GPU Programming in MATLAB

By Jill Reese, MathWorks, and Sarah Zaranek, MathWorks

Multicore machines and hyper-threading technology have enabled scientists, engineers, and financial analysts to speed up computationally intensive applications in a variety of disciplines. Today, another type of hardware promises even higher computational performance: the graphics processing unit (GPU).

Originally used to accelerate graphics rendering, GPUs are increasingly applied to scientific calculations. Unlike a traditional CPU, which includes no more than a handful of cores, a GPU has a massively parallel array of integer and floating-point processors, as well as dedicated, high-speed memory. A typical GPU comprises hundreds of these smaller processors (Figure 1).

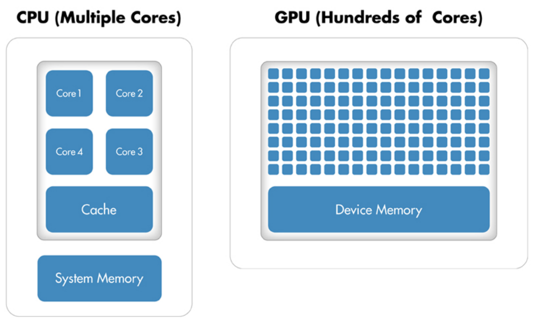


Figure 1. Comparison of the number of cores on a CPU system and a GPU.

The greatly increased throughput made possible by a GPU, however, comes at a cost. First, memory access becomes a much more likely bottleneck for your calculations. Data must be sent from the CPU to the GPU before calculation and then retrieved from it afterwards. Because a GPU is attached to the host CPU via the PCI Express bus, the memory access is slower than with a traditional CPU.[1] This means that your overall computational speedup is limited by the amount of data transfer that occurs in your algorithm. Second, programming for GPUs in C or Fortran requires a different mental model and a skill set that can be difficult and time-consuming to acquire. Additionally, you must spend time fine-tuning your code for your specific GPU to optimize your applications for peak performance.

This article demonstrates features in Parallel Computing Toolbox™ that enable you to run your MATLAB® code on a GPU by making a few simple changes to your code. We illustrate this approach by solving a second-order wave equation using spectral methods.

Why Parallelize a Wave Equation Solver?

Wave equations are used in a wide range of engineering disciplines, including seismology, fluid dynamics, acoustics, and electromagnetics, to describe sound, light, and fluid waves.

An algorithm that uses spectral methods to solve wave equations is a good candidate for parallelization because it meets both of the criteria for acceleration using the GPU (see "Will Execution on a GPU Accelerate My Application?"):

It is computationally intensive. The algorithm performs many fast Fourier transforms (FFTs) and inverse fast Fourier transforms (IFFTs). The exact number depends on the size of the grid (Figure 2) and the number of time steps included in the simulation. Each time step requires two FFTs and four IFFTs on different matrices, and a single computation can involve hundreds of thousands of time steps.

It is massively parallel. The parallel FFT algorithm is designed to "divide and conquer" so that a similar task is performed repeatedly on different data. Additionally, the algorithm requires substantial communication between processing threads and plenty of memory bandwidth. The IFFT can similarly be run in parallel.

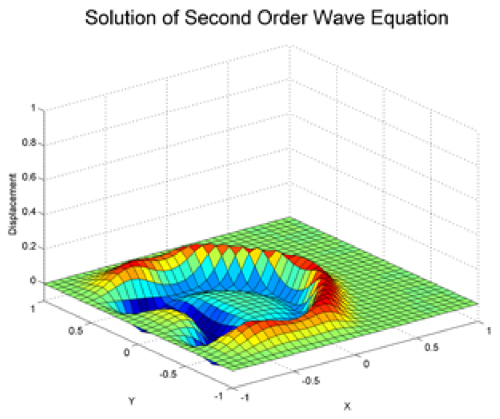


Figure 2. A solution for a second-order wave equation on a 32 x 32 grid.

Will Execution on a GPU Accelerate My Application?

A GPU can accelerate an application if it fits both of the following criteria:

Computationally intensive—The time spent on computation significantly exceeds the time spent on transferring data to and from GPU memory.

Massively parallel—The computations can be broken down into hundreds or thousands of independent units of work.

Applications that do not satisfy these criteria might actually run slower on a GPU than on a CPU.

GPU Computing in MATLAB

Before continuing with the wave equation example, let’s quickly review how MATLAB works with the GPU.

FFT, IFFT, and linear algebraic operations are among more than 100 built-in MATLAB functions that can be executed directly on the GPU by providing an input argument of the type GPUArray, a special array type provided by Parallel Computing Toolbox. These GPU-enabled functions are *overloaded*—in other words, they operate differently depending on the data type of the arguments passed to them.

For example, the following code uses an FFT algorithm to find the discrete Fourier transform of a vector of pseudorandom numbers on the CPU:

```
A = rand(2^16,1);

B = fft(A);
```

To perform the same operation on the GPU, we first use the `gpuArray` command to transfer data from the MATLAB workspace to device memory. Then we can run `fft`, which is one of the overloaded functions on that data:

```
A = gpuArray(rand(2^16,1));

B = fft(A);
```

The `fft` operation is executed on the GPU rather than the CPU since its input (a GPUArray) is held on the GPU.

The result, B, is stored on the GPU. However, it is still visible in the MATLAB workspace. By running `class(B)`, we can see that it is a GPUArray.

```
class(B)

ans =

parallel.gpu.GPUArray
```

We can continue to manipulate B on the device using GPU-enabled functions. For example, to visualize our results, the `plot` command automatically works on GPUArrays:

```
plot(B);
```

To return the data back to the local MATLAB workspace, you can use the `gather` command; for example

```
C = gather(B);
```

`C` is now a double in MATLAB and can be operated on by any of the MATLAB functions that work on doubles.

In this simple example, the time saved by executing a single FFT function is often less than the time spent transferring the vector from the MATLAB workspace to the device memory. This is generally true but is dependent on your hardware and size of the array. Data transfer overhead can become so significant that it degrades the application's overall performance, especially if you repeatedly exchange data between the CPU and GPU to execute relatively few computationally intensive operations. It is more efficient to perform several operations on the data while it is on the GPU, bringing the data back to the CPU only when required[2].

Note that GPUs, like CPUs, have finite memories. However, unlike CPUs, they do not have the ability to swap memory to and from disk. Thus, you must verify that the data you want to keep on the GPU does not exceed its memory limits, particularly when you are working with large matrices. By running `gpuDevice`, you can query your GPU card, obtaining information such as name, total memory, and available memory.

Implementing and Accelerating the Algorithm to Solve a Wave Equation in MATLAB

To put the above example into context, let's implement the GPU functionality on a real problem. Our computational goal is to solve the second-order wave equation

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$$

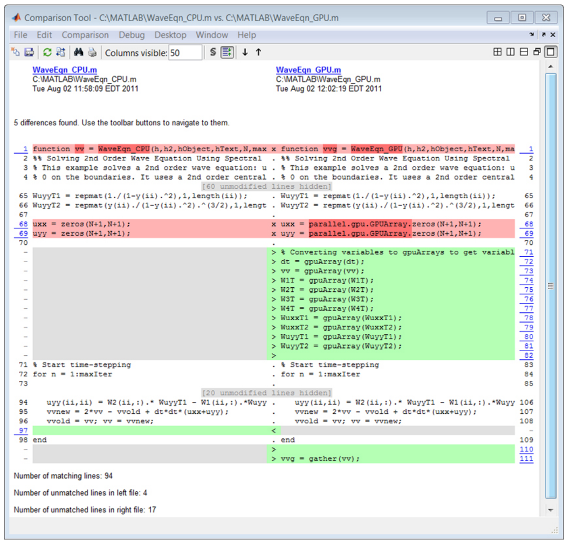
with the condition u = 0 on the boundaries. We use an algorithm based on spectral methods to solve the equation in space and a second-order central finite difference method to solve the equation in time.

Spectral methods are commonly used to solve partial differential equations. With spectral methods, the solution is approximated as a linear combination of continuous basis functions, such as sines and cosines. In this case, we apply the Chebyshev spectral method, which uses Chebyshev polynomials as the basis functions.

At every time step, we calculate the second derivative of the current solution in both the x and y dimensions using the Chebyshev spectral method. Using these derivatives together with the old solution and the current solution, we apply a second-order central difference method (also known as the leap-frog method) to calculate the new solution. We choose a time step that maintains the stability of this leap-frog method.

The MATLAB algorithm is computationally intensive, and as the number of elements in the grid over which we compute the solution grows, the time the algorithm takes to execute increases dramatically. When executed on a single CPU using a 2048 x 2048 grid, it takes more than a minute to complete just 50 time steps. Note that this time already includes the performance benefit of the inherent multithreading in MATLAB. Since R2007a, MATLAB supports multithreaded computation for a number of functions. These functions automatically execute on multiple threads without the need to explicitly specify commands to create threads in your code.

When considering how to accelerate this computation using Parallel Computing Toolbox, we will focus on the code that performs computations for each time step. Figure 3 illustrates the changes required to get the algorithm running on the GPU. Note that the computations involve MATLAB operations for which GPU-enabled overloaded functions are available through Parallel Computing Toolbox. These operations include FFT and IFFT, matrix multiplication, and various element-wise operations. As a result, we do not need to change the algorithm in any way to execute it on a GPU. We simply transfer the data to the GPU using `gpuArray` before entering the loop that computes results at each time step.


Comparison Tool - C:\MATLAB\WaveEqn_CPU.m vs. C:\MATLAB\WaveEqn_GPU.m
File
Edit
Comparison
Debug
Desktop
Window
Help
Columns visible: 50

```
WaveEqn_CPU.m                                          WaveEqn_GPU.m
C:\MATLAB\WaveEqn_CPU.m                                C:\MATLAB\WaveEqn_GPU.m
Tue Aug 02 11:58:09 EDT 2011                           Tue Aug 02 12:02:19 EDT 2011


5 differences found. Use the toolbar buttons to navigate to them.

 1 function vv = WaveEqn_CPU(h,h2,hObject,hText,N,max x function vvg = WaveEqn_GPU(h,h2,hObject,hText,N,ma   1
 2 %% Solving 2nd Order Wave Equation Using Spectral  . %% Solving 2nd Order Wave Equation Using Spectral   2
 3 % This example solves a 2nd order wave equation: u . % This example solves a 2nd order wave equation: u   3
 4 % 0 on the boundaries. It uses a 2nd order central . % 0 on the boundaries. It uses a 2nd order central   4
                              [60 unmodified lines hidden]
65 WuyyT1 = repmat(1./(1-y(ii).^2),1,length(ii));    . WuyyT1 = repmat(1./(1-y(ii).^2),1,length(ii));    65
66 WuyyT2 = repmat(y(ii)./(1-y(ii).^2).^(3/2),1,lengt . WuyyT2 = repmat(y(ii)./(1-y(ii).^2).^(3/2),1,lengt 66
67                                                    .                                                    67
68 uxx = zeros(N+1,N+1);                             x uxx = parallel.gpu.GPUArray.zeros(N+1,N+1);       68
69 uyy = zeros(N+1,N+1);                             x uyy = parallel.gpu.GPUArray.zeros(N+1,N+1);       69
70                                                    .                                                    70
 -                                                    > % Converting variables to gpuArrays to get variabl 71
 -                                                    > dt = gpuArray(dt);                                 72
 -                                                    > vv = gpuArray(vv);                                 73
 -                                                    > W1T = gpuArray(W1T);                               74
 -                                                    > W2T = gpuArray(W2T);                               75
 -                                                    > W3T = gpuArray(W3T);                               76
 -                                                    > W4T = gpuArray(W4T);                               77
 -                                                    > WuxxT1 = gpuArray(WuxxT1);                         78
 -                                                    > WuxxT2 = gpuArray(WuxxT2);                         79
 -                                                    > WuyyT1 = gpuArray(WuyyT1);                         80
 -                                                    > WuyyT2 = gpuArray(WuyyT2);                         81
 -                                                    >                                                    82
71 % Start time-stepping                              . % Start time-stepping                              83
72 for n = 1:maxIter                                  . for n = 1:maxIter                                  84
73                                                    .                                                    85
                              [20 unmodified lines hidden]
94    uyy(ii,ii) = W2(ii,:).* WuyyT1 - W1(ii,:).*Wuyy .    uyy(ii,ii) = W2(ii,:).* WuyyT1 - W1(ii,:).*Wuyy 106
95    vvnew = 2*vv - vvold + dt*dt*(uxx+uyy);         .    vvnew = 2*vv - vvold + dt*dt*(uxx+uyy);         107
96    vvold = vv; vv = vvnew;                         .    vvold = vv; vv = vvnew;                         108
97                                                    <                                                     -
98 end                                                . end                                                109
 -                                                    >                                                   110
 -                                                    > vvg = gather(vv);                                 111

Number of matching lines: 94

Number of unmatched lines in left file: 4

Number of unmatched lines in right file: 17
```

Figure 3. Code Comparison Tool showing the differences in the CPU and GPU versions of the code. The GPU and CPU versions share over 84% of their code in common (94 lines out of 111).

After the computations are performed on the GPU, we transfer the results from the GPU to the CPU. Each variable referenced by the GPU-enabled functions must be created on the GPU or transferred to the GPU before it is used.

To convert one of the weights used for spectral differentiation to a GPUArray variable, we use

```
W1T = gpuArray(W1T);
```

Certain types of arrays can be constructed directly on the GPU without our having to transfer them from the MATLAB workspace. For example, to create a matrix of zeros directly on the GPU, we use

```
uxx = parallel.gpu.GPUArray.zeros(N+1,N+1);
```

We use the `gather` function to bring data back from the GPU; for example:

```
vvg = gather(vv);
```

Note that there is a single transfer of data to the GPU, followed by a single transfer of data from the GPU. All the computations for each time step are performed on the GPU.

Comparing CPU and GPU Execution Speeds

To evaluate the benefits of using the GPU to solve second-order wave equations, we ran a benchmark study in which we measured the amount of time the algorithm took to execute 50 time steps for grid sizes of 64, 128, 512, 1024, and 2048 on an Intel® Xeon® Processor X5650 and then using an NVIDIA® Tesla™ C2050 GPU.

For a grid size of 2048, the algorithm shows a 7.5x decrease in compute time from more than a minute on the CPU to less than 10 seconds on the GPU (Figure 4). The log scale plot shows that the CPU is actually faster for small grid sizes. As the technology evolves and matures, however, GPU solutions are increasingly able to handle smaller problems, a trend that we expect to continue.

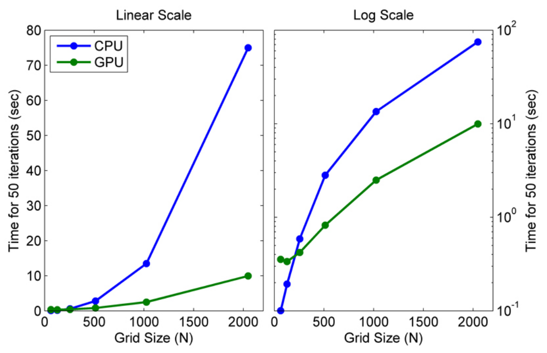


Figure 4. Plot of benchmark results showing the time required to complete 50 time steps for different grid sizes, using either a linear scale (left) or a log scale (right).

Advanced GPU Programming with MATLAB

Parallel Computing Toolbox provides a straightforward way to speed up MATLAB code by executing it on a GPU. You simply change the data type of a function's input to take advantage of the many MATLAB commands that have been overloaded for GPUArrays. (A complete list of built-in MATLAB functions that support GPUArray is available in the Parallel Computing Toolbox documentation.)

To accelerate an algorithm with multiple simple operations on a GPU, you can use `arrayfun`, which applies a function to each element of an array. Because `arrayfun` is a GPU-enabled function, you incur the memory transfer overhead only on the single call to `arrayfun`, not on each individual operation.

Finally, experienced programmers who write their own CUDA code can use the CUDAKernel interface in Parallel Computing Toolbox to integrate this code with MATLAB. The CUDAKernel interface enables even more fine-grained control to speed up portions of code that were performance bottlenecks. It creates a MATLAB object that provides access to your existing kernel compiled into PTX code (PTX is a low-level parallel thread execution instruction set). You then invoke the `feval` command to evaluate the kernel on the GPU, using MATLAB arrays as input and output.

Summary

Engineers and scientists are successfully employing GPU technology, originally intended for accelerating graphics rendering, to accelerate their discipline-specific calculations. With minimal effort and without extensive knowledge of GPUs, you can now use the promising power of GPUs with MATLAB. GPUArrays and GPU-enabled MATLAB functions help you speed up MATLAB operations without low-level CUDA programming. If you are already familiar with programming for GPUs, MATLAB also lets you integrate your existing CUDA kernels into MATLAB applications without requiring any additional C programming.

To achieve speedups with the GPUs, your application must satisfy some criteria, among them the fact that sending the data between the CPU and GPU must take less time than the performance gained by running on the GPU. If your application satisfies these criteria, it is a good candidate for the range of GPU functionality available with MATLAB.

GPU Glossary

CPU (central processing unit). The central unit in a computer responsible for calculations and for controlling or supervising other parts of the computer. The CPU performs logical and floating point operations on data held in the computer memory.

GPU (graphics processing unit). Programmable chip originally intended for graphics rendering. The highly parallel structure of a GPU makes them more effective than general-purpose CPUs for algorithms where processing of large blocks of data is done in parallel.

Core. A single independent computational unit within a CPU or GPU chip. CPU and GPU cores are not equivalent to each other; GPU cores perform *specialized operations* whereas CPU cores are designed for *general-purpose programs*.

CUDA®. A parallel computing technology from NVIDIA® that consists of a parallel computing architecture and developer tools, libraries, and programming directives for GPU computing.

Device. A hardware card containing the GPU and its associated memory.

Host. The CPU and system memory.

Kernel. Code written for execution on the GPU. Kernels are functions that can run on a large number of threads. The parallelism arises from each thread independently running the same program on different data.

References

1. See Chapter 6 (Memory Optimization) of the NVIDIA "CUDA C Best Practices" documentation for further information about potential GPU-computing bottlenecks and optimization of GPU memory access.

2. See Chapter 6 (Memory Optimization) of the NVIDIA "CUDA C Best Practices" documentation for further information about improving performance by minimizing data transfers.

Products Used

- MATLAB
- Parallel Computing Toolbox

Learn More

- Spectral Methods, Lloyd N. Trefethen
- Introduction to MATLAB GPU Computing
- Accelerating Signal Processing Algorithms with GPUs and MATLAB

See more articles and subscribe at mathworks.com/newsletters.

Published 2011
91967v01