Introduction

CUDALink allows *Mathematica* to use the CUDA parallel computing architecture on Graphical Processing Units (GPUs). It contains functions that use CUDA-enabled GPUs to boost performance in a number of areas, such as linear algebra, financial simulation, and image processing. *CUDALink* also integrates CUDA with existing *Mathematica* development tools, allowing a high degree of automation and control.

Getting Started

To use any *CUDALink* functions, the application has to be loaded.

```
In[1]:= Needs["CUDALink`"]
```

`CUDAQ` tells you whether a CUDA-capable device is available and can be used.

```
In[2]:= CUDAQ[]

Out[2]= True
```

If `CUDAQ` returns `False`, then *CUDALink* will not work. For more information, read "*CUDALink* Setup".

`CUDAInformation` tells you more information about the graphics processing unit. Here the GPU is a Quadro FX unit with 96 cores and 1GB of graphics memory.

```
In[3]:= CUDAInformation[]

Out[3]=

{1 → {Name → Quadro FX 2800M, Clock Rate → 1 500 000, Compute Capabilities → 1.1,
   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 → 8192, Texture Alignment → 256,
   Multiprocessor Count → 12, Core Count → 96, Execution Timeout → 1,
   Integrated → False, Can Map Host Memory → False, Compute Mode → Default,
   Texture1D Width → 8192, Texture2D Width → 65 536, Texture2D Height → 32 768,
   Texture3D Width → 2048, Texture3D Height → 2048, Texture3D Depth → 2048,
   Texture2D Array Width → 8192, Texture2D Array Height → 8192,
   Texture2D Array Slices → 512, Surface Alignment → 256,
   Concurrent Kernels → False, ECC Enabled → False, Total Memory → 1 048 117 248}}
```

In *Mathematica* you can make matrices of real random numbers with the `RandomReal` command.

```
In[4]:= randM = RandomReal[1, {4000, 4000}];
```

You can multiply the matrix with itself, as shown below.

```
In[5]:= AbsoluteTiming[randM.randM;]

Out[5]= {3.8204592, Null}
```

You can use the graphics processor to do the multiplication using `CUDADot`.

```
In[6]:= AbsoluteTiming[CUDADot[randM, randM];]
```

```
Out[6]= {1.0840260, Null}
```

It is even faster to load the data onto the GPU with `CUDAMemoryLoad`. The result is a `CUDAMemory` expression; this can be used as a handle to the data for more computations.

```
In[7]:= randMG = CUDAMemoryLoad[randM]
```

```
Out[7]= CUDAMemory[<10116>, Double]
```

Now you can pass the `CUDAMemory` to `CUDADot`, which stores the result on the GPU and returns a new `CUDAMemory` expression.

```
In[8]:= AbsoluteTiming[res = CUDADot[randMG, randMG]]
```

```
Out[8]= {0.1357474, CUDAMemory[<16579>, Double]}
```

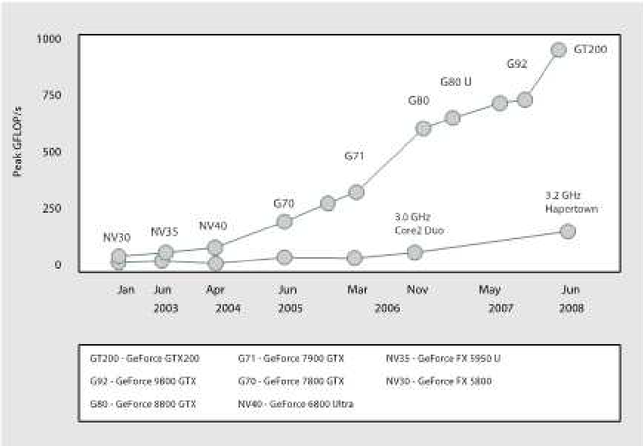
You can retrieve the data from the graphics processor with `CUDAMemoryGet`. Here, the dimensions of the result are shown to be as expected.

```
In[9]:= Dimensions[CUDAMemoryGet[res]]
```

```
Out[9]= {4000, 4000}
```

CUDA and *CUDALink*

CUDA is used as the computing engine for NVIDIA graphics processing units (GPUs); it provides a programming interface that can be called by software applications such as *Mathematica* with *CUDALink*. It allows CUDA GPUs to be used for parallel computations, allowing many concurrent threads to run. The following graph shows the performance of CPUs and GPUs—measured in billions of floating-point operations per second (GFLOP/s).



Evolution of CPU and GPU GFLOPS since 2003 (source: *NVIDIA CUDA Manual*).

CUDALink allows *Mathematica* users to call the CUDA programming layer directly; it also provides users higher-level functions, provided in a number of CUDA libraries, for solutions in areas such as high-performance core linear algebra and Fourier transforms.

CUDALink Application Areas

CUDALink provides functions in various application areas. These include carefully tuned linear algebra, discrete Fourier transforms, and image processing algorithms. This section gives an introduction to some of these applications.

Image Processing

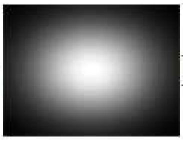
CUDALink offers many image processing algorithms that have been carefully tuned to run on GPUs. These include the binary image operations (`CUDAImageAdd`, `CUDAImageSubtract`, `CUDAImageMultiply`, and `CUDAImageDivide`), the morphology operators (`CUDAErosion`, `CUDADilation`, `CUDAOpening`, and `CUDAClosing`), and image convolution (`CUDAImageConvolve`).

To use any of the *CUDALink* functionality, you first need to include the *CUDALink* application, as shown below.

```
In[1]:= Needs["CUDALink`"]
```

Now you can apply CUDA-based image processing functions directly to images. This example carries out channel-wise multiplication of two input images using `CUDAImageMultiply`.

```
In[2]:=
```

CUDAImageMultiply[ , ]

Out[2]=

Since the CUDA functions are *Mathematica* functions, they can be used in conjunction with other functions (like `Manipulate`). In the following example, a linear interpolation is carried out with `Manipulate` being used to vary the interpolation parameter.

In[3]:=

Manipulate[CUDAImageAdd[CUDAImageMultiply[ , 1 - alpha],

CUDAImageMultiply[ , alpha]], {alpha, 0.0, 1.0, .001}]

Out[3]=

CUDA functions can also be used with *Mathematica*'s curated data, `Import` and `Export` functions, as well as its visualization functions. In the following, the core computation is done with CUDA, while using *Mathematica* for all the other functions.

In[4]:=
```
gisData = Import[ "http://exampledata.wolfram.com/sdtsdem.tar.gz", "Data" ];
ReliefPlot[CUDAImageConvolve[gisData, {{1, 0, -1}}],
 ColorFunction → "GreenBrownTerrain"]
```

Out[5]=

Fourier Analysis

The functions `CUDAFourier` and `CUDAInverseFourier` carry out Fourier transforms and inverse

Fourier transforms using CUDA.

To use any of the *CUDALink* functionality, you first need to include the *CUDALink* application, as shown below.

```
In[1]:= Needs["CUDALink`"]
```

```
In[2]:= vec = Range[1., 10];
        CUDAFourier[vec]
```

```
Out[3]= {17.3925 + 0. i, -1.58114 - 4.86624 i, -1.58114 - 2.17625 i, -1.58114 - 1.14876 i,
         -1.58114 - 0.513743 i, -1.58114 + 8.09721×10^-16 i, -1.58114 + 0.513743 i,
         -1.58114 + 1.14876 i, -1.58114 + 2.17625 i, -1.58114 + 4.86624 i}
```

If the input to a CUDA function is a `CUDAMemory` handle, the result will also be a `CUDAMemory` handle.

```
In[4]:= gpuVec = CUDAMemoryLoad[vec];
        CUDAFourier[gpuVec]
```

```
Out[5]= CUDAMemory[<7091>, ComplexDouble]
```

You can then retrieve the data from the GPU with `CUDAMemoryGet`.

```
In[6]:= CUDAMemoryGet[%]
```

```
Out[6]= {17.3925 + 0. i, -1.58114 - 4.86624 i, -1.58114 - 2.17625 i, -1.58114 - 1.14876 i,
         -1.58114 - 0.513743 i, -1.58114 + 8.09721×10^-16 i, -1.58114 + 0.513743 i,
         -1.58114 + 1.14876 i, -1.58114 + 2.17625 i, -1.58114 + 4.86624 i}
```

This general principle applies to all *CUDALink* functions. It makes it easy to test and develop, and then to work by keeping data on the GPU, which improves efficiency.

CUDALink Programming

A key feature of *CUDALink* is how it makes it easy to develop new GPU programs and integrate them into your *Mathematica* work. This requires that you have installed a C compiler.

To use any of the *CUDALink* functionality, you first need to include the *CUDALink* application, as shown below.

```
In[14]:= Needs["CUDALink`"]
```

Here, a simple CUDA function is loaded. It takes an input vector and doubles each element.

```
In[15]:= doubleFun = CUDAFunctionLoad["
          __global__ void doubleVec(mint * in,  mint length) {
              mint index = threadIdx.x + blockIdx.x * blockDim.x;

              if (index < length)
                in[index] = 2*in[index];
            }", "doubleVec", {{_Integer}, _Integer}, 256]
```

```
Out[15]= CUDAFunction[<>, doubleVec, {{_Integer}, _Integer}]
```

`CUDAFunctionLoad` requires that you have a C compiler installed. If this does not work for you, try to consult "C Compiler".

Here is an input vector that will be used to call the function.

```
In[16]:= vec = Range[20];
```

This calls the `CUDAFunction`.

```
In[17]:= doubleFun[vec, 20]
```

```
Out[17]= {{2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40}}
```

This loads data to the GPU and calls your function on the data. The result is a `CUDAMemory` expression.

```
In[5]:= gpuVec = CUDAMemoryLoad[vec];
        doubleFun[gpuVec, 20]
```

```
Out[6]= {CUDAMemory[<17648>, Integer]}
```

This retrieves the result from the GPU.

```
In[7]:= CUDAMemoryGet[First[%]]
```

```
Out[7]= {2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40}
```

You can get information on your function with `CUDAFunctionInformation`, as shown in the following.

```
In[18]:= CUDAFunctionInformation[doubleFun]
```

```
Out[18]= {Source →
    __global__ void doubleVec(mint * in,  mint length) {
        mint index = threadIdx.x + blockIdx.x * blockDim.x;

        if (index < length)
          in[index] = 2*in[index];
      }, ID → 21 362 576, BinaryFile →
      /home/usr2/dcampbell/.Mathematica/ApplicationData/CUDALink/BuildFolder/
        dcampbelllx-29298/CUDAFunction-959.cubin,
     Function → doubleVec, ArgumentTypes → {{_Integer}, _Integer},
     Options → {Platform → 1, Device → 1, Defines → ,
       SystemDefines → -DUSING_CUDA_FUNCTION=1  -Dmint=int -DReal_t=float,
       SystemIncludeDirectories → , IncludeDirectories → , CompileOptions → ,
       ShellcommandFunction → None, ShellOutputFunction → None,
       TargetPrecision → Single, BlockDimensions → 256}}
```

Related Guides

- Running Computations on GPU Using CUDA
- Running Computations on GPU Using OpenCL
- C/C++ Language Interface
- Calling C Compilers from *Mathematica*

Related Tutorials

- *CUDALink* User Guide
- Setup and Operations
- Functions
- Programming
- Memory
- Applications
- Multiple Devices
- Reference

Give Feedback