WOLFRAM WHITE PAPER

Heterogeneous Programming with the Wolfram Language

WOLFRAM

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. Since 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 like the Wolfram Language, which makes heterogeneous computing easy.

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

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

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 area packages, 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 has the ability to perform computations on the GPU and thus facilitates heterogeneous computing. For developers, this new 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.

Free-Form Linguistic Input

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

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

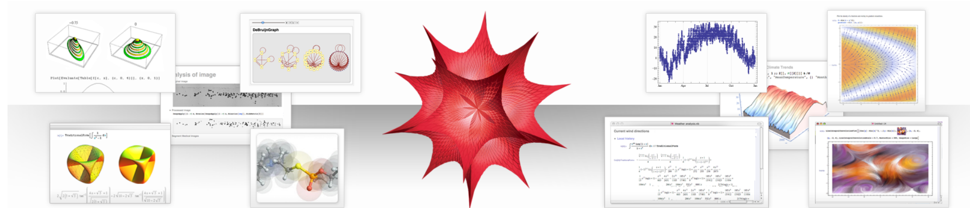
The principle behind the Wolfram Language is full integration of symbolic and numeric computing capabilities. Through its full automation and preprocessing mechanisms, users reap the power of a hybrid computing system without needing knowledge of specific methodologies and algorithms.

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.

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 Wolfram Language functions extremely flexible, streamlined, and consistent.



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.

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.

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.

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.

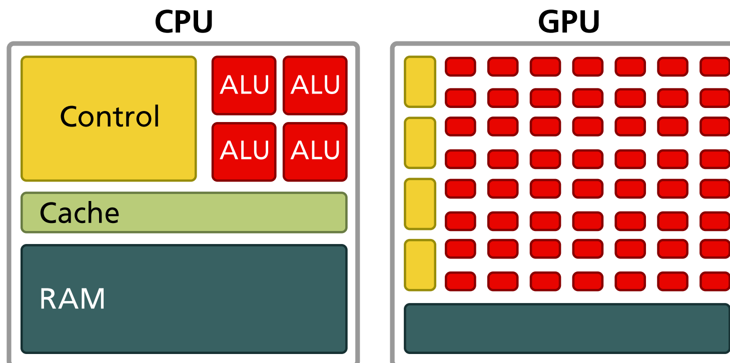
Scalability

Wolfram Research's gridMathematica allows Wolfram Language programs to be parallelized on many machines in cluster or grid configuration. Also available is webMathematica, which allows Wolfram Language programs to be run on a web server.

Motivations for Heterogeneous Computing in the Wolfram Language

Over the past few years, multicore systems have transitioned from being found only on specialty devices to commodity devices. With CPUs routinely having 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.

By providing an interface for running programs on both the CPU and the GPU, the Wolfram Language embodies the heterogeneous computing philosophy, allowing users to choose either the CPU or GPU based on the program's performance.

CPU Multicore Integration in the Wolfram Language

Most of the built-in functions, like linear algebra, image processing, and wavelets, already make use of all cores on the system. The Wolfram Language also provides the user with 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 gridMathematica as well as be used in conjunction with the GPU capabilities in the Wolfram Language.

Compile

The Wolfram Language'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 the Wolfram Language's internal virtual machine—Wolfram Virtual Machine (WVM), or C. `Compile` is used internally by many Wolfram Language 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.

Compile Command	**Compile** Instructions	**Compile** Targets
`Compile [{x}, Sin[x] + x^2 - 1/(x^2) ]`	`Instruction[Sin, r1, a1]` `Instruction[Power, r2, a1]` `Instruction[Recip, r3, r2]` `Instruction[Sub, {a1, r3}]` `Instruction[Plus, res, {r1, r2}]`	`Compile[{x}, Sin[x] + x^2 - 1/(x^2), CompilationTarget -> "C" ]`
The Wolfram Language's **Compile** command translates Wolfram Language 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 precisions and tracks when overflows and underflows occur.	The Wolfram Language byte code is a machine-independent representation of arithmetic scalar instructions as well as list manipulation instructions. The list instructions represent common functions such as **Map** and **Fold**. The instructions are simple unary, binary, or ternary instructions that are inert, allowing the user to further manipulate them symbolically.	Code can be written to target the Wolfram Language instruction set to the Wolfram Virtual Machine (an internal virtual machine bundled with the Wolfram Language), C, LLVM IR, or CUDA and OpenCL. The Wolfram Language has built-in support to target either the Wolfram Virtual Machine or C, allowing the user to write code in a high-level language that is capable of running at the same speed as C.

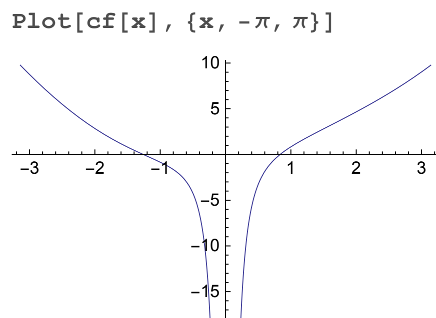
To use the `Compile` command, the user passes in the input variables along with a sequence of Wolfram Language expressions. `Compile` will automatically perform code optimization—the following code, for example, performs common subexpression elimination resulting in evaluating x^2 once:

```
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 Wolfram Language function. It can be visualized as follows:

```
Plot[cf[x], {x, -π, π}]
```



The Compile statement generates byte code from Wolfram Language expressions. 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 the Wolfram Language.

Compilation target

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

```
cf = Compile[{x}, Sin[x] + x^2 - 1/x^2, CompilationTarget → "C"]
```

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

Since the Wolfram Language is highly expressive, programs that take dozens or hundreds of lines in C code can be written in two or three lines in the Wolfram Language 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 the Wolfram Language:

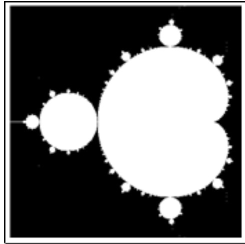
```
compileMandelbrot = Compile[{{c, _Complex}},
   Boole[Norm[
      FixedPoint[#^2 + c &, 0, 1000, SameTest → (Norm[#] ≥ 4 &)]] ≥ 4],
   CompilationTarget → "C", RuntimeAttributes → Listable
  ];
```

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:

```
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

The Wolfram Language can 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, 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:

```
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;
     ], {ii, Length[centers]}
    ];
    res
   ], CompilationTarget -> "C", RuntimeAttributes -> Listable
  ];
```

Here, we define a Wolfram Language function that calls the preceding compiled function:

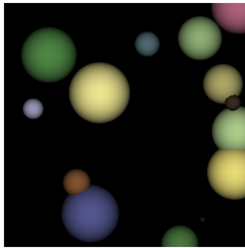
```
rayTraceCompile[centers_, radii_, colors_, x_, y_] :=
 Module[{ xx, yy},
  xx = Transpose[ConstantArray[x, Length[y]]];
  yy = ConstantArray[y, Length[x]];
  raySpheresIntersectionColor[centers, radii, colors, xx, yy]
 ]
```

This constructs the scene, generating 100 spheres in random positions and creating the ray origins. We use the Wolfram Language symbol `ColorData` to get colors that follow the "`DarkBands`" color scheme:

```
width = height = 300;
nx = ny = 300;
numSpheres = 100;
centers = Table[{RandomReal[{-width/2, width/2}],
    RandomReal[{-height/2, height/2}], RandomReal[100]}, {numSpheres}];
radii = RandomReal[{5, 30}, numSpheres];
colors = ReplaceAll[ColorData["DarkBands"] /@
    Range[0, 1, 1/numSpheres], RGBColor → List];
x = width * (N[Range[0, nx]] / nx - .5);
y = height * (N[Range[0, ny]] / ny - .5);
```

This runs the ray tracer and visualizes the result:

```
imgc = rayTraceCompile[centers, radii, colors, x, y];
Image[imgc]
```



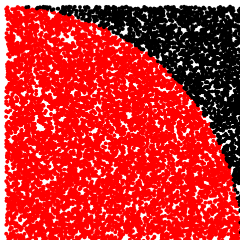
Again, the speed of this program is comparable to a program written in C, but unlike the C version, our code is around 20 lines long.

Parallel Computing

Built into the Wolfram Language is the capability for multicore computing. The Wolfram Language'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. The Wolfram Language'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 cover approximately a quarter of a unit circle:

```
pts = RandomReal[1, {10 000, 2}];
Graphics[
 {AbsolutePointSize[2], {Red, Point[Select[pts, Norm[#] ≤ 1 &]]},
  {Black, Point[Select[pts, Norm[#] > 1 &]]}}]
```



The Wolfram Language has extensive support for list processing. This makes writing the preceding Monte Carlo π approximation trivial:

```
n = 1 000 000;
Mean[Table[If[Norm[RandomReal[1, {2}]] ≤ 1, 1.0, 0.0], {n}]]

0.784838
```

The Wolfram Language has parallel primitives such as `ParallelMap` and `ParallelTable`, as well as `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:

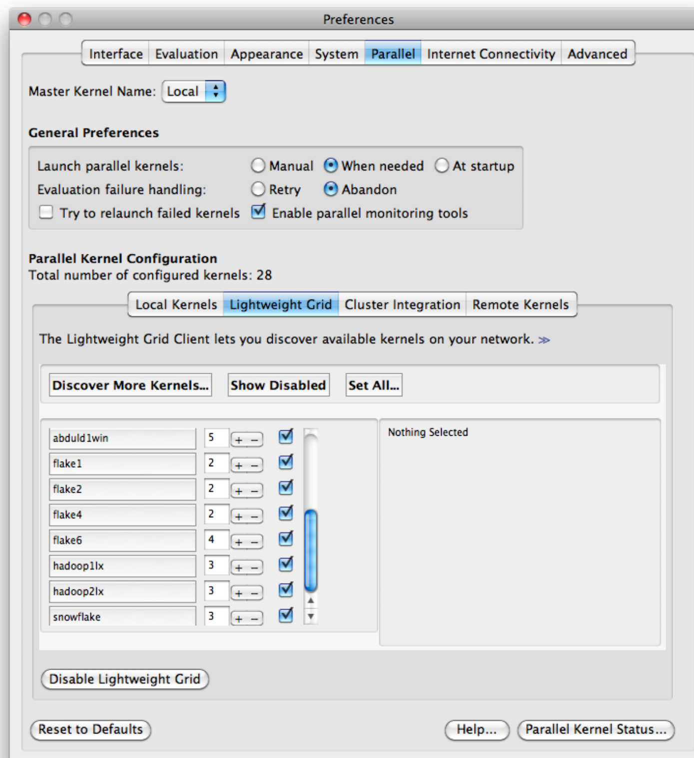
```
n = 1 000 000;
Mean[
 Parallelize[
  Table[If[Norm[RandomReal[1, {2}]] ≤ 1, 1.0, 0.0], {n}]
 ]
]

0.784934
```

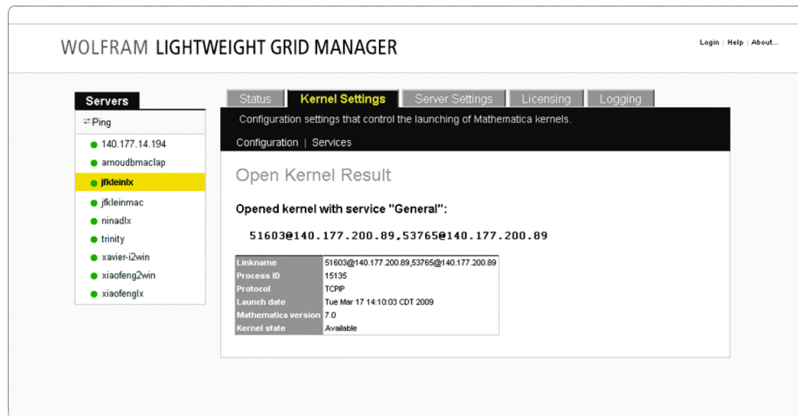
Parallel tools allow fine-grained control of how to split the parallel tasks and how many parallel kernels to run. A queueing mechanism is also available to saturate the CPU usage and achieve the best speedup.

gridMathematica

gridMathematica extends the Wolfram Language'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 the Wolfram Language's parallel tools, gridMathematica allows for fine-grained control over the scheduling of tasks, giving priority to certain kernels. gridMathematica also interfaces with existing grid management software from Microsoft, Sun, LFS, PBS, and more. It also contains its own management software called Wolfram Lightweight Grid Manager.



Programs written using the Wolfram Language's parallel tools are valid gridMathematica programs, and thus the Wolfram Language is easily scalable from single multicore systems to clusters.

GPU Integration in the Wolfram Language

CUDALink and OpenCLLink offer a high-level interface to the GPU built on top of the Wolfram Language's development technologies. They allow users to execute code on their GPU with minimal effort. Because Wolfram has fully integrated and automated the GPU's capabilities using the Wolfram Language, hiding unnecessary complexity associated with GPU programming and lowering the learning curve for GPU computation, users experience a more productive and efficient development cycle.

CUDALink and OpenCLLink Support

CUDALink is supported on all CUDA-enabled NVIDIA hardware. OpenCLLink 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. OpenCLLink also supports CPU implementations of OpenCL provided by either AMD or Intel.

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

Making GPU Programming Easy

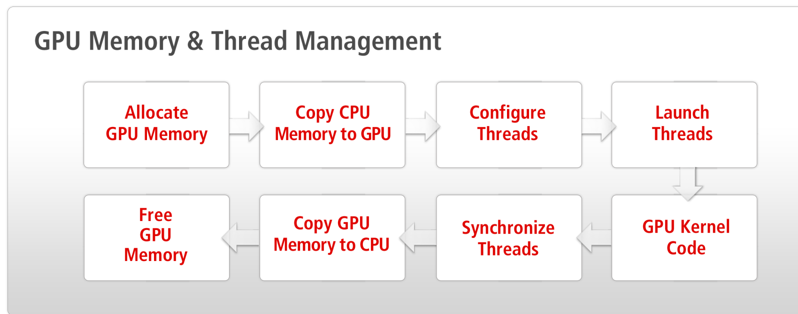
By removing repetitive and low-level GPU-related tasks, the Wolfram Language makes GPU programming simple.

Automation of development project management

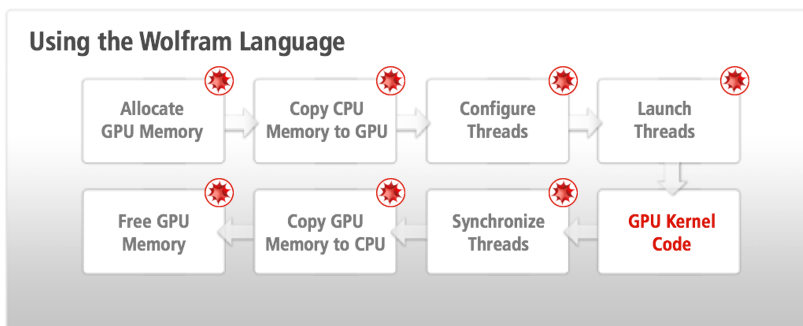
Unlike other development frameworks that require the user to manage project setup, platform dependencies, and device configuration, the Wolfram Language makes the process of GPU programming transparent and automated.

Automated GPU memory and thread management

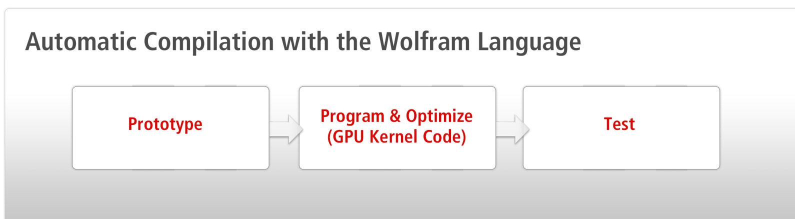
A GPU program written from scratch delegates memory and thread management to the programmer. This bookkeeping is required in addition to the need to write the GPU kernel.



With the Wolfram Language, memory and thread management are 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.



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



Integration with the Wolfram Language's built-in capabilities

The Wolfram Language's GPU integration provides full access to its 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

The Wolfram Language 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

The Wolfram Language automatically finds, configures, and makes GPU devices available to users. The Wolfram Language will automatically detect and configure the required tools to make it simple to use the GPU.

Through the Wolfram Language'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

The Wolfram Language 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.

The Wolfram Language's CUDALink

CUDALink is a built-in Wolfram Language application that provides an interface for using CUDA within the Wolfram Language. Through CUDALink, users can execute CUDA programs from the Wolfram Language with little effort. Because the Wolfram Language makes intelligent choices about GPU memory transfer, users experience better execution speed compared with handwritten CUDA programs.

Setting Up CUDALink

CUDALink supplies a set of tools to query the system's GPU hardware. To use CUDALink operations, users have to first load the CUDALink application:

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

`CUDAQ` queries whether the current hardware and system configuration support CUDALink. It will also install the required libraries from the Wolfram server on first invocation:

```
CUDAQ[]

True
```

`CUDAInformation` gives information on the available CUDA hardware:

```
CUDAInformation[]

{1 → {Name → NVS 5100M, Clock Rate → 1 210 000, Compute Capabilities → 1.2, 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 → 6,
   Core Count → 48, Execution Timeout → 1, Integrated → False, Can Map Host Memory → True,
   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, TCC Enabled → False, Total Memory → 993 460 224}}
```

CUDALink Programming

Programming the GPU in the Wolfram Language is straightforward. It begins with writing a CUDA program. The following CUDA program negates the colors of a multichannel image:

```
src = "
__global__ void cudaColorNegate(mint
    *img, mint *dim, mint channels) {
    int width = dim[0], height = dim[1];
    int xIndex = threadIdx.x + blockDim.x*blockIdx.x;
    int yIndex = threadIdx.y + blockDim.y*blockIdx.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];
    }
}";
```

The source code is passed to `CUDAFunctionLoad` and the user gets a Wolfram Language function as output:

```
CUDAColorNegate = OpenCLFunctionLoad[src, "cudaColorNegate",
  {{_Integer, "InputOutput"},
   {_Integer, "Input"}, _Integer}, {16, 16}]
```

```
CUDAFunction[<>, cudaColorNegate,
 {{_Integer, InputOutput}, {_Integer, Input}, _Integer}]
```

Now you can apply this new CUDA function to an image:

img = ;

```
CUDAColorNegate[img, ImageDimensions[img], ImageChannels[img]]
```

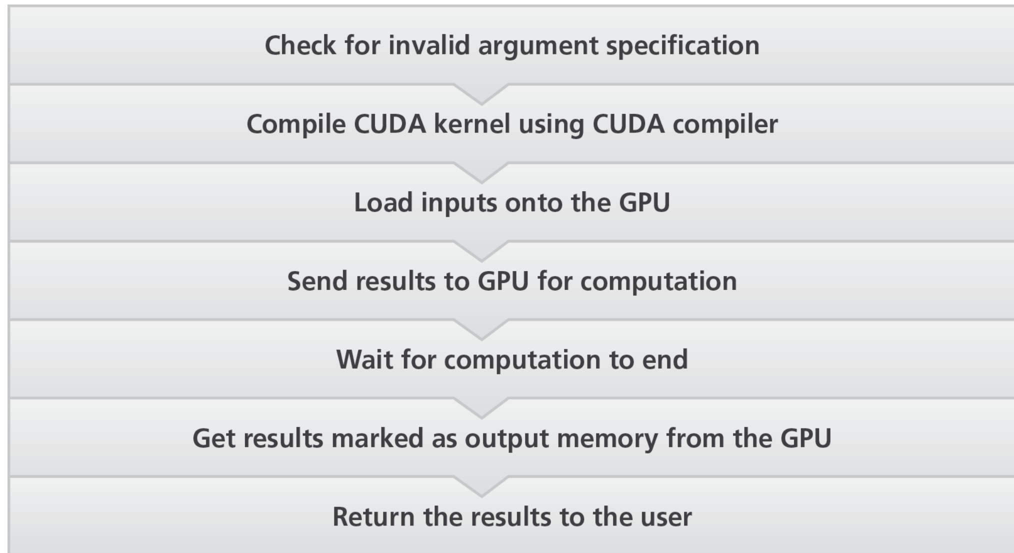
{ }

CUDAFunctionLoad follows a simple syntax where the first argument is the CUDA 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 workgroup size (block dimension):

```
src = "__global__ void kernel(mint * in, mint len) {...}";

CUDAFunctionLoad[src, "kernel", {{_Integer}, _Integer}, blockDim]
```

Several things need to happen behind the scenes at this stage to load the CUDA code into the Wolfram Language efficiently.



First, we need to check to see if the arguments to CUDAFunctionLoad are valid. Since this is the most common source of errors, we have to catch it early. Second, we compile the CUDA function and cache it. The output from this step is a CUDAFunction that behaves like any Wolfram Language function.

When a CUDAFunction 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.

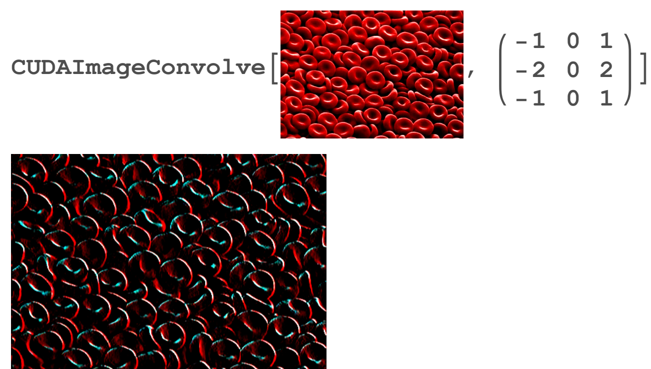
CUDALink Applications in the Wolfram Language

In this section we detail some CUDALink applications. CUDALink comes with a dozen or so built-in functions that do not require the user to know CUDA programming. The built-in functions mirror the Wolfram Language's functions, so they are easy to learn and use.

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 image multiplication, division, subtraction, and addition.

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:

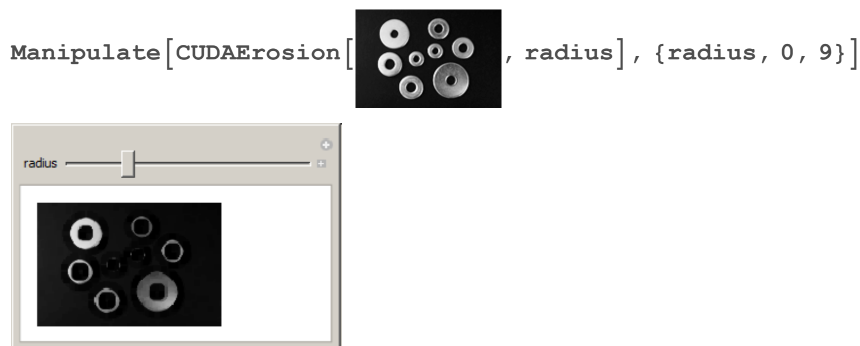


Convolving a microscopic image with a Sobel mask to detect edges.

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

Multiplication of two images.

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



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:

$$\text{CUDADot}\left[\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 4 \\ 5 & 5 & 5 & 5 & 5 \end{pmatrix}, \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 4 \\ 5 & 5 & 5 & 5 & 5 \end{pmatrix}\right] \text{ // MatrixForm}$$

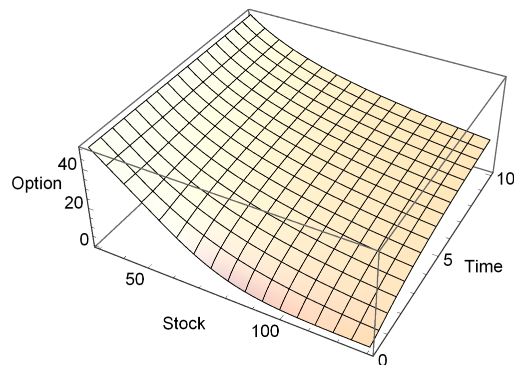
$$\begin{pmatrix} 15 & 15 & 15 & 15 & 15 \\ 30 & 30 & 30 & 30 & 30 \\ 45 & 45 & 45 & 45 & 45 \\ 60 & 60 & 60 & 60 & 60 \\ 75 & 75 & 75 & 75 & 75 \end{pmatrix}$$

Performing matrix multiplication.

Financial Engineering

CUDALink has built-in financial options pricing capabilities, which use 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:

```
ListPlot3D[ParallelMap[CUDAFinancialDerivative[{"American", "Put"},
    {"StrikePrice" → 80., "Barriers" → 100, "Expiration" → #},
    {"CurrentPrice" → Range[30., 130., 1], "InterestRate" → 0.06,
     "Volatility" → 0.45, "Dividend" → 0.02, "Rebate" → 5.}] &,
  Range[0.2, 10, 0.2]], DataRange → {{30, 130}, {0.2, 10}},
 AxesLabel → {"Stock", "Time", "Option"}]
```



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:

```
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:

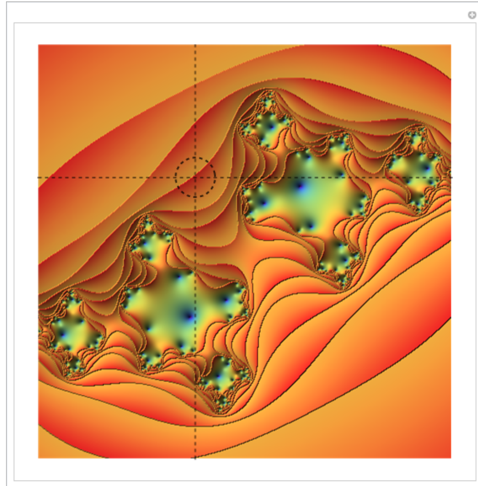
```
JuliaCalculate = CUDAFunctionLoad[code, "julia_kernel",
   {{_Real, "Output"}, _Integer, _Integer, _Real, _Real},
   {16, 16}, "Defines" → {"MAX_ITERATIONS" → 10,
     "ZOOM_LEVEL" → "0.0050", "BAILOUT" → "4.0"}];
```

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

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

This creates an interface using `Manipulate` and `ReliefPlot` where you can adjust the value of the constant *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.

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 preceding CUDA code into the Wolfram Language:

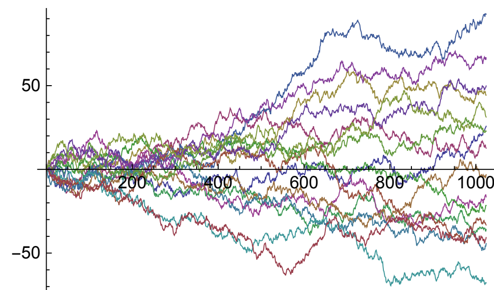
```
cudaBM = CUDAFunctionLoad[code, "brownianMotion",
   {{_Real, "Output"}, _Integer, _Integer},
   64, "UnmangleCode" → False];
```

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 the Wolfram Language. And with little effort, programs can be written so that they execute on either the CPU or GPU, depending on the detected hardware.

The Wolfram Language's OpenCLLink

In addition to CUDA support, the Wolfram Language provides OpenCL support. By providing the same usage syntax as CUDALink, the Wolfram Language is unique in enabling easy porting of CUDA applications to OpenCL and vice versa. In this section, we show how to load OpenCL programs into the Wolfram Language.

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:

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

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

```
OpenCLQ[]
True
```

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

```
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_nv_d3d9_sharing", "cl_nv_d3d10_sharing", "cl_khr_d3d10_sharing", "cl_nv_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" → 695 091 200, "Image Support" → True,
 "Maximum Read Image Arguments" → 128, "Maximum Write Image Arguments" → 8,
 "Maximum Image2D Width" → 4096, "Maximum Image2D Height" → 32 768,
 "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 Fused MAD"},
 "Global Memory Cache Type" → "Read Write", "Global Memory Cache Line Size" → 128,
 "Global Memory Cache Size" → 229 376, "Global Memory Size" → 2 780 364 800,
 "Maximum Constant Buffer Size" → 65 536, "Maximum Constant Arguments" → 9,
 "Local Memory Type" → "Local", "Local Memory Size" → 49 152, "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 the Wolfram Language using OpenCL is as straightforward as CUDA. Here is the OpenCL source code for the same color negate operation:

```
src = "
__kernel void openclColorNegate(__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];
    }
}";
```

As with CUDALink, the source code is passed to `OpenCLFunctionLoad`, and the user gets a Wolfram Language function as output:

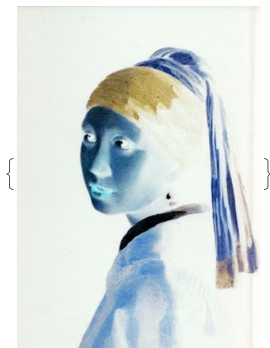
```
OpenCLColorNegate = OpenCLFunctionLoad[src, "openclColorNegate",
  {{_Integer, "InputOutput"},
   {_Integer, "Input"}, _Integer}, {16, 16}]

OpenCLFunction[<>, openclColorNegate,
 {{_Integer, InputOutput}, {_Integer, Input}, _Integer}]
```

Now the function can be used:

img = ;

```
OpenCLColorNegate[img, ImageDimensions[img], ImageChannels[img]]
```

{  }

`CUDAFunctionLoad` follows the same syntax as CUDALink, 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 workgroup size (block dimension):

```
src = "_kernel void kernel (_global mint * in, mint len) {...}";

OpenCLFunctionLoad[src, "kernel", {{_Integer}, _Integer}, blockDim]
```

As with CUDALink, behind-the-scenes processing needs to be performed to make sure that invocation of the function is efficient.

OpenCLLink Applications in the Wolfram Language

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

Black–Scholes Equation

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

```
code = "
#ifdef USING_DOUBLE_PRECISIONQ
#ifdef OPENCLLINK_USING_NVIDIA
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
#else /* OPENCLLINK_USING_NVIDIA */
#pragma OPENCL EXTENSION cl_amd_fp64 : enable
#endif /* OPENCLLINK_USING_NVIDIA */
#endif /* USING_DOUBLE_PRECISIONQ */

#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);
    }
}";
```

`Real_t` is a type defined by the Wolfram Language 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 preceding OpenCL code into the Wolfram Language:

```
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 April 2011. This data is curated by Wolfram Research and accessible via a web connection:

```
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:

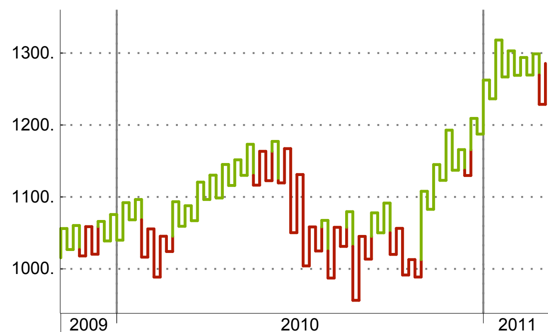
```
Needs["Calendar`"]
numberOfOptions = Length[data[[1]]];
call = ConstantArray[0.0, numberOfOptions];
S = data[[2]];
X = 1.1 * data[[2]];
T = (DaysBetween[#, {2011, 4, 20}] & /@ data[[1]]) / 365.;
R = RandomReal[{0.03, 0.07}, numberOfOptions];
Q = RandomReal[{0.01, 0.04}, numberOfOptions];
V = RandomReal[{0.10,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

The following runs the computation on the OpenCL device:

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

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

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



Computing with Data from an Excel File

The Wolfram Language supports many import and export formats. One such format is Excel (XLS), which we use in this example as the 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, __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 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 Wolfram Language 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 results. We allocate the data as single-precision 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, 0.889275, 0.991108,
 0.35102, 0.689786, 1., 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:

```
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, neighbrs;
  if (xIndex < width && yIndex < height) {
    curr = prev[index];
    for (ii = -1, neighbrs = -curr; ii <= 1; ii++) {
      if (xIndex + ii >= 0 && xIndex+ii < width) {
        for (jj = -1; jj <= 1; jj++) {
          if (yIndex+jj >= 0 && yIndex+jj < height)
            neighbrs += prev[xIndex + ii + (yIndex+jj)*width];
        }
      }
    }
    if (curr == 1)
      nxt[index] = (neighbrs == 2 || neighbrs == 3) ? 1 : 0;
    else
      nxt[index] = (neighbrs == 3) ? 1 : 0;
  }
}";
```

This loads the function using `OpenCLFunctionLoad`. We set the workgroup size to 16×16:

```
OpenCLGameOfLife =
 OpenCLFunctionLoad[src, "gol_kernel", {{_Integer, "Input"},
   {_Integer, "Output"}, _Integer, _Integer}, {16, 16}]
```

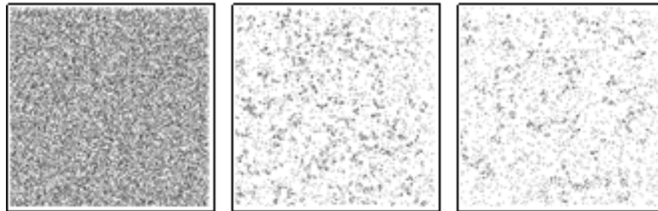
```
OpenCLFunction[<>, gol_kernel,
 {{_Integer, Input}, {_Integer, Output}, _Integer, _Integer}]
```

We set the initial state using random choice, setting 70% of the 512×512 initial states to zero, while the rest 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 Wolfram Language distribution:

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

This loads `OpenCLFunction`. Note that you can pass the vector type "`float4`" into the OpenCL program, and the Wolfram Language handles the conversion:

```
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:

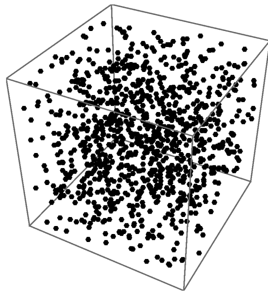
```
pos =
  OpenCLMemoryLoad[RandomReal[512, {numParticles, 4}], "Float[4]"];
vel = OpenCLMemoryLoad[RandomReal[1, {numParticles, 4}], "Float[4]"];
newPos = OpenCLMemoryAllocate["Float[4]", {numParticles, 4}];
newVel = OpenCLMemoryAllocate["Float[4]", {numParticles, 4}];
```

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];
```

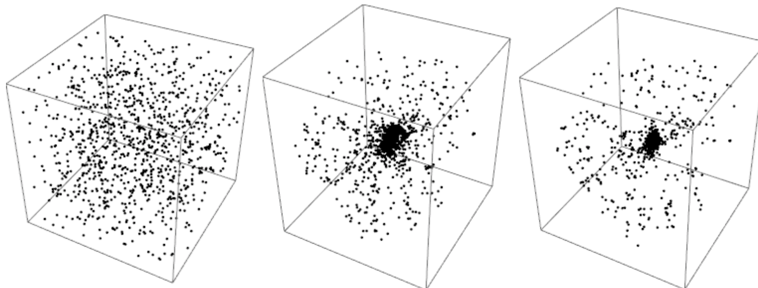
This plots the body points:

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



This animates the result using `Dynamic`:

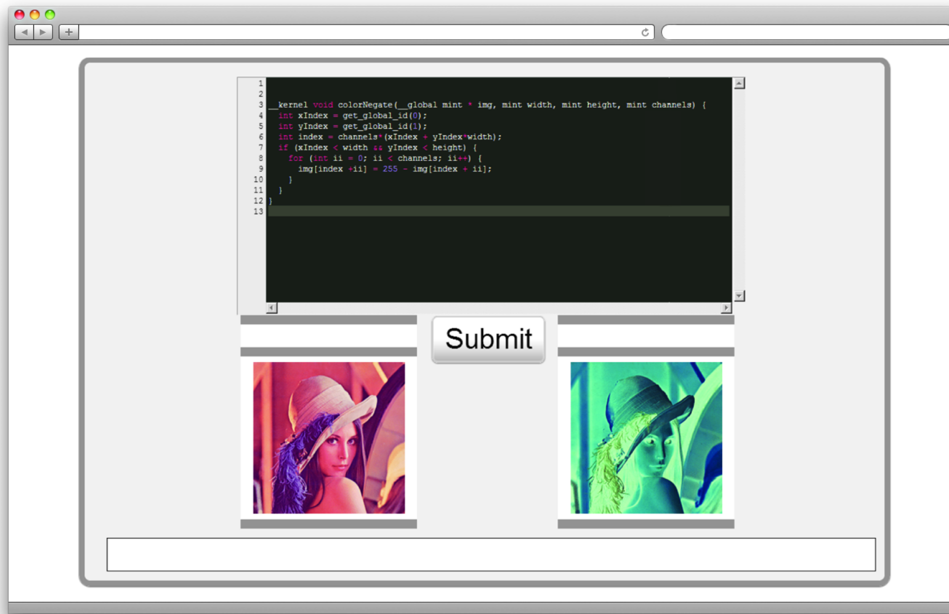
```
Graphics3D[Point[
  Dynamic[Refresh[
    NBody[pos, vel, numParticles,
     deltaT, epsSqrt, 256 * 4, newPos, newVel, 1024];
    NBody[newPos, newVel, numParticles, deltaT,
     epsSqrt, 256 * 4, pos, vel, 1024];
    Take[#, 3] & /@ OpenCLMemoryGet[pos], UpdateInterval → 0]]]]
```



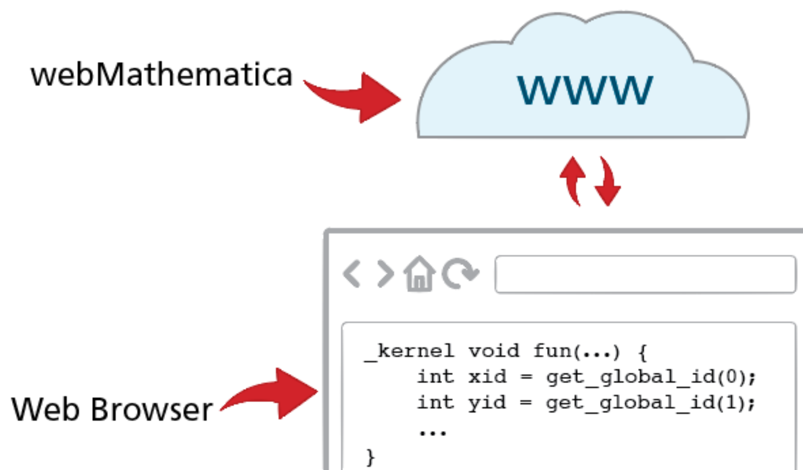
Real-time animation of the N-body simulation.

OpenCL on the Web with webMathematica

Wolfram Research also offers webMathematica, which allows you to deploy Wolfram Language 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 preceding shows a teaching module developed to enable students to program an OpenCL kernel without being exposed to either Wolfram Language 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.



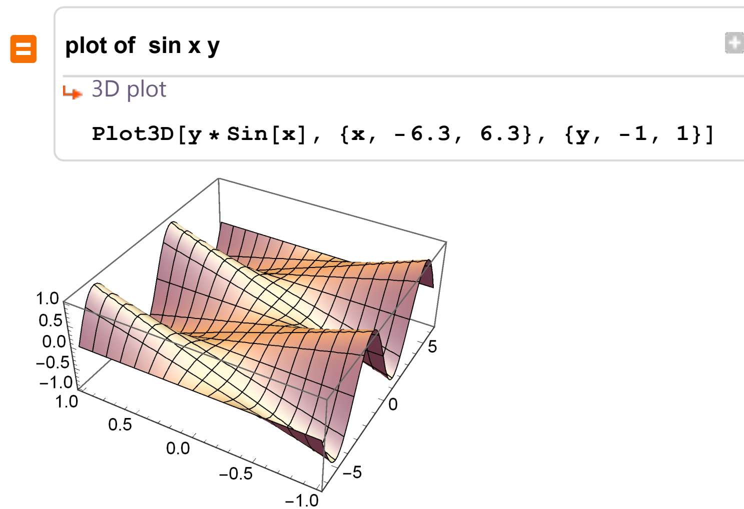
Aside from the academic applications, in some cases it is desirable to have a powerful workstation where users can invoke OpenCL computation from within the browser or mobile devices—invoking a financial computation using the latest stock data from a smart phone, for example. webMathematica is a solution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

The Wolfram Language Advantage

If you were to combine the performance computing aspects in the Wolfram Language with the following Wolfram Language features, you could develop nontrivial heterogeneous programs intuitively.

Free-Form Linguistic Input

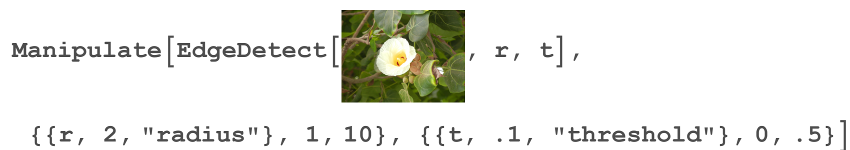
The Wolfram Language is unique in providing an avenue for users to write programs in plain English. The Wolfram Language uses Wolfram|Alpha to interpret the result, showing both the Wolfram Language input code and the corresponding output:

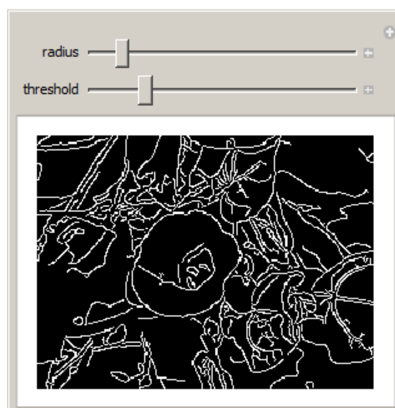


Simple Interface Creation

The Wolfram Language makes it simple to create interactive user interfaces. The interfaces can be used to experiment with parameter values or as teaching modules, or deployed using the Computable Document Format.

The following creates an interface that allows users to adjust the radius and threshold parameters for the Canny edge detector:

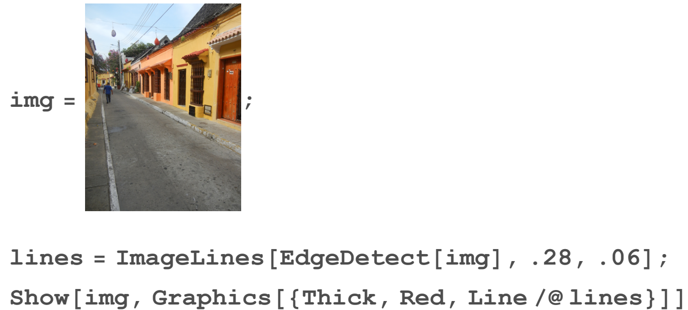




Broad Field Coverage

By using both the CPU and the GPU and making them available to the user, the Wolfram Language 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 the Wolfram Language 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:



```
img = [image];
```

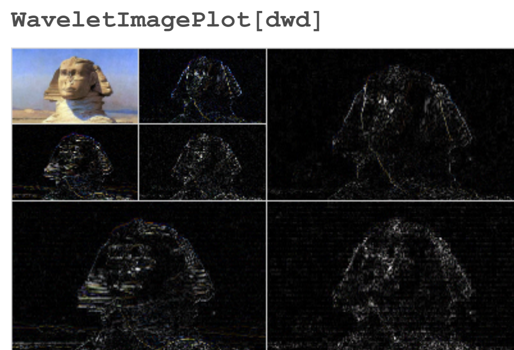
```
lines = ImageLines[EdgeDetect[img], .28, .06];
Show[img, Graphics[{Thick, Red, Line /@ lines}]]
```

Here is another example that computes the discrete wavelet transform of an image:

```
dwd = DiscreteWaveletTransform[[image], Automatic, 2];
```

We can plot the wavelet decomposition as an image pyramid:

```
WaveletImagePlot[dwd]
```



Import/Export

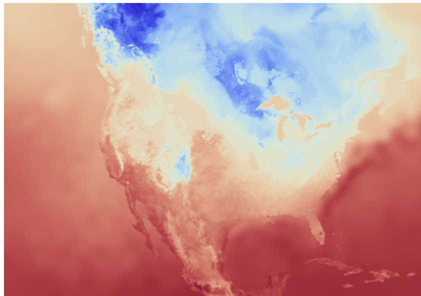
The Wolfram Language has extensive support for importing and exporting data from hundreds of formats. These formats include PNG and JPEG for images, LaTeX and EPS for typesetting, and XLS and CSV for spreadsheet data.

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

This renders the dataset as an image:

```
Colorize[ImageAdjust[Image[Reverse[data]]],
 ColorFunction → "ThermometerColors"]
```



C Code Generation

The Wolfram Language has the ability to export expressions written using `Compile` to a C file. The C file can then be compiled and either run as a Wolfram Language command (for native speed) or integrated with an external application using the Wolfram Runtime Library.

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

Symbolic C Code

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

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

This gets all constants in the Wolfram Language context and uses SymbolicC'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 a C string 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, users can manipulate it using standard Wolfram Language techniques. Here, we convert all the macros to constant values:

```
s = ReplaceAll[s, CDefine[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 Catalan = 0.915965594177219;
const double Degree = 0.017453292519943295;
const double E = 2.718281828459045;
const double EulerGamma = 0.5772156649015329;
const double Glaisher = 1.2824271291006226;
const double GoldenRatio = 1.618033988749895;
const double Khinchin = 2.6854520010653062;
const double MachinePrecision = 15.954589770191003;
const double Pi = 3.141592653589793;
```

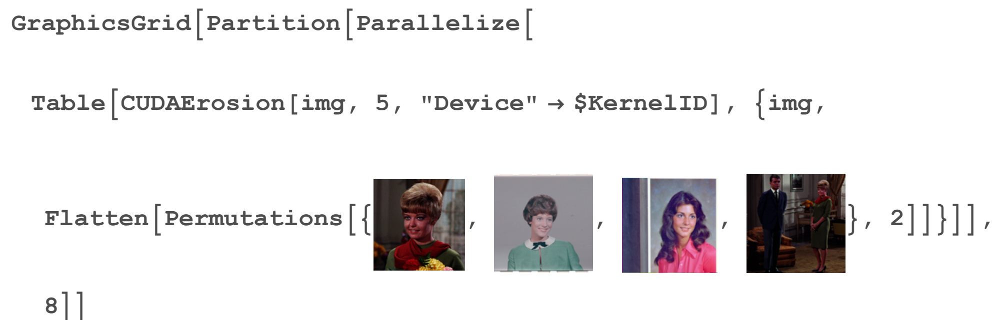
Using the Wolfram Language's symbolic code generation tools, you can easily write domain-specific languages that facilitate meta-programming—programs that write other programs.

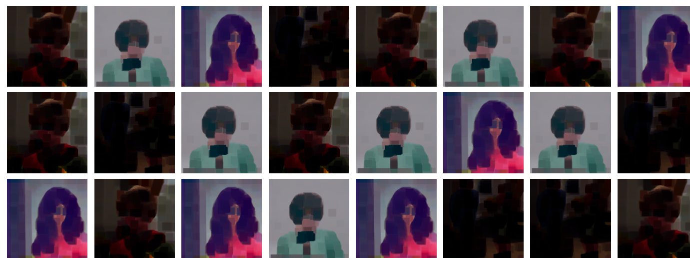
Scalability

The Wolfram Language programs everything from low-end netbooks to high-end workstations and clusters. Through our support of all GPU cards and automatic floating-point precision detection, the Wolfram Language 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[CUDAErosion[img, 5, "Device" → $KernelID], {img,
   Flatten[Permutations[{ , , , }, 2]]}]],
   8]]
```





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

Summary

The Wolfram Language 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.

The Wolfram Language'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/how-to-buy

Recommended Next Steps

Try the Wolfram Language in Mathematica for free:
www.wolfram.com/mathematica/trial

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

Learn more about heterogeneous computing with the Wolfram Language:

US and Canada:
1-800-WOLFRAM (965-3726)
info@wolfram.com

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

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

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

© The Wolfram Companies. Trademarks: Mathematica, Wolfram Language, Wolfram Knowledgebase, Wolfram Workbench, Wolfram CDF Player, gridMathematica, webMathematica, Wolfram Lightweight Grid, Computable Document Format, Wolfram Symbolic Transfer Protocol, Wolfram System, Wolfram|Alpha. MKT 1237 TCS-262 0204.BW