# Table of Contents

Table of Contents ................................................................. 2  
Contacting AccelerEyes ......................................................... 3  
Chapter 1: Introduction to GPU Computing ................................. 5  
  Graphics Processors for General Purpose Computing ......................... 5  
  Important Concepts ........................................................................ 7  
  Why Jacket? .................................................................................. 8  
Chapter 2: Product Overview ..................................................... 9  
  Jacket Overview ........................................................................... 9  
  The Jacket System ......................................................................... 10  
  Jacket Applications and Benefits ..................................................... 11  
  Jacket Product Family ...................................................................... 12  
Chapter 3: Installation and Activation ........................................... 13  
  Terms of Use .................................................................................. 13  
  Pre-Requisites ............................................................................... 13  
  Jacket Installation .......................................................................... 15  
  Manual Activation ........................................................................... 16  
Chapter 4: Getting Started .......................................................... 19  
  Casting to the GPU .......................................................................... 19  
  Something Familiar – Multiplying 2 Matrices ................................. 20  
  Something Familiar – Approximating Pi ........................................... 20  
  Timing Jacket ................................................................................ 21  
  Typical Workflow ............................................................................ 22  
  Example 1: Vectorized Code .......................................................... 23  
  Example 2: GFOR Code ................................................................. 27  
Chapter 5: Optimizing Performance ............................................ 29  
  Jacket Examples ............................................................................. 31  
  Jacket Limitations .......................................................................... 32  
  Trace Saving ................................................................................... 33  
  Parallel computing using gfor/gend .................................................. 34
Contacting AccelerEyes

Online
Web:  http://www.accelereyes.com
Forums:  http://www.accelereyes.com/forums
Twitter:  http://twitter.com/accelereyes
LinkedIn:  http://www.linkedin.com/companies/314593

Email
Sales:  sales@accelereyes.com
Support:  support@accelereyes.com
Careers:  jobs@accelereyes.com

Phone:  +1 (800) 570-1941
(Ext: 1 for Sales)
(Ext: 2 for Technical support)

Physical & Mailing Address:
AccelerEyes
75 5th St NW STE 204
Atlanta, GA, 30308, USA

Other Documentation
Online Docs:  http://www.accelereyes.com/wiki
FAQ:  http://www.accelereyes.com/faq
Revision History
April 3, 2010  -  First revision (Jacket v1.3)
Chapter 1 – Introduction to GPU Computing

Graphics Processors for General Purpose Computing

Over the past few years, specialized coprocessors from floating point hardware to field programmable gate arrays have enjoyed a widening performance gap with traditional x86-based processors. Of these, graphics processing units (GPUs) have advanced at an astonishing rate, currently capable of delivering over 1 TFOPS of single precision performance and over 300 GFLOPS (see Figure 1 below) of double precision while executing up to 240 simultaneous threads in one low-cost package. As such, GPUs have gained significant popularity as powerful tools for high performance computing (HPC) achieving 20-100 times the speed of their x86 counterparts in applications such as physics simulation, computer vision, options pricing, sorting, and search.

A GPU is a highly parallel computing device designed for the task of graphics rendering. However, the GPU has evolved in recent years to become a more general processor, allowing users to flexibly program certain aspects of the GPU to facilitate sophisticated graphics effects and even scientific applications. In general, the GPU has become a powerful device for the execution of data-parallel, arithmetic (versus memory) intensive applications in which the same operations are carried out on many elements of data in parallel. Example applications include the iterative solution of PDEs, video processing, machine learning, and 3D medical imaging.

What is the reason for the large performance gap between many-core GPUs and general purpose multi-core CPUs? The answer lies in the fundamental architectural design of these two processors, as illustrated in Figure 2. The design of a CPU is optimized for sequential code performance. It makes use of sophisticated control logic to allow instructions from a single thread of execution to execute in parallel or even out of sequential order while maintaining the appearance of sequential execution. More importantly, large cache memories are provided to reduce the instruction and data access latencies of large complex applications. Neither control
logic nor cache memories contribute to the peak calculation speed. As of 2008, the new general purpose multi-core microprocessors typically have four large processor cores designed to deliver strong sequential code performance.

![Figure 2: Architectural Differences between GPUs & CPUs](image)

The design philosophy of the GPUs has historically been motivated by the fast growing video game industry that exerts tremendous economic pressure for the ability to perform massive numbers of floating point calculations in advanced games. Therefore, the design goal for GPU vendors is to look for ways to maximize the chip area and power budget dedicated to floating point calculations. The general philosophy for GPU design is to optimize for the execution of massive number of threads. The hardware takes advantage of a large number of execution threads to find work to do when some of them are waiting for long-latency memory accesses, minimizing the control logic required for each execution thread. Small cache memories are provided to help control the bandwidth requirements of these applications so that multiple threads that access the same memory data do not need to all go to DRAM. As a result, much more chip area is dedicated to the floating-point calculations.

With the release of NVIDIA’s Compute Unified Device Architecture (CUDA), the hardware architecture of modern GPUs can be viewed as a series of Single Instruction Multiple Data (SIMD) multiprocessors, each capable of processing a set of instructions on different memory elements in a clock cycle. Typically, the GPU is accessed for computation or graphics rendering through its device driver using a graphics API, such as OpenGL or DirectX, or through specialized APIs such as CUDA (which shares the same name with NVIDIA’s hardware architecture). GPU computing with CUDA is a more modern and capable GPGPU computing approach than utilizing the former two APIs. CUDA forms the underlying workhorse for Jacket.

CUDA is a software architecture and API geared towards the utilization of the GPU as a computing device rather than a graphics rendering device. The CUDA software includes a GPU device driver, a runtime system that serves as an abstraction over the driver, and also runtime libraries that CUDA applications may link to in order to provide GPU-enabled FFT and BLAS support, among others. CUDA also includes a compiler toolchain which provides extensions onto the C/C++ languages for the construction of GPU applications. While programming the GPU with the CUDA toolchain, the GPU is viewed as a coprocessor to the CPU, or host, which orchestrates the executions carried out by the GPU as needed. In order to utilize the GPU to its fullest potential, the CPU must minimize data communication with the GPU, due to limited bus
bandwidth, and maximize data parallelism in the tasks given to the GPU to maximize usage of GPU processors. Though the GPU can be viewed as capable of executing a large number of general threads in parallel, GPU programming is still typically accomplished through the specification of kernels which operate across an array of data elements. These kernels are limited in their length and the amount of local memory they use. The potential bottlenecks involved in computing with the GPU include memory allocation, memory transfer, and kernel execution. In the ideal case, each of these tasks is done sparingly to ensure that minimal overhead is accrued over the lifetime of an application. Jacket, which we describe in the following sections minimizes these tasks transparently and yields high GPU/CPU performance for MATLAB applications with minimal effort from the user.

Important Concepts
A few basic concepts should be understood in getting started with GPU computing. These are outlined in the following few paragraphs:

Data-parallel Computations
In order to understand what algorithms work well on the GPU, it is important to understand the difference between data parallelism and task parallelism. There are many ways to define this, but simply put and in our context:

- **Task parallelism** is the simultaneous execution on multiple cores of many different functions across the same or different datasets.
- **Data parallelism** (aka SIMD) is the simultaneous execution on multiple cores of the same function across the elements of a dataset.

In particular GPUs are especially well-suited to address problems that can be expressed as data-parallel computations with high arithmetic intensity – a high ratio of arithmetic operations to memory operations. Data-parallel processing maps data elements to parallel processing threads. Many applications that process large data sets can use a data-parallel programming model to speed up simulations. For example, image and media processing applications such as post-processing of images, video encoding and decoding, image scaling, stereo vision, and pattern recognition very naturally map image blocks and pixels to parallel processing threads. Moreover, many algorithms outside the field of image rendering and processing are also accelerated by data-parallel processing ranging from general signal processing or physics simulation to computational finance or computational biology.

However, it should be understood that the GPU is designed as a numeric computing engine and it will not perform well on some tasks. Therefore, one should expect that most applications will use both CPUs and GPUs, executing the sequential parts on the CPU and numeric intensive parts on the GPUs. This **heterogeneous computing model** is *fully supported* by Jacket for joint CPU-GPU execution of an application.
Chapter 1 – Introduction to GPU Computing

Data Sizes and Transfers between Host and GPU
Another important concept in understanding the limitation with the heterogeneous computation model defined above is the significant overhead of memory transfers between the host CPU and the GPU. In general, the overhead of time spent in sending data to the GPU and bringing it back neutralizes any performance benefit obtained by computing on the GPU for smaller sized datasets. Moreover, GPUs offer best performance gains when all the computing resources, processing cores and memory, are maximally utilized. Therefore, from the user’s perspective, an analysis of data sizes is the best way to determine which jobs to offload to the GPU.

Why Jacket?
Jacket exists to enable domain professionals, including scientists, engineering, and analysts, to get the benefits of GPU computing without the hassle of GPU-specific programming constructs. Jacket overcomes this problem by providing a middleware approach to GPU programming, with MATLAB1 as the frontend point of interaction for the user. MATLAB with millions of users worldwide is the platform of choice for engineers and scientists alike, for rapid algorithm prototyping. MATLAB is an extensible interactive programming environment for numerical analysis built on a vector language called M. The M-language, like other vector languages, provides users with a high-level interface at which operations may be specified over large sets of data at once making the expression of data-parallel algorithms natural. M is also dynamically typed, adheres to pass-by-value semantics, and is integrated into a well developed interpreted environment. With these characteristics, M has proven to be a powerful, user-friendly language. Using Jacket, the M-language and MATLAB transparently adapt to GPGPU computing. Unlike other GPU solutions, Jacket provides GPU computation and graphics ability from a language which is inherently parallel and interpreted, thereby providing a standard, extensible, and simple method of programming for the GPU in an already proven rapid prototyping environment. Jacket adds few GPU-specific datatypes to MATLAB with overloaded operators and entire CPU-bound MATLAB programs can be converted into GPU-enabled programs through as little as adding a ‘g’ prefix onto memory allocation commands. Otherwise, the user interacts with MATLAB as they normally would either from the command line or when running scripts.

1 MATLAB® and its logo are registered trademarks of The MathWorks.
Chapter 2 – Product Overview

Jacket Overview

Jacket connects MATLAB to the GPU. MATLAB is a technical computing language that integrates computation, visualization, and programming in an easy-to-use environment that has found wide popularity both in the industry and academia. It is used across the breadth of technical computing applications including mathematical computations, algorithm development, data analysis, data visualization, and application development. With the GPU as a backend computation engine, Jacket brings together the best of three important computational worlds: computational speed, visualization, and the user-friendliness of M programming.

Jacket enables developers to write and run code on the GPU in the native M language used in MATLAB. Jacket accomplishes this by automatically wrapping the M language into a GPU compatible form. By simply casting input data to Jacket’s GPU data structure, MATLAB functions are transformed into GPU functions. Jacket also preserves the interpretive nature of the M language by providing realtime, transparent access to the GPU compiler.

Optimizing M-code for performance has traditionally been done by rewriting functions in C/C++ using MATLAB’s MEX interface. As multi-core and multi-CPU systems have been adopted, MATLAB users have turned to PCT and MDCS for parallel execution while C/C++ users have had to adopt MPI or OpenMP. Even with enhanced parallel tools, parallelization is difficult and time consuming and is completely focused on the CPU, ignoring the benefits of data-parallelism on GPUs. Jacket offers performance and productivity enhancements to augment parallelization efforts or to simply improve performance with existing computational resources.

Figure 3: Jacket Software for MATLAB
Chapter 2 – Product Overview

The Jacket System
The Jacket system consists of the following main parts:

Integration with MATLAB
Once Jacket is installed, it is transparently integrated with the MATLAB’s user interface and the user can start working interactively through the MATLAB desktop and command window as well as write M-functions using the MATLAB editor and debugger. All Jacket data is visible in the MATLAB workspace, along with any other MATLAB matrices.

GPU Data Types
Jacket provides GPU counterparts to MATLAB’s CPU data types, such as real and complex double, single, uint32, int32, logical, etc. Any variable residing in the host (CPU) memory can be cast to Jacket’s GPU data types. Jacket’s memory management system allocates and manages memory for these variables on the GPU automatically, behind-the-scenes. Any functions called on GPU data will execute on the GPU automatically without any extra programming.

GPU Functions
Jacket provides the largest available set of GPU functions in the world, ranging from functions like sum, sine, cosine, and complex arithmetic to more sophisticated functions like matrix inverse, singular value decomposition, Bessel functions, and Fast Fourier Transforms. The supported set of functions continues to grow with every release of Jacket (see the Function Reference Guide).

Runtime
The Jacket runtime is the most advanced GPU runtime in the world, providing automated memory management, compile-on-the-fly, and execution optimizations for Jacket-enable code.

Graphics
Jacket’s Graphics Toolbox is the only tool in the world that enables a merger of GPU visualizations with computation. With Jacket a simple graphics command can be added at the end of a simulation loop to visualize data as it is being computed while maintaining performance. Try out gsurf, gimage, gscatter3, gvolume, and more! (See the Graphics Toolbox Wiki page)

SDK and Compiler
The Developer SDK makes integration of custom CUDA code into Jacket’s runtime very easy. With a few simple SDK functions, your CUDA code can benefit from the optimized Jacket platform. When Jacket applications have completed the development, test, and optimization stages and are ready for deployment, the Jacket MATLAB Compiler allows users to generate license-free executables for distribution to larger user bases. (See the SDK and JMC Wiki pages)

Help
Interactive help for any Jacket function is available using Jacket’s ghelp function.
Chapter 2 – Product Overview

Jacket Applications and Benefits

Jacket opens the door for MATLAB and GPU programming to many new problems, including scientific computation, financial modeling, biological modeling, video game development, and many others. Jacket is ideal for:

- **Rapid Prototyping** – Now you can leverage the rapid prototyping benefits of MATLAB for GPU programming.

- **GPU Taste Testing** – Are you seriously considering GPU computing for your product development? Jacket enables you to taste test the benefits of the GPU before investing a lot of time and money in expensive developmental programs.

- **Eliminating the Port** – Are you a company which prototypes in MATLAB and then ports code to C/C++ or some other language? Are you unhappy with the speeds that your prototypes achieve in MATLAB? Now you will find that your MATLAB code runs in real time by leveraging the power of GPUs within the MATLAB environment.

- **3D Visualizations** – Do you need to render your results using the latest visualization technologies? Do you spend a lot of time trying to figure out how to simply display your results? With Jacket’s Graphics Toolbox, you can now do state-of-the-art visualizations in MATLAB, without resorting to time-intensive non-MATLAB approaches to visualization.

GPU computing with Jacket delivers several benefits to end-users:

- **Tremendous Speed** – Jacket-enabled MATLAB scripts achieve speed improvements of 10X – 50X, and in some cases much greater, over equivalent CPU versions.

- **Incredible Visualizations** – Jacket’s Graphics Toolbox dramatically improves MATLAB visualizations. The idea is simple – let the GPU visualize and compute without moving data back to the CPU. The Graphics Toolbox uses an OpenGL rendering system in real-time. These OpenGL programs run and may be modified entirely within the interpretive MATLAB environment.

- **User-Friendliness** – Jacket is NOT another GPU language. Rather, Jacket eliminates the need to learn GPU-specific programming languages in order to access the benefits of the GPU. Jacket-enabled MATLAB also inherits all of the user-friendly features commonly associated with MATLAB programming.

- **Low Cost HPC Solutions** – GPU video cards have become standard in most desktop and laptop computers, eliminating the cost of expensive cluster computing alternatives for high-performance computing (HPC) tasks.
Chapter 2 – Product Overview

Jacket Product Family

Jacket provides a level of productivity and return on investment far beyond any other development platform on the market. While there are many tools that enable performance improvement, Jacket delivers performance at a whole new level by unleashing the power of GPUs to deliver maximum FLOPS per dollar. Jacket Base runs on a single GPU in a system.

Workstations, SMP servers, and Personal Supercomputers (PSCs) with multiple GPUs can benefit from Jacket MGL that extends the single GPU support of the base Jacket product to as many as eight (8) GPUs in a single machine. Where HPC resources, such as GPU clusters, are already in place, Jacket HPC can harness the power of 8 GPUs or more in a cluster or cloud environment.

![Jacket Product Family Diagram](image)

Figure 4: Jacket Product Family

Jacket DLA extends the real, single precision linear algebra functionality available in the base Jacket product to include complex and double precision functions. Where applications require double precision or complex matrices, Jacket DLA is an available add-on product.

Jacket GFX provides powerful 3D visualization capabilities for Jacket applications. Jacket GFX is included with the base Jacket product at no additional cost.

Jacket SDK makes integration of custom CUDA code into Jacket’s runtime very easy. With a few simple SDK functions, your CUDA code can benefit from the optimized Jacket platform.

Jacket JMC enables the license-free distribution of Jacket-based compiled executables. JMC works in conjunction with the MATLAB Compiler to produce distributable GPU-accelerated executable programs.
This chapter provides details for installing and activating Jacket licenses.

**Terms of Use**
See <jacket>/doc/EULA.pdf for a copy of the Jacket End User License Agreement (EULA). By downloading and using this software, you agree to the Jacket EULA.

**Pre-Requisites**
In order to use Jacket, you must have the following pre-requisites:

1. Jacket supports the following Operating Systems:
   a. Windows XP/Vista/7 (32 & 64 bit)
   b. Linux (32 & 64 bit) – Fedora 10+, OpenSUSE 11+, RHEL 5.x, CentOS 5.x, SLES 10.2
   c. Mac OS X (32-bit only)

2. CPU memory should be greater than or equal to GPU memory.

3. A CUDA-capable NVIDIA GPU

4. NVIDIA CUDA Drivers
   b. Jacket v1.3 requires CUDA drivers 190.38 or higher for Windows systems and 190.53 or higher for Linux systems

5. MATLAB (R2006B or later)
   a. [http://www.mathworks.com](http://www.mathworks.com)

6. For some Windows systems, the .NET framework must be installed

7. NVIDIA GPUs of all CUDA capability levels (e.g. 1.0, 1.1, 1.2, & 1.3) are fully supported.

**Linux Specific Instructions**
As of Jacket 1.3, the CUDA Toolkit is packaged with Jacket. However, for Linux systems, if the machine has a separate CUDA Toolkit in addition to the one provided by Jacket, then you must edit LD_LIBRARY_PATH to give Jacket’s shared libraries a higher precedence, as follows:

   a. For 32-bit machines: `export LD_LIBRARY_PATH=<jacket>/engine/cuda/lib:$LD_LIBRARY_PATH`
   b. For 32-bit machines: `export LD_LIBRARY_PATH=<jacket>/engine/cuda/lib64:$LD_LIBRARY_PATH`
MAC OS Specific Instructions
For Mac OS only, there is the added prerequisite to have the CUDA Toolkit installed on the machine. Jacket 1.3 requires CUDA Toolkit 2.3, available for download here:


If running from Terminal, you need to set LD_LIBRARY_PATH in your shell configuration script (.bashrc, .cshrc, etc.). For example, bash would require:

```
LD_LIBRARY_PATH=<jacket>/engine:/usr/local/cuda/lib:$LD_LIBRARY_PATH
```

If running from the MATLAB icon, you should also edit “$HOME/.MacOSX/environment.plist" to include this same path as shown in the sample listing in Figure 5. Logout and login again for the changes to take effect.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
  <dict>
    <key>PATH</key>
    <string>/bin:/sbin:/usr/bin:/usr/X11R6/bin</string>

    <key>LD_LIBRARY_PATH</key>
    <string>/usr/local/jacket/engine:/usr/local/cuda/lib</string>
  </dict>
</plist>
```

Figure 5: Sample ‘environment.plist’ file. The text in red shows the required updates.
Chapter 3 – Installation and Activation

Jacket Installation

For Windows:

*Note: You will need to manually uninstall previous versions of Jacket which used the Nullsoft-based installer. You can do this from the “Add/Remove Programs” Windows utility.*

Run the automated installer included in the download.

- The default directory installation directory is **C:/Program Files/AccelerEyes/Jacket**
- Previous versions of Jacket will need to be manually removed “Add/Remove Programs”
- An uninstaller is provided
- An automated activation wizard is provided. Alternatively, you can activate manually by using `gactivate` and the Manage License page on the website.

To use Jacket, simply add paths in MATLAB, as follows:

```
>> addpath('<path to AccelerEyesJacket>/engine');  % for Jacket
>> addpath('<path to AccelerEyesJacket>/gfx');     % for Graphics
>> addpath('<path to AccelerEyesJacket>/gfx/mgl'); % ... Toolbox
```

Alternatively, you may run the “Start Jacket” shortcut in the Start Menu. This script conveniently starts MATLAB and automatically adds paths to Jacket.

For Linux:

*Note: To uninstall previous versions of Jacket, simply remove the installation directory.*

Install using the downloaded “.run” file, as follows:

(Shell prompt) `sh Jacket-linux<version>.run`

- The default installation directory is **/usr/local/jacket**
- To uninstall, simply delete the installation directory
- Previous versions of Jacket will be removed in the installation process
- You must edit `LD_LIBRARY_PATH` to give Jacket’s shared libraries a higher precedence:
  - For 32-bit machines: `export LD_LIBRARY_PATH=<jacket>/engine/cuda/lib:<cuda>/lib`
  - For 32-bit machines: `export LD_LIBRARY_PATH=<jacket>/engine/cuda/lib64:<cuda>/lib64`

To use Jacket, simply add paths in MATLAB, as follows:

```
>> addpath('<path to jacket>/engine');  % for Jacket
>> addpath('<path to jacket>/gfx');     % for Graphics
>> addpath('<path to jacket>/gfx/mgl'); % ... Toolbox
```
Chapter 3 – Installation and Activation

For Mac:
Install the downloaded disk image file.

To use Jacket, as usual simply add paths in MATLAB, as follows:

```
>> addpath('<path to jacket>/engine');  % for Jacket
>> addpath('<path to jacket>/gfx');     % for Graphics
>> addpath('<path to jacket>/gfx/mgl'); % ... Toolbox
```

Manual Activation

Manual activation of a Jacket license is done in a two-step process: 1) run `gactivate` and 2) download the jlicense.dat file. (Note: Windows users can skip this step by using the installer’s automated activation feature).

Run `gactivate`

The Jacket `gactivate` command is used to determine the Host ID of your system which is required for designated computer licenses. The output of `gactivate` is shown below:

```
>> gactivate
Welcome to Jacket: The GPU Engine for MATLAB!
Checking CUDA driver... PASSED
Checking Jacket runtime... WAITING

AccelerEyes Jacket v1.3.0 (build 3804)
CUDA driver: 191.87, CUDA toolkit 2.3
Memory: 0 CPU-used, 0 GPU-used, 3943 GPU-free (in MB)
License Type: Designated Computer
License Features: jacket sdk mgl4 dla
Multi-GPU: Licensed for 4 GPUs
Detected CUDA-capable GPUs:
GPU0 Quadro FX 5800, 1265 MHz, 4095 MB VRAM, Compute 1.3 (single,double)

Checking Jacket runtime (cont)... PASSED

CONGRATULATIONS, JACKET HAS BEEN SUCCESSFULLY ACTIVATED!

If this is a trial version, you can upgrade to a purchased version by contacting sales@accelereyes.com. To upgrade from a trial license to a purchased license, please paste one of the Host IDs listed below into the Manage Licenses page (http://www.accelereyes.com/licenses) to get a Jacket license file, "jlicense.dat". Choose the Host ID that corresponds to the most stable network interface.

Host ID # 1 = 001966A33CE2
Chapter 3 – Installation and Activation

Generate & Download the License File
Once you have your Host ID, you need to download a jlicense.dat file from the AccelerEyes website at this link:  http://www.accelereyes.com/licenses

In the Manage License page, you will find two sections, one for trials and one for purchased licenses. In the purchased license section, you will find all of your current licenses, as shown in Figure 6. The following license fields are displayed:
- License Number
- Purchase Date
- Maintenance Expiration – licenses are only available for Jacket versions released before this date
- Support Expiration – the expiration date of phone support for this license
- Custom Label – choose a custom label for your license (optional)
- Features – all addon features associated with this license
- Host ID – the Host ID to which this license is bound
- License Download – The link to download jlicense.dat for valid licenses

Note, in Figure 6, we show three states in which licenses may exist:
- License #110000413 shows “Too Many Activations” indicating that this license must be activated by phone to AccelerEyes’ Support.
- License #110000415 shows “Not Available for 1.2.2” since the Maintenance expired before 1.2.2 was released.
- License #110000417 is active and available. It already has a generated license file that is downloadable from the provided link.
- License #110989648 is active and available. It has not yet been activated and needs a Host ID to be submitted to generate the downloadable license file.

<table>
<thead>
<tr>
<th>Purchased Designated Computer Licenses</th>
</tr>
</thead>
<tbody>
<tr>
<td>License Number</td>
</tr>
<tr>
<td>C110090413</td>
</tr>
<tr>
<td>C110090415</td>
</tr>
<tr>
<td>C110090417</td>
</tr>
<tr>
<td>C110989548</td>
</tr>
</tbody>
</table>

Submit Host Information

Figure 6: Manage Licenses for Manual Activation
Once you have generated your license file, place is in the <jacket>/engine directory and activation is complete. You can verify it is activated by re-running gactivate and looking for the phrase:  “CONGRATULATIONS, JACKET HAS BEEN SUCCESSFULLY ACTIVATED!”
Test Installation & Activation:
After Jacket has been activated, you can use the `ginfo` function to verify the installation and configuration of the GPU hardware in your machine. Figure 7 shows the information that is returned by the `ginfo` command.

![Figure 7: Output of ginfo command](image)

The output of the `ginfo` function provides information about the version of Jacket currently installed and a list of all the GPUs available for computation with Jacket. A single session of MATLAB can use only one of these GPUs at time. But multiple sessions of MATLAB can each be pointed at a separate GPU using the Jacket `gselect` function. The GPU that is currently selected for computations has a message (in use) printed next to it. However, you can use `gselect` to change this selection to another GPU you wish to use. This selection must be made before any other GPU computations are performed.

```plaintext
>> gselect( <GPU #> );  % see ginfo to get the GPU number
```
In this chapter, you will find information to help you in getting started using Jacket.

**Casting to the GPU**

Once you have correctly installed and activated Jacket, you will be able to create GPU matrices in the same way that CPU data structures are created. The functions in Table 1 are used to create GPU data structures:

<table>
<thead>
<tr>
<th>Jacket Function</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>gsingle</td>
<td>Casts a MATLAB matrix to a single precision floating point GPU matrix.</td>
<td>&gt;&gt; A = gsingle(B);</td>
</tr>
<tr>
<td>gdouble</td>
<td>Casts a MATLAB matrix to a double precision floating point GPU matrix.</td>
<td>&gt;&gt; A = gdouble(B);</td>
</tr>
<tr>
<td>glogical</td>
<td>Casts a MATLAB matrix to a binary GPU matrix. All non-zero values are set to '1'. The input matrix can be a GPU or CPU datatype.</td>
<td>&gt;&gt; A = glogical(B); &gt;&gt; A = glogical(0:4);</td>
</tr>
<tr>
<td>guint32, gint32</td>
<td>Cast a MATLAB matrix to a signed and unsigned 32 bit integer GPU matrix respectively.</td>
<td>&gt;&gt; A = guint32(B); &gt;&gt; A = gint32(B);</td>
</tr>
<tr>
<td>guint8, guint8</td>
<td>Cast a MATLAB matrix to a signed and unsigned 8 bit integer GPU matrix respectively.</td>
<td>&gt;&gt; A = guint8(B); &gt;&gt; A = gint8(B);</td>
</tr>
<tr>
<td>gzeros</td>
<td>Creates a matrix of zeros analogous to the MATLAB zeros function.</td>
<td>&gt;&gt; A = gzeros(5);</td>
</tr>
<tr>
<td>gones</td>
<td>Creates a matrix of ones analogous to the MATLAB ones function.</td>
<td>&gt;&gt; A = gones(5);</td>
</tr>
<tr>
<td>geye</td>
<td>Creates an identity matrix analogous to the MATLAB eye.</td>
<td>&gt;&gt; A = geye(5);</td>
</tr>
</tbody>
</table>

Once a GPU data structure has been created, any operations on that GPU matrix are performed on the GPU rather than the CPU. To turn off GPU computation, simply cast the data back to the CPU using one of the MATLAB data types, e.g. double. These functions are used as follows:

```matlab
>> A = gdouble(B); % to push B to the GPU from the CPU
>> B = double(A);  % to pull A from the GPU back to the CPU
```

The functions listed in Table 1 are all you need to know for basic Jacket GPU computing.
Chapter 4 – Getting Started

Something Familiar – Multiplying 2 Matrices
In this section, we provide a quick example using Jacket for GPU matrix multiplication.

```matlab
>> X = gdouble( magic( 3 ) );
>> Y = gones( 3, 'double' );
>> A = X * Y

A =
    15    15    15
    15    15    15
    15    15    15
```

Note that the only difference between this code and CPU-based MATLAB code is the introduction of the `gdouble` casting functions.

Something Familiar – Approximating Pi
In this section, we provide a quick example using Jacket to approximate Pi on the GPU. This example is included in the Jacket examples directory:

```matlab
>> NSET = 1000000;
>> X = gdouble( rand( 1, NSET ) );
>> Y = gdouble( rand( 1, NSET ) );
>> distance_from_zero = sqrt( X.*X + Y.*Y );
>> inside_circle = (distance_from_zero <= 1);
>> float_pi = 4 * sum(inside_circle) / NSET

float_pi =
    3.1421
```

Note that the only difference between this code and CPU-based MATLAB code is the introduction of the `gdouble` casting functions.
Timing Jacket

Jacket contains two functions that are important to use when timing GPU code using MATLAB’s tic/toc functions: gsync and geval. Due to Jacket’s lazy execution, these functions must be used to ensure that Jacket operations complete prior to the final toc. The following is a description of those functions.

```plaintext
>> %GSYNC Block until all queued GPU computation is complete.
>> % GSYNC Waits until the GPU has completed all enqueued instructions
>> % Examples:
>> %     gsync  % ensure GPU is ready to begin
>> %     tic
>> %     for i = 1:n
>> %       a = a .* a;  % some computation
>> %     end
>> %     geval(a) % ensure computation is evaluated
>> %     gsync  % wait until GPU finishes executing
>> %     toc
>> % See also GEVAL, GONES, GZEROS.
```

```plaintext
>> %GEVAL Evaluate computation and leave results on GPU.
>> % GEVAL(A) performs computation A and leaves result out on the GPU.
>> % GEVAL(A,B,C,...) performs multiple computations.
>> % Examples:
>> %     A = gones(3) + 8;
>> %     geval(A);
>> %     B = gones(3) + 8;
>> %     C = B * gones(3) + i;
>> %     geval(B, C);
>> % See also GSYNC, GONES, GZEROS.
```

Refer to the Jacket Function Reference Guide for details on these and any other Jacket function.
Typical Workflow
The recommended workflow to Jacketize MATLAB code is shown in Figure 8 below.

![Figure 8: Typical Jacket Workflow](image)

The first step is to profile the code using the MATLAB Profiler\(^2\). The results of the profiler can help determine where the program is spending most of its time. The next step is to cast the input variables of slow code segments to GPU data types provided by Jacket (e.g. `gdouble`) as shown in the above example. The subsequent MATLAB operations that are performed on the GPU matrices will automatically be executed on the GPU. For most data parallel computations this is all that is necessary to get performance boost from the GPU. However, there are several factors that may affect Jacket’s performance and if desired speedup is not achieved, additional analysis and some modification of the original M-code may help optimize performance. The next section outlines some of the strategies that maybe helpful in this regard.

Example 1: Vectorized Code

In order to demonstrate a standard workflow for Jacketizing MATLAB code, we provide an example that demonstrates the process. In this example, we show a script that performs a 3D Finite Difference Time Domain (FDTD) simulation of electromagnetic wave propagation. The three steps that are required to Jacketize this script are as follows: Profile, Jacketize, Optimize.

Profile. The first step is to profile the script using the MATLAB Profiler. Figure 9 shows how to start the profiler in MATLAB.

Once the profiler window appears enter your function name in the “Run this code” field. Once the simulation has completed, the profiler window will show details of the computational time for each instruction in the MATLAB script. For example, in this code an initial run of the profiler indicates that the program spends most of its time in computation of finite differences. Nearly 76% of the time is spent in this part of the code. All these computations are inside the simulation for-loop and are candidate computations to be dispatched to the GPU using Jacket.
Chapter 4 – Getting Started

Jacketize. The next step is to cast input data to this segment to the GPU. Generator functions such as zeros, ones, eye, and rand should be replaced with the Jacket equivalent versions, gzeros, gones, geye, and grand. The modifications made for this example are:

```matlab
ex=zeros(ie,jb,kb,'single');
ey=zeros(ib,je,kb,'single');
ez=zeros(ib,jb,ke,'single');
hx=zeros(ib,je,ke,'single');
ex=gzeros(ie,jb,kb,'single');
ey=gzeros(ib,je,kb,'single');
ez=gzeros(ib,jb,ke,'single');
hx=gzeros(ib,je,ke,'single');
```
And at the end of the GPU script, we cast back to CPU as shown below:

\[
\begin{align*}
\text{ex} &= \text{single} (\text{ex}) ; \\
\text{ey} &= \text{single} (\text{ey}) ; \\
\text{ez} &= \text{single} (\text{ez}) ; \\
\text{hx} &= \text{single} (\text{hx}) ; \\
\text{hy} &= \text{single} (\text{hy}) ; \\
\text{hz} &= \text{single} (\text{hz}) ;
\end{align*}
\]

However, even though this script has been Jacketized, it fails to achieve performance boost from the GPU. In fact, the Jacketized version runs slower. What is wrong? The problem is the way finite differences are being computed using offset sub-referencing, which is very memory intensive. The better approach is use convolution to do the computation without rearranging the data. With this simple optimization, a big boost in performance is realized.

**Optimize.** Now we look at the computation of finite differences and determine an alternative to memory intensive indexing. One natural way to perform this computation on the GPU is to formulate it as a convolution operation with 3x3x3 Sobel difference operator. The computations that need to be replaced are outlined in red below.

```matlab
for n=1:nmax

%***********************************************************************
%  Update electric fields
%***********************************************************************

ex(1:ie,2:je,2:ke)=ca*ex(1:ie,2:je,2:ke)+...
   cb*(hz(1:ie,2:je,2:ke)-hz(1:ie,1:je-1,2:ke)+...
        hy(1:ie,2:je,1:ke-1)-hy(1:ie,2:je,2:ke));

ey(2:ie,1:je,2:ke)=ca*ey(2:ie,1:je,2:ke)+...
   cb*(hx(2:ie,1:je,2:ke)-hx(2:ie,1:je,1:ke-1)+...
        hs(1:ie-1,1:je,2:ke)-hs(2:ie,1:je,2:ke));

ez(2:ie,2:je,1:ke)=ca*ez(2:ie,2:je,1:ke)+...
   cb*(hx(2:ie,1:je-1,1:ke)-hx(2:ie,2:je,1:ke)+...
        hy(2:ie,2:je,1:ke)-hy(1:ie-1,2:je,1:ke));

ez(is,js,1:ke)=ez(is,js,1:ke)+...
   srcconst*(n-n_delay)*exp(-((n-n_delay)^2/tau^2));

%***********************************************************************
% Formulate it as a convolution operation with 3x3x3 Sobel difference operator.
% The computations that need to be replaced are outlined in red below.
%***********************************************************************
```

The first step is to create the kernels for long difference operators as shown below:

```matlab
diff_kernel1 = zeros(3,3,3); diff_kernel1(2,2,2)=1; diff_kernel1(2,3,2)=-1;
diff_kernel2 = zeros(3,3,3); diff_kernel2(2,2,2)=-1; diff_kernel2(2,2,3)=1;
diff_kernel3 = zeros(3,3,3); diff_kernel3(2,2,2)=1; diff_kernel3(2,2,3)=-1;
diff_kernel4 = zeros(3,3,3); diff_kernel4(2,2,2)=-1; diff_kernel4(3,2,2)=1;
```
And then simulation updates for the electric and magnetic fields are updated as convolution operations using `convn`.

```matlab
for n=1:nmax

% Update electric fields

% Obtaining the convolution results

temp1 = convn(hz(1:ie,1:je,2:ke), diff_kernel1, 'same');
temp2 = convn(hz(1:ie,2:je,1:ke), diff_kernel2, 'same');
ex(1:ie,2:je,2:ke) = ca*ex(1:ie,2:je,2:ke) + cb* (temp1(:, 2:end, :) ... + temp2(:, :, 2:end));

% Updating the magnetic fields

temp1 = convn( hx(2:ie,1:je,1:ke), diff_kernel3, 'same');
temp2 = convn( hx(1:ie,1:je,2:ke), diff_kernel4, 'same');
ey(2:ie,1:je,2:ke) = ca*ey(2:ie,1:je,2:ke) + cb* (temp1(:, :, 2:end) ... + temp2(2:end, :, :));

temp1 = convn( hy(2:ie,1:je,1:ke), diff_kernel5, 'same');
temp2 = convn( hy(1:ie,2:je,1:ke), diff_kernel6, 'same');
ez(2:ie,2:je,1:ke) = ca*ez(2:ie,2:je,1:ke) + cb* (temp1(:, 2:end, :) ... + temp2(2:end, :, :));
ez(is,js,1:ke) = ez(is,js,1:ke) + ...
srcconst*(n-ndelay)*exp(-(n-ndelay)*(n-ndelay)/(tau^2));
```

Now if we run this Jacketized code we get a big speedup. For example, running the code on a dual-core Intel CPU with an NVIDIA Quadro FX 5800 GPU, a performance improvement of 16X is achieved on the GPU.
Example 2: GFOR Code

Another very powerful feature available with Jacket is the `gfor/gend` programming construct. This feature is discussed in detail in Chapter 5. Here we show a simple example to illustrate typical Jacket workflow. We take an example case where it is required to compute several small sized 2D FFTs, as follows:

```matlab
>> N = 128; % matrix size
>> M = 200; % number of tiled matrices
>> %Create Data
>> [Ac Bc] = deal(complex(ones(N,N,M, 'single'),0));
>> % Compute 200 (128x128) FFTs
>> for ii = 1:M
>>    Ac(:,:,ii) = fft2(Bc(:,:,ii));
>> end
```

**Profiler.** Although this is a very simple script we run it through the MATLAB Profiler to get insights into the computational burden of the script. The results from the profiler are shown in Figure 11. It shows that 200 calls are made to the `fft2` function.

![Profiler result](image)

**Figure 11: Profile of Example 2**
Chapter 4 – Getting Started

Jacketize. This script can be Jacketized by modifying a single generator function, as follows:

```matlab
>> N = 128; % matrix size
>> M = 200; % number of tiled matrices
>> % Create Data
>> [Ac Bc] = deal(complex( gones(N,N,M, 'single'),0));
>> % Compute 200 (128x128) FFTs
>> for ii = 1:M
>>     Ac(:,:,ii) = fft2(Bc(:,:,ii));
>> end
>> % Bring the results back to CPU
>> Ac = single(Ac);
```

This simple Jacketization results in a 60% speed improvement.

Optimize. But with `gfor/gend`, it is possible to achieve more performance. Each iteration of the `for`-loop is independent from the rest and can therefore be computed in parallel, as follows:

```matlab
>> N = 128; % matrix size
>> M = 200; % number of tiled matrices
>> % Create Data
>> [Ac Bc] = deal(complex( gones(N,N,M, 'single'),0));
>> % Compute 200 (128x128) FFTs
>> gfor ii = 1:M
>>     Ac(:,:,ii) = fft2(Bc(:,:,ii));
>> gend
>> % Bring the results back to CPU
>> Ac = single(Ac);
```

The new `gfor` modified script achieves a 600% speedup on a dual-core Intel machine with an NVIDIA Quadro FX 5800 GPU.
Chapter 5 – Optimizing Performance

To achieve the best performance with Jacket, it is helpful to keep in mind some tips:

1. **Vectorized Code** – Both MATLAB and Jacket perform best on vectorized code. They both take advantage of the inherent parallelism of the M-language which is extremely powerful when utilized wisely. For a good guide to writing vectorized MATLAB code, see: [http://www.mathworks.com/support/tech-notes/1100/1109.html](http://www.mathworks.com/support/tech-notes/1100/1109.html).

2. **Memory Transfers** – Avoid excessive memory transfers. Each casting operation to and from the GPU pushes or pulls data back and forth from CPU memory to GPU memory. Jacket minimizes these memory transfers automatically in normal operation, but excessive casts between CPU and GPU memory may reduce performance.

3. **Serial vs Parallel Operations** – Remember, CPUs are serial computing devices and GPUs are parallel computing devices. For small or serial operations, the best performance is likely achieved on the CPU. For large or parallel operations, the best performance is likely achieved on the GPU. You can control everything through the casting operations.

4. **Hybrid vs Purebred Computations** – Computations involving pure GPU data (e.g. a `gdouble` times a `gdouble`) tend to be faster than computations involving a hybrid of GPU and CPU data (e.g. a `gdouble` times a `double`).

   ```matlab
   >> geye( 3 ) * ones( 3 );   % Bad hybrid computation
   >> geye( 3 ) * gones( 3 );  % Good purebred computation
   ```

5. **Loops** – Some GPU-based computations must pass through Jacket’s compile on-the-fly system, which invokes NVIDIA’s compiler, nvcc. Compiling kernels is computationally expensive when done with nvcc. Jacket minimizes this expense by employing a lazy execution design and trace saving. However, in loops, it is important to watch out for iterating parameters which may be forcing an nvcc compilation with every loop iteration. Random numbers too, can force compilations in certain cases and slow down performance (see [http://blog.accelereyes.com/blog/2010/02/09/109/](http://blog.accelereyes.com/blog/2010/02/09/109/) for more).

   ```matlab
   >> A = geye( 3, 'double' );
   >> for n = 1:10,
   >>   A * n  % Since ‘n’ is a CPU-based parameter, each
   >>   % iteration will incur a costly nvcc
   >>   % compilation.
   >> end
   ```
Chapter 5 – Optimizing Performance

Advanced users may wish to avoid these costly nvcc compilations in loops by making sure that the loop iterating parameters used in computations are GPU-based.

```matlab
>> A = geye(3, 'double');
>> for n = gdouble(1:10),
    >> A * n  % Since ‘n’ is a GPU-based parameter, only the
    >>     % first iteration will incur an nvcc
    >>     % compilation.
>> end
```

6. **Lazy Execution** – Jacket employs a lazy execution design to provide optimal performance for your application. Lazy execution means that Jacket does not launch GPU kernels until the results are requested, either in a display or a subsequent CPU-based computation. There are exceptions to this rule to allow for optimal kernel configurations (e.g. Jacket does not allow kernels to get overly bulky).

```matlab
>> X = gdouble( magic(3) );
>> Y = gones(3, 'double');
>> A = X * Y;  % NO computation is performed, A is not needed
>> A;
>> A          % NOW, the computation is performed, since the
             % results are needed for the output display.
A =
 15    15    15
 15    15    15
 15    15    15
```

If you wish to force a GPU computation, the `geval` and `gsync` functions are available. **Note**, `geval` and `gsync` are especially important when benchmarking code.

```matlab
>> X = gdouble( magic(3) );
>> Y = gones(3, 'double');
>> A = X * Y;  % NO computation is performed, A is not needed.
>> A;
>> gsync;      % NOW, the computation is performed, even
>>            % though the output is not required anywhere.
```

7. **Warming Up Computations** – The Jacket runtime includes a compile-on-the-fly subsystem so that GPU kernels are optimally adapted for your data and application. The downside of such systems is that there is a slight overhead for the compilation. You can minimize the impact of this by “warming up” computations.

```matlab
function [ out_ ] = myfunction( in_ )
    % Jacket function where out_ = const*in_^2
    const_ = gsingle(0.667);
    out_   = gzeros(size(in_));   % Preallocate
    out_   = const_ * in_.*^2;    % Compute output
end
```
where \( \text{in}_\) is the \text{gsingle} input to be processed, and \( \text{out}_\) is the output from \text{myfunction}. An underscore is used to identify Jacket variables. In the M-file which controls our simulation, we can then do the following:

```matlab
>> % Size of quadratic input matrix
>> Size = 1000;
>>
>> % Preallocate variables
>> in_ = grand(Size,Size);
>> out_ = gzeros(Size,Size);
>>
>> % Warm up Jacket and CUDA for 'myfunction'
>> dummy_ = gzeros(5,5); % Predefine dummy variable
>> dummy_ = myfunction( grand(5,5) ); % warm-up the GPU for
>> % myfunction
>> clear dummy_; % Clear workspace for dummy_
>>
>> % Do the computation and get the timing
>> t1 = tic;
>> out_ = myfunction( in_ );
>> Telapsed = toc(t1)
```

As seen above the warm-up is done by calling precisely the relevant Jacket function - but with a very small input matrix. Small input arrays are sufficient to warm-up the computation.

**Jacket Examples**

Many examples are also available with Jacket in the \(<\text{Jacket Installation Folder}>\)/examples directory. All you need to do in order to run these examples is to either add the path to an example folder using \text{addpath} function or navigate into the example folder. You can also download these examples individually from the AccelerEyes website documentation page found here:

- [http://www.accelereyes.com/services/functionality](http://www.accelereyes.com/services/functionality)
- [http://www.accelereyes.com/services/applications](http://www.accelereyes.com/services/applications)

You can also visit the Jacket FAQ page at [www.accelereyes.com/faq](http://www.accelereyes.com/faq). This page lists category-wise, all frequently faced issues with download, installation and use of Jacket.
Chapter 5 – Optimizing Performance

Jacket Limitations
As Jacket continues to become increasingly feature-rich, more and more MATLAB functions will be entering the category of Fully Supported (FS). In the meantime, it is worthwhile to get acquainted with some of the current limitations and ways to write code that works around these.

Subscripting
Subscripting is a very powerful feature of the M-language. There are three basic types of indexing available in MATLAB: subscripted, linear, and logical. Examples include:

```
>> A = [ 11 14 17; ... 
    12 15 18; ... 
    13 16 19 ];
>> A( 2:3 , 2 )    % Subscripted Indexing (Output is [15 16]') - SUPPORTED
>> A( 6 )          % Linear Indexing (Output is 16) - SUPPORTED
>> A( A > 17 )     % Logical Indexing (Output is [18 19]') – NOT SUPPORTED
>> B = find( A > 17 ); A( B )     % SUPPORTED
```

Currently, Jacket only supports logical indexing with the find function. At times you may wish to index with GPU data type. This feature is supported in Jacket with the limitation that there can only be up to 3 GPU index variables at a time, as follows:

```
>> A( gdouble(1:2), gdouble(1:2) , gdouble(1:2) );
```

Kernels Size/Type Limits ( filter2/conv2, convn )
The filter2/conv2 and convn functions are popular in MATLAB. At present, these functions are in the Partially Supported (PS) category. There are a number of limitations in using these functions with Jacket. First, the data type of the kernel input parameter cannot be of a GPU type. Second, the size of the kernel for the asymmetric case is limited to 10x10 or smaller for filter2/conv2 but for symmetric kernels can be much larger. The kernel size is limited to 3x3x3 for convn. Third, filter2/conv2 only supports same and valid as inputs for the shape parameter while convn only supports same. The example usages of these functions are shown below:

```
>> X = gones( 10, 10 ); % Data to be filtered
>> B = ones( 3, 3 ); % Kernel (3x3 CPU data)
>>
>> A1 = filter2( B, X );          % Default 'shape' is 'same'
>> A2 = filter2( B, X, 'same' ) % Supported
>> A3 = conv2( B, X, 'same' ) % Supported
>> A4 = filter2( gdouble(B), X ); % Not supported due to GPU kernel
>>
>> B = gones( 10, 10, 10 ); % 3D Data Matrix
>> A = ones( 3, 3, 3 ); % Kernel (3x3x3 CPU data)
>> C = convn( B, A, 'same' ); % Supported
>> C = convn( B, A ); % Not supported – default 'shape' is 'full'
```
Chapter 5 – Optimizing Performance

Trace Saving

Jacket now includes a feature that enables the compiled traces of Jacket code to be saved to disk. The Jacket engine directory includes a cache.jkt file that contains a pre-compiled Jacket code to avoid common compilations, thus eliminating the compile time that occurs on first runs of code.

You can also save the compiled traces of your Jacket code to cache.jkt files that you can share with others, which will include compiled traces of your Jacket code. In order to do this, run Jacket’s gcache function, which has the following functionality:

```bash
>> help gcache
GCACHE   Save GPU compiled code for given MATLAB script
          During the course of a MATLAB session, GPU computations are compiled and
          stored in an on-disk cache to improve performance between MATLAB sessions.
          GCACHE allows granular control over this functionality. During typical
          usage of MATLAB, a default cache (locations for each OS given below) is
          updated to contain all GPU compiled code utilized thus far. However, for
          specific scripts, the user may save compiled GPU code to a user-defined
          cache which can then be loaded in another MATLAB session to avoid time-
          consuming compilation. The GCACHE command allows the user to manipulate the
          cache by flushing it, saving it, or loading it.

Syntaxes:
  GCACHE FLUSH Flushes any entries in the cache, leaving an empty cache in
               memory.
  GCACHE LOAD Loads default cache, merging it with any GPU code already in
               memory.
  GCACHE LOAD <FILENAME> Loads the specified cache, merging it with any GPU
               code already in memory.
  GCACHE SAVE Will save all cache entries in memory to the default cache.
  GCACHE SAVE <FILENAME> Will save all cache entries in memory to the
               specified file.

Example:
gcache my_cache.jkt
This will create my_cache.jkt on disk containing all compiled GPU code for
the current MATLAB session. Run this after a large script to avoid
recompilation in the next MATLAB session or share this with other users.

OS Specific default cache locations:
  - Windows; $TEMP/cache.jkt
  - Linux/OSX; $HOME/.cache.jkt
```

Copyright 2010 AccelerEyes
Parallel computing using gfor/gend

(Note: This is a preliminary feature and only supports a subset of the functionality that regular Jacket code supports. In general, for both MATLAB and Jacket code, it is better to vectorize as much as possible to avoid for-loops and gfor-loops.)

The gfor/gend loop construct may be used to simultaneously launch all of the iterations of a for-loop on the GPU. While the standard MATLAB for-loop performs each iteration sequentially, Jacket's gfor-loop performs each iteration at the same time. Jacket does this by tiling out the values of all loop iterations and then performing computation on those tiles in one pass.

Many features and functions of Jacket are not supported within gfor-loops. The currently supported functions supported in the gfor/gend construct are given below:

- Basic element-wise arithmetic (addition, subtraction, multiplication, division, pow, exp)
- fft, fft2, fftn, and their inverses
- transpose, ctranspose, and diag
- matrix-matrix, matrix-vector, vector-vector multiply (mtimes)
- subscripted assignment/referencing
- reductions (sum, min, max, any, all)

This is not a comprehensive list, so please use Jacket’s ghelpe to inquire about any specific function’s gfor support. We appreciate your feedback on the Jacket Forums as we continue to expand this functionality and improve its performance.

Usage

Jacket supports bulk multiplications of matrix-matrix, matrix-vector, and vector-vector types using the gfor/gend construct. This is especially useful with many small matrices.

```matlab
>> A = gones(n);
>> B = gones(1,n);
>> gfor k = 1:n
    >> B(k) = A(k,:) * A(:,k);  % vector-vector multiply
>> gend

>> A = gones(n,n,m);
>> [B C] = deal(gones(n));
>> gfor k = 1:m
    >> C(:,k) = A(:,:,k) * B;   % matrix-vector multiply
>> gend

>> A = gones(n,n,m);
>> B = gones(n);
>> gfor k = 1:m
    >> A(:,k) = A(:,k) * B; % matrix-matrix multiply
>> gend
```
The iterator can be involved in expressions.

```matlab
>> A = gones(n,n,m);
>> B = gones(n);
>> gfor k = 1:2:m
     A(:,:,k) = k*B + sin(k+1);  % expressions
>> gend
```

More complicated subscripting is also supported.

```matlab
>> A = gones(n,n,m);
>> B = gones(n,10);
>> gfor k = 1:2:m
     A(:,1:10,k) = k*B;  % subscripting
>> gend
```

Within the loop, you can use a result you just computed.

```matlab
>> [A B C] = deal(gones(n));
>> gfor k = 1:n
     A(:,k) = 4 * B(:,k);
     C(:,k) = 4 * A(:,k); % use it again
>> gend
```

Here's another way to write that.

```matlab
>> [A B C] = deal(gones(n));
>> gfor k = 1:n
     a = 4 * B(:,k);
     A(:,k) = a;
     C(:,k) = 4 * a;
>> gend
```

You can read and modify a result in place as long as the accesses are independent.

```matlab
>> A = gones(n);
>> gfor k = 1:n
     A(:,k) = sin(k) + A(:,k);
>> gend
```

The same subscripting and assignment behaviors used with GPU data also work with `gfor`.

```matlab
>> A = gones(n,n,m,k);
>> m = m * k;  % precompute since cannot have expressions in
>> gfor k = 1:m
     A(:,:,k) = 4*A(:,:,k); % collapse last dimension
>> gend
```
Chapter 5 – Optimizing Performance

Limitations
This preliminary implementation of `gfor` has the following restrictions.

1. No conditional statements in the body of the loop, (i.e. no branching). However, you can often find ways to overcome this limitation. Consider the following two examples:

Example 1: Problem
```matlab
>> A = gones(n,m);
>> gfor k = 1:n
    >> if k > 10  % bad
    >>   A(:,k) = k + 1;
    >> end
>> gend
```

However, you can do a few tricks to overcome this limitation by expressing the conditional statement as a multiplication by logical values. For instance, the block of code above can be converted to run as follows, without error:

Example 1: Solution
```matlab
>> A = gones(n,m);
>> gfor k = 1:n
    >> condition = k > 10; % good
    >> A(:,k) = ~condition*A(:,k) + condition*(k+1);
>> gend
```

Another example of overcoming the conditional statement limitation in `gfor` is as follows:

Example 2: Problem
```matlab
>> A = gones(n,n,m);
>> B = grand(n);
>> gfor k = 1:4
    >> if mod(k,2) ~= 0
       >> A(:,:,k) = B + k;
    >> else
       >> A(:,:,k) = B * k;
    >> end
>> gend
```

Instead, you can make two passes over the same data, each pass performing one branch.

Example 2: Solution
```matlab
>> A = gones(n,n,m);
>> B = grand(n);
>> gfor k = 1:2:4
    >> A(:,:,k) = B + k;
>> gend
>> gfor k = 2:2:4
    >> A(:,:,k) = B * k;
>> gend
```
Chapter 5 – Optimizing Performance

2. Each loop iteration is independent of all other loop iterations.

```matlab
>> B = 0;
>> gfor k = 1:n
  >> B = B + k; % bad
>> gend
```

3. No cell array assignment.

```matlab
>> gfor k = 1:n
  >> A{k} = k; % bad
>> gend
```

4. Nesting `gfor`-loops within `gfor`-loops is unsupported. You may interleave `for`-loops as long as they are completely independent of the `gfor`-loop iterator.

```matlab
>> gfor k = 1:n
  >> gfor j = 1:m % bad
  >> % ...
  >> gend
>> gend
```

5. `gfor` must be on a line by itself and trailing comments are allowed.

```matlab
>> gfor k = 1:n; A(:,k) = k;
>> gend % bad
>> gfor k = 1:n % this comment is okay
  >> A(:,k) = k;
>> gend
```

6. Do not use the iterator after `gend`. Its value will not be that of the final iteration.

```matlab
>> gfor k = 1:n
  >> % ...
  >> gend
  >> A = A / k; % bad
```

7. Since each computation is done in parallel for all iterator values, you need to have enough card memory available to do all iterations simultaneously. If the problem exceeds memory, it will trigger standard “out of memory” errors.

8. Jacket will throw a warning if you try to use `i` as an iterator name for `gfor`. Within MATLAB functions, the `gfor` iterator must not use the variable names `i` or `j`, since these are reserved for complex variables (this is a MATLAB bug). Use instead, `k` or some other variable.