two matrices together.

```mathematica
CUDAOT[{{1, 1, 1, 1}, {2, 2, 2, 2}, {3, 3, 3, 3}, {4, 4, 4, 4}, {5, 5, 5, 5}}, {{1, 1, 1, 1}, {2, 2, 2, 2}, {3, 3, 3, 3}, {4, 4, 4, 4}, {5, 5, 5, 5}}] // MatrixForm
```

Performing matrix multiplication.

**Financial Engineering**

`CUDALink` has built-in financial options pricing capabilities, which use either the analytic solution, the binomial solution, or Monte Carlo methods depending on the type of option selected. The following shows the American put option’s surface plot as the spot price and expiry vary.
A three-dimensional plot of the CUDA-evaluated American put. In this case, we utilize parallel programming over CPUs in addition to that provided by the GPU.

**Complex Dynamics**

CUDALink enables you to easily investigate computationally intensive complex dynamics structures. We will compute the Julia set, which is a generalization of the Mandelbrot set. The following implements the CUDA kernel.

```mathematica
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 = log(0.1f + sqrt(x*x + y*y));
        set[xIndex + yIndex*width] = c;
    }
}"
```

This loads the CUDAFunction. Notice that the syntax is the same as OpenCLFunctionLoad. While we did not show macros being used in OpenCLLink, macros are used here to allow the compiler to further optimize the code.

```mathematica
JuliaCalculate = CUDAFunctionLoad[code, "julia_kernel", {(_Real, "Output"),
    _Integer, _Integer, _Real, _Real}, {16, 16}, "Defines" ->
    {"MAX_ITERATIONS" -> 10, "ZOOM_LEVEL" -> 0.0050, "BAILOUT" -> "4.0"}];
```

The width and height are set and the output memory is allocated.

```mathematica
{width, height} = {512, 512};
jset = CUDA Memory Allocate[{Real, {height, width}}];
```
This creates an interface using Manipulate and ReliefPlot where you can adjust the value of the constant $c$ interactively.

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

Interactive computation and rendering of a Julia set.

Brownian Motion

Brownian motion is a very important concept in many scientific fields. It is used in computational chemistry, physics, and finance. In the following code, we use the CURAND kernel library to generate random numbers for computing sample paths of Brownian motion.

code = "
#include "curand_kernel.h"
extern "C" __global__ void
brownianMotion(Real_t *out, mint pathLen, mint pathN) {
    curandState rngState;
    Real_t sum = 0;
    int index = threadIdx.x + blockIdx.x*blockDim.x;
    curand_init(1234, index, 0, &rngState);
    if (index < pathN) {
        out[index] = sum;
        for (int ii = 1; ii < pathLen; ii++) {
            sum += curand_normal(&rngState);
            out[ii*pathN + index] = sum;
        }
    }
}
"

The following loads the above CUDA code into Mathematica.

cudaBM = CUDAFunctionLoad[code, "brownianMotion",
{(_Real, "Output"), _Integer, _Integer}, 64, "UnmangleCode" \to False];

The following sets the function parameters. We use a low path length and path number to make it easy to see the motion path.
pathLen = 1024;
pathN = 16;
out = ConstantArray[0, {pathLen, pathN}];

The following visualizes the result.

res = Transpose[First[cudaBM[out, pathLen, pathN, pathN]]];
ListLinePlot[res]

The possibilities are open for more complicated and broader applications of the GPU capabilities in Mathematica. And, with little effort, programs can be written so that they execute on either the CPU or GPU depending on the detected hardware.

**The Mathematica Advantage**

If you were to combine the performance computing aspects in Mathematica with the following Mathematica features, you could develop non-trivial heterogeneous programs intuitively.

**Free-Form Linguistic Input**

Mathematica is unique in providing an avenue for users to write programs in plain English. Mathematica uses Wolfram|Alpha to interpret the result, giving the output and the interpreted Mathematica program to get the output.

plot of sin x y

Plot3D[y * Sin[x], {x, -6.3, 6.3}, {y, -1, 1}]

**Simple Interface Creation**

Mathematica makes it simple to create interactive user interfaces. The interfaces can be used to experiment with parameter values, as teaching modules, or deployed using Wolfram’s Computable Document Format™.
The following creates an interface that allows users to adjust the radius and threshold parameters for the Canny edge detector.

\[
\text{Manipulate} \left[ \text{EdgeDetect}[\text{img}, r, t], \right. \\
\left. \{(r, 2, "radius"), 1, 10\}, \{(t, .1, "threshold"), 0, .5\} \right]
\]

**Broad Field Coverage**

By using both the CPU and the GPU, and making them available to the user, *Mathematica* embodies the heterogeneous message. Users have written code that uses both the CPU and GPU concurrently on multiple machines to solve tasks in computer vision, medical imaging, mathematics, and physics.

Since *Mathematica* has broad field coverage, a reference implementation is likely to exist. This makes benchmarking and testing simple. This, for example, finds all lines in an input image.

\[
\text{img} = \text{Import["example.png"]}; \\
\text{lines} = \text{ImageLines}[\text{EdgeDetect}[\text{img}, .28, .06]]; \\
\text{Show}[\text{img}, \text{Graphics}[\{\text{Thick, Red, Line/@lines}\}]]
\]
Here is another example that computes the discrete wavelet transform of an image.

\[
dwd = \text{DiscreteWaveletTransform}[\text{image}, \text{Automatic}, 2];
\]

We can plot the wavelet decomposition as an image pyramid.

\[
\text{WaveletImagePlot}[dwd]
\]

**Import/Export**

*Mathematica* has extensive support for importing and exporting data from hundreds of formats. These formats include PNG and JPEG for images, LaTeX and EPS for typesetting, and XLS and CSV for spreadsheet data.

For example, the following imports the dataset from a GRIB file. This file format is common in meteorology to store historical and forecast weather data.

\[
data = \text{Import["ExampleData/temperature.grb", \{"Datasets", "Temperature", 1\}]};
\]

This renders the dataset as an image.

\[
\text{Colorize[ImageAdjust[Image[Reverse[data]]],}
\text{ColorFunction \rightarrow \"ThermometerColors\"]}
\]

**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 using the Wolfram Runtime Library.

**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.

```mathematica
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.

```mathematica
addOne[3]
4
```

CUDAlink and OpenCLLink are written using LibraryLink and thus are prime examples of LibraryLink's capabilities.

**Symbolic C Code**

Using Mathematica's symbolic capabilities, users can generate C programs within Mathematica. The example presented here creates macros for common math constants and manipulates the expression to convert the macros to constant declarations. To use Mathematica's symbolic C code generation capabilities, you first need to import the SymbolicC package.

```mathematica
Needs["SymbolicC"]
```

This gets all constants in the Mathematica system context and uses SymbolicC's CDefine to declare a C macro.

```mathematica
s = Map[CDefine[ToString[#], N[#]] & , Map[ToExpression, 
    Select[Names["System\star"], MemberQ[Attributes[\#, Constant] &]]]
{CDefine[Catalan, 0.15965594177219],
 CDefine[Degree, 0.017453292519943295],
 CDefine[E, 2.71828],
 CDefine[EulerGamma, 0.5772156649015329],
 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 a C string using the ToCCodeString function.

```mathematica
ToCCodeString[s]
```

```c
#define Catalan  0.15965594177219
#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, users can manipulate it using standard Mathematica techniques. Here, we convert all the macros to constant values.
s = ReplaceAll[s,
CDeclare["name", val_] -> CDeclare["const", double], CAssign[name, val]]
{CDeclare["const", double], CAssign[Catalan, 0.915966]],
CDeclare["const", double], CAssign[Degree, 0.0174533]],
CDeclare["const", double], CAssign[E, 2.71828]],
CDeclare["const", double], CAssign[EulerGamma, 0.577216]],
CDeclare["const", double], CAssign[Glaisher, 1.28243]],
CDeclare["const", double], CAssign[GoldenRatio, 1.61803]],
CDeclare["const", double], CAssign[Khinchin, 2.68545]],
CDeclare["const", double], CAssign[MachinePrecision, 15.9546]],
CDeclare["const", double], CAssign[Pi, 3.14159]]

Again, the code can be converted to a C string using ToCCodeString.

ToCCodeString[s]
const double change = 0.915965594177219;
const double change = 0.017453292519943295;
const double change = 2.718281828459045;
const double change = 0.5772156649015329;
const double change = 1.2824271291006226;
const double change = 1.6180339887499895;
const double change = 2.6854520010653062;
const double change = 15.954589770191003;
const double change = 3.141592653589793;

Using Mathematica’s symbolic code generation tools, you can easily write domain-specific languages that facilitate meta-programming—programs that write other programs.

Scalability
Mathematica programs from low-end netbooks to high-end workstations and clusters. Through our support of all GPU cards and automatic floating point precision detection, Mathematica facilitates scalable GPU programming.

Multi-GPU programming, for example, is as simple as wrapping Parallelize around a GPU function. The following performs an image morphological operation on the GPU using all four GPU cards installed on a system.

GraphicsGrid[Partition[Parallelize[
Table[CUDAImageFilter[img, 5, "Device" -> $KernelID], {img, Flatten[Permutations[{{1, 2, 3, 4}, {1, 2}, {1, 2}, {1, 2}}, 2]]}], 8]]

Wolfram Research also has many technological offerings that make scaling upward and downward possible. The Mathematica licensing is also adaptive, allowing users to choose the most convenient cost-effective plan for their needs.
Summary

*Mathematica* provides several key built-in technologies that allow for easy transitioning to using heterogeneous-based computing. As proof of the simplicity, in the past few pages we wrote a dozen heterogeneous programs in diverse fields that would have been difficult to do in any other system.

*Mathematica*’s advantage lies in being able to provide all these features built into the product, having them be portable across operating systems, providing an intuitive interface for their use through careful functionality design, and making them scalable to low-end and high-end systems.

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

![Mathematica](image)

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

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

Learn more about OpenCL programming in *Mathematica*:

<table>
<thead>
<tr>
<th>Region</th>
<th>Phone</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>U.S. and Canada</td>
<td>1-800-WOLFRAM (965-3726)</td>
<td><a href="mailto:info@wolfram.com">info@wolfram.com</a></td>
</tr>
<tr>
<td></td>
<td>Europe: +44-(0)1993-883400</td>
<td><a href="mailto:info@wolfram.co.uk">info@wolfram.co.uk</a></td>
</tr>
<tr>
<td>Outside U.S. and Canada</td>
<td>+1-217-398-0700</td>
<td><a href="mailto:info@wolfram.com">info@wolfram.com</a></td>
</tr>
<tr>
<td></td>
<td>Asia: +81-(0)3-3518-2880</td>
<td><a href="mailto:info@wolfram.co.jp">info@wolfram.co.jp</a></td>
</tr>
</tbody>
</table>