MATLAB's Racing Jacket

For those conducting intensive calculations on large data sets, recent advances in GPU computing have been little short of a revelation; no more waiting days for results, or having to invest in your own giant cluster or supercomputer. On that basis, GPU computing seems a natural fit with quants' favourite MATLAB, which probably explains why the people at Accelereyes dreamt up their Jacket application for GPU-enabling MATLAB. Andy Webb takes it for a gallop round the Automated Trader track.

Anyone opening Automated Trader for the first time wouldn't take long to realise that our readers spend quite a lot of their time thinking about speed, which means that we all spend a corresponding amount of time writing about it. Until now, much of our focus has been on speed at the point of trade execution; fastest connectivity, colocation, in-processor-cache matching engines – and so on. We haven't expended so many words on speeding up offline calculations, such as the severe number crunching required when optimising portfolios or parameters for trading models.

That's all about to change, for while there's nothing to stop you using Accelereyes Jacket as part of a real time trading program (though in view of MATLAB API calls, perhaps not a very high frequency one), it has exceptional potential for speeding up the sort of offline calculations mentioned above. Though, as will be seen, "speeding up" is something of an understatement...

The basics

Jacket is a companion application for MATLAB that provides access to GPU hardware that is compliant with NVIDIA's CUDA architecture. While CUDA is a C language based environment, you don't have to write C code to use Jacket with MATLAB. Jacket protects you from all the heavy lifting implicit in C programming by providing a layer of abstraction that enables you to carry on writing standard MATLAB code (with a few minor changes) while still being able to take advantage of the CUDA environment and associated CUDA-enabled GPU hardware.

Perhaps the most striking things about Jacket are the simplicity and depth of its conception. Rather than creating a whole pile of new GPU-specialised MATLAB functions, Jacket goes right to the heart of matter by providing a dedicated set of GPU data structures that can be used instead of their corresponding standard MATLAB equivalents. By using a Jacket data structure calculations and manipulations relating to that structure will be automatically done on your GPU, not your CPU. (We hate to think what the Accelereyes guys had to get up to behind the scenes in C/CUDA world to pull off this rather handy trick, but we're just glad it was them not us.)

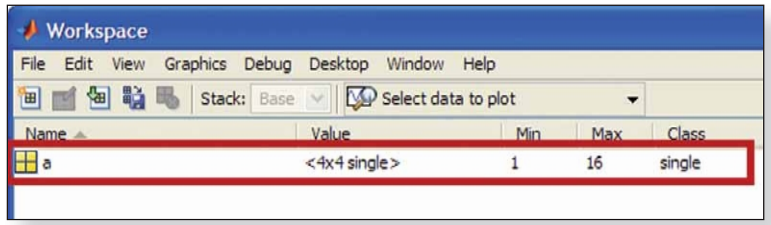
There are a total of eight different Jacket data structures for MATLAB. To take the simplest example, the Jacket "gsingle" structure casts a MATLAB matrix to a single precision floating point GPU matrix. So entering the following in the MATLAB command window:

```
a = single([1 2 3 4; 5 6 7 8; 9 10 11 12; 13 14 15 16]);
```

...creates a simple 4 by 4 matrix of standard MATLAB type single, as can be seen by subsequently entering "a" (without quotes) in the command window, which generates the following output:

```
a =
   1    2    3    4
   5    6    7    8
   9   10   11   12
  13   14   15   16
```

...also confirmed by a quick glance at the MATLAB Workspace window (see Figure 1).



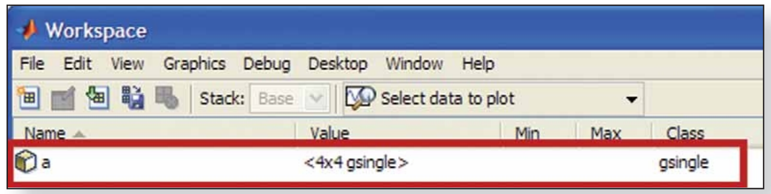
Name	Value	Min	Max	Class
a	<4x4 single>	1	16	single

Figure 1

Then, by entering the following command, we enter Jacket GPU world:

```
a= gsingle(a);
```

The MATLAB Workspace window provides confirmation of the change (see Figure 2).



Name	Value	Min	Max	Class
a	<4x4 gsingle>			gsingle

Figure 2

From here on (unless we cast it to another data type) functions applied to the variable "a" will (if supported by Jacket - more on this later) be conducted on the GPU, rather than the CPU. There is a performance penalty to casting data structures to and fro between Jacket and MATLAB data types, because in effect you are shunting data to and fro between CPU and GPU memory. However, if you have a task that can be massively accelerated by the GPU, but that also requires a call to a MATLAB function not yet supported by Jacket, it is still well worth moving the data back onto the CPU. All that is required is:

```
a= single(a);
```

The hardware

Before moving on to examine Jacket's capabilities in more detail, a word about the GPU hardware on which it runs. As mentioned earlier, Jacket provides a layer of abstraction between MATLAB and NIVIDIA's CUDA environment, which takes advantage of CUDA-enabled hardware. At present the term 'CUDA-enabled hardware' covers a broad range of NVIDIA graphics and dedicated GPU computing cards, including some 80 GeForce cards (minimum 256Mb memory required), 40+ Quadro cards, plus NVIDIA's dedicated high performance computing (HPC) Tesla hardware.

While you obviously get what you pay for in terms of performance, this extensive coverage means that there's a good chance that plenty of existing standard workstations are already CUDA-enabled, by virtue of their installed graphics cards. Even if a workstation isn't already CUDA-enabled, the costs of making it so are hardly prohibitive. For example, a video card like the top of the range GeForce GTX 295 with 1792Mb of memory can be picked up for less than USD300 with a bit of searching – and that delivers 480 CUDA processing cores. (Which when you consider that a decent CPU might have only four cores, rather puts things into perspective.) Dedicated Tesla hardware is rather pricier; for example, the Tesla C1060 (see Figure 3) card with 4Gb of memory can be found for around USD1200.

Our hardware rig for this review consisted of a workstation fitted with a Xeon W3570 3.2Ghz CPU (four cores), 12Gb of RAM and three Tesla C1060s. The motherboard was an Asus P6T7 WS with seven PCIe x16 Gen2 slots (PCIe x16 Gen2 slots are necessary to get the best out of the C1060s; lower spec slots will work but throttle their performance.) The graphics card was an NVIDIA Quadro NVS 450, which is CUDA-enabled, but wasn't used as part of the Jacket testing. While the single W3570 CPU certainly meant the machine wasn't top of the line in CPU terms, this setup nevertheless allowed us to make reasonably realistic comparisons of MATLAB's performance in three modes:

- CPU with implicit multiprocessing[1] turned on (the default in MATLAB since R2007a).
- CPU with the MATLAB Parallel Computing Toolbox (using all four of the CPUs cores)
- Jacket using a single C1060

▸

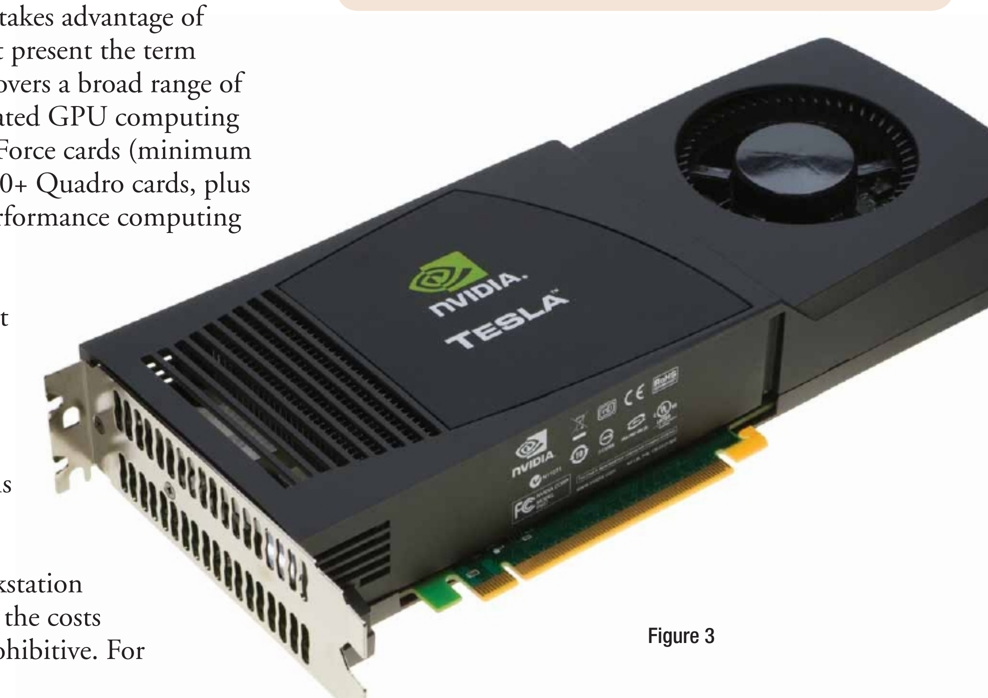


Figure 3

---

[1] *Implicit Multiprocessing - multithreaded computation runs in a single instance of MATLAB and generates simultaneous instruction streams on a multiple CPU (multiprocessor or multi core) system. The multiple processors share the memory of a single computer. The work to be processed is implicitly partitioned for execution on multiple threads. Multithreaded computation in MATLAB is enabled by default. When enabled, MATLAB automatically detects the number of CPUs on the system and recommends the number of threads based on that. On the assumption that few if any MATLAB users would still be running MATLAB on a single core machine, implicit multiprocessing was left on for the purposes of the review.*

In addition, Jacket has recently added support for MATLAB's Parallel Computing Toolbox (PCT) and Distributed Computing Server (DCS). This enables the use of multiple items of CUDA-enabled hardware on a single machine (PCT) or multiple machines (DCS). Unfortunately, due to hardware availability and time constraints, we were unable to conduct any meaningful testing of this by the time we went to press. However, an update on this will be published on www.automatedtrader.net in March.

A magic start

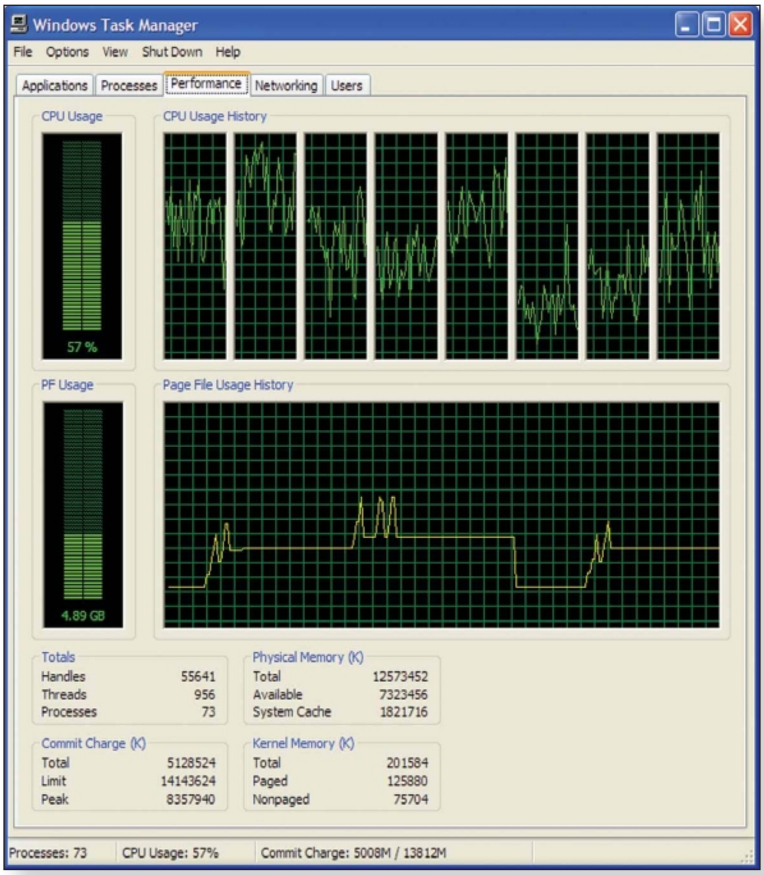
We kicked our testing off with the simple (but not particularly useful) exercise of multiplying two magic squares[2]. The code below creates two magic squares X and Y, which are 10,000 rows by 10,000 columns in size, multiplies them together and assigns their output to the variable A.

```
tic;

Y = magic(10000);
X = magic(10000);
A= X*Y;

time = toc;
```

As Figure 4 shows, MATLAB's implicit multiprocessing made a decent job of distributing the workload across the test workstation's CPU (Note: 'CPU Usage History' shows eight threads, not four cores). Using just the CPU with implicit multithreading turned on, this script took 63.063 seconds to complete.

By contrast, the Jacket version of the script (which simply converted the magic number matrices to the Jacket gdouble data type before multiplying) took 21.1643 seconds to complete; a worthwhile improvement.

In both cases, these results were the average achieved over ten consecutive attempts. This approach of running multiple consecutive attempts can in some circumstances distort the results, because "warming up" a computation under CUDA often tends to reduce the execution time. Therefore the first execution of the Jacket version of the code is often the slowest by an appreciable margin. However, in this case the Jacket version of the code performed almost identically on each iteration.



Totals
Handles 55641
Threads 956
Processes 73

Physical Memory (K)
Total 12573452
Available 7323456
System Cache 1821716

Commit Charge (K)
Total 5128524
Limit 14143624
Peak 8357940

Kernel Memory (K)
Total 201584
Paged 125880
Nonpaged 75704

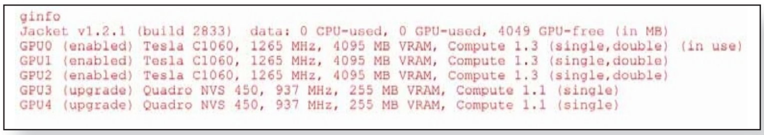
Processes: 73 CPU Usage: 57% Commit Charge: 5008M / 13812M

Figure 4

The performance gap when conducting the same test using random number matrices (rather than magic squares) of the same size was rather larger - 51.9049 seconds and 10.7688 seconds respectively (both figures also an average of ten attempts).

Squeezing to the max

As mentioned earlier, GPU computations can run faster as they "warm up"; the first run of code in a new MATLAB session will often be slower than subsequent passes through the same code. You can also warm up Jacket to improve its initial performance by running its "ginfo" command:



```
ginfo
Jacket v1.2.1 (build 2833)  data: 0 CPU-used, 0 GPU-used, 4049 GPU-free (in MB)
GPU0 (enabled) Tesla C1060, 1265 MHz, 4095 MB VRAM, Compute 1.3 (single,double) (in use)
GPU1 (enabled) Tesla C1060, 1265 MHz, 4095 MB VRAM, Compute 1.3 (single,double)
GPU2 (enabled) Tesla C1060, 1265 MHz, 4095 MB VRAM, Compute 1.3 (single,double)
GPU3 (upgrade) Quadro NVS 450, 937 MHz, 255 MB VRAM, Compute 1.1 (single)
GPU4 (upgrade) Quadro NVS 450, 937 MHz, 255 MB VRAM, Compute 1.1 (single)
```

Figure 5

However, this only warms up Jacket and not the underlying NVIDIA CUDA environment. Fortunately one of Jacket's users (Torben Larsen, Professor in Radio Frequency Electronics and Systems at Aalborg University, Denmark) has written a rather neat function that deals with this issue, which you can find here:

▶

---

[2] MATLAB's magic function magic(n) returns an n-by-n matrix constructed from the integers 1 through n^2 with equal row and column sums. The order n must be a scalar greater than or equal to 3.

http://www.accelereyes.com/wiki/index.php?title=TL-Warm_up_of_Jacket_and_CUDA

Running this prior to any heavyweight production code should ensure optimal performance.

As explained in our MATLAB review (Q2 2009 issue), MATLAB vectors and matrices are most efficiently manipulated when stored in columns in contiguous blocks of RAM. If there is a "vectorised" method of programming something that allows the use of contiguous memory blocks, then that is preferred to the 'loop equivalent' method. The same applies to Jacket, which also takes advantage of the inherent parallelism of the MATLAB M-language.

While avoiding unnecessary loops when writing Jacket based functions is important for this reason anyway, a further factor is NVIDIA's nvcc compiler. Compiling kernels with nvcc is computationally expensive and it is important to avoid doing this by inadvertently using iterating parameters that might force an nvcc compilation on each iteration of a loop. The classic way to fail to do this is to have a mix of Jacket and MATLAB data types used in a calculation in a loop or even in just general calculations. So for example...

```
x = geye(5000);

for y = 1:10000,

x * y;

end
```

...is bad news because it multiplies x (a Jacket data type) by y (a MATLAB data type) within a loop. Needless to say, we managed to fall into this trap almost immediately, when instead we should have used...

```
x = geye(5000);

for y = gdouble(1:10000),

x * y;

end
```

...because both x and y are now Jacket data types, the calculations are pure, not hybrid.

Looping on acid

As mentioned above, the general ideal when writing MATLAB/Jacket code is to avoid unnecessary loops. However, where the use of loops is inevitable, Jacket has a seriously fast ace up its sleeve in the form of its gfor/gend loops. While the standard MATLAB for/end construct conducts each iteration of a loop sequentially, the Jacket gfor/gend construct runs all loop iterations simultaneously. It does this by "tiling out" the values of all loop iterations and then calculating the values of all tiles at the same time using the individual cores on a CUDA-enabled GPU. The acceleration made possible by this is, to put it mildly, pretty significant. For example, the following not very useful pair of conventional nested for/end loops...

```
x = rand(750,750);
y_c = zeros(iterations, iterations, matrixsize);
for k = 1:iterations
    for m = 1:size(x(:,1))
        y_c(m,:,k) = abs(fft(x(m,:)));
    end
end
```

…took an average (ten attempts with negligible difference between each result) of 34.83 seconds with the function inputs "iterations" and "matrixsize" both set to 750. However, the following "Jacketised" code…

```
x = gsingle(rand(750,750));
y = gzeros(iterations, iterations, matrixsize);
for k = 1:iterations
    gfor m = 1:size(x(:,1))
    y(m,:,k) = gforce(abs(fft(x(m,:))));
    gend
end
```

…which replaces just the inner of the two for/end loops with a gfor/gend loop and specifies Jacket data types, completed its first attempt 2.100765 seconds, with all subsequent nine attempts in the range 1.811475 to 1.816984 seconds.

Obviously MATLAB already offers - assuming you buy its PCT - parallel looping of this type with its parfor/end construct. So we thought it might be instructive to compare the performance of parfor and gfor loops with another not very real world example that also uses MATLAB's discrete Fourier transform (fft) function:

```
y= zeros(100,100);
for k=1:100
    x= rand(100);
    parfor m=1:10000,
      y=  fft(x)*m;
    end
end
```

In order to access all the cores on the test rig's CPU for the calculation, we created a MATLAB PCT pool with four workers with the following command...

```
matlabpool open 4
```

...before running the code fragment, which took an average (over ten very similar attempts) of 29.268 seconds to complete, with the MATLAB PCT keeping all four cores well loaded throughout the test.
By comparison, the Jacket version of the code:

```
y= gzeros(100,100);
for k=1:100
    x= grand(100);
    gfor m=1:10000,
       y = fft(x)*m;
       gforce(y);
    gend
end
```

…took an average of 4.1538 seconds. To be fair to MATLAB's parfor loop, while the difference is obviously very significant, a major part of the work load is the ten thousand iterations in the nested inner loop. The parfor loop can only distribute this across four cores and while each of those cores was running at a considerably faster clock speed than those on the GPU, the GPU has 60 times as many cores at its disposal. This obviously gives the Jacket gfor/gend construct something of head start when tiling out loops.

What you can do, what you can't

We've been tracking Jacket for a while and have noticed that Accelereyes has a pretty aggressive timeline when it comes to extending coverage of native MATLAB functions. At the back of the Jacket manual is a grid showing a comprehensive list of MATLAB functions together with four completion categories: fully supported, partially supported, soon to be supported and not supported. From looking at the manual for version 1.2 of Jacket dated October 1st 2009 and comparing it with the manual for the current version that we tested (version 1.2.2 dated January 4th 2010) it's reassuring to note the migration ▶

rate (in the right direction!) across the support columns.

During the review, we spotted one significant (partial) omission from the list of supported MATLAB functions – left matrix divide (mldivide or \ in MATLAB) for double precision values (or in Jacket's case gdouble). Single precision values are supported, but unfortunately there is plenty of legacy code out there using mldivide that needs double precision and is thus not fully portable to Jacket. There's also quite a lot of code out there - and certain members of the Automated Trader review team should hang their heads in shame at this point – that abuses the MATLAB inv function to solve the system of linear equations when they should be using mldivide anyway, so this is an opportunity for faster/better code on two counts. (Those same sinful members of the review team can also dream on, because Jacket doesn't currently support inv either.) Happily, when we raised this with Accelereyes we were told that gdouble support for mldivide should appear soon in Jacket version 1.3.0 or 1.3.1.

Lazy execution and vanishing variables

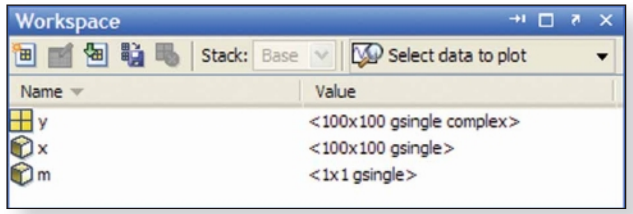
One important thing to be aware of is that Jacket employs a lazy execution design to ensure optimal performance, which means that it does not launch GPU kernels until the results are requested, either in a display or subsequent CPU-based computation. (There are some exceptions to this rule, such as preventing kernels becoming too large to run on the GPU).

Needless to say, we missed the significance of this when conducting our initial testing. To begin with, we were getting demented performance on substantial calculations; except to be more accurate they **would** have been substantial calculations – if we had actually forced Jacket to do them. To repeat a code fragment from above...

```
y= gzeros(100,100);
for k=1:100
    x= grand(100);
    gfor m=1:10000,
       y = fft(x)*m;
       gforce(y);  ◄
    gend
end
```

...our omission and the source of our embarrassment is arrowed and in bold. As its name implies, the gforce command *compels Jacket to calculate values.* The fragment above took an average of 4.1538 seconds to run; but before we added the gforce command – 0.0845 seconds. Cough…

We would stress that this absolutely isn't a problem (quite the opposite in fact in terms of efficiency) but it is definitely something to bear in mind. A similar point – and one that we would cite as a partial excuse for our gforce debacle – is the case of the vanishing variable. We noticed that when we completed code test runs, the names of Jacket data type variables and their description appeared as normal in the MATLAB Workspace window (see Figure 6).



Name	Value
y	<100x100 gsingle complex>
x	<100x100 gsingle>
m	<1x1 gsingle>

Figure 6

However, if you double clicked (in this case on x) to inspect the contents of a variable, it appeared (see Figure 7) to be empty...

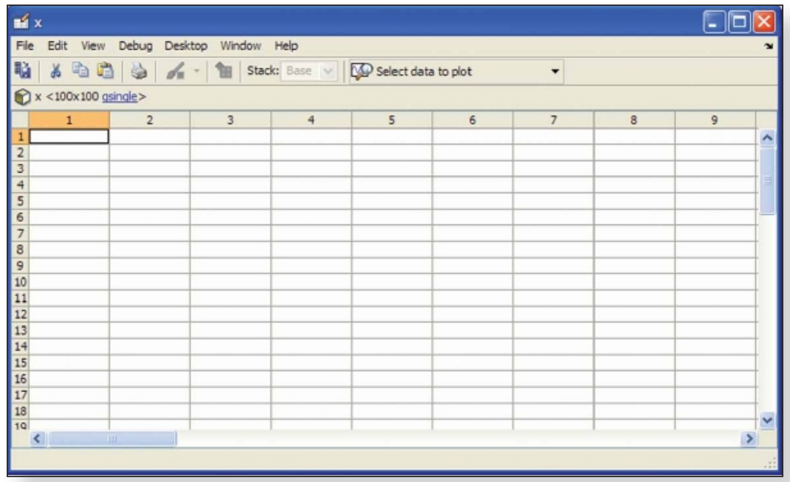
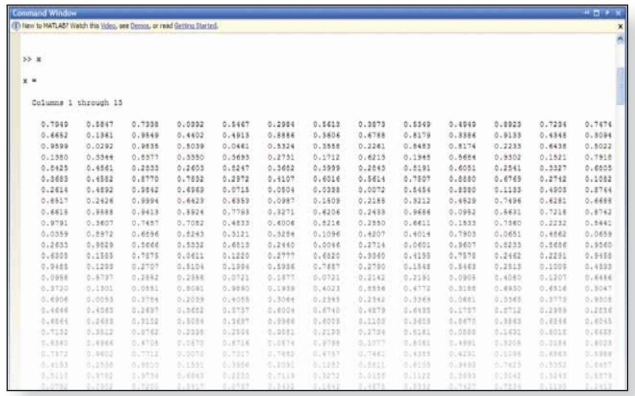


Figure 7

...but if you output to the MATLAB Command Window (see Figure 8) you got numbers - lots of numbers:

It turns out that you cannot as yet see Jacket variable contents in the MATLAB variable inspector. However, as an alternative to clogging up the Command Window, you can simply cast the variable back to a MATLAB type with...

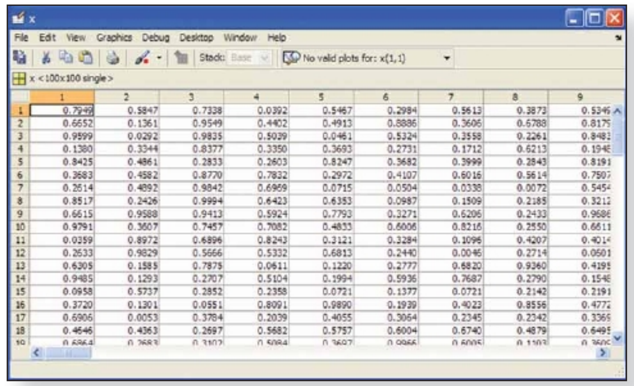
```
x = single(x);
```


Command Window
New to MATLAB? Watch this Video, see Demos, or read Getting Started.

```
>> x

x =

  Columns 1 through 13
```

Figure 8

...and your data magically reappears in the variable inspector (see Figure 9).


x
File Edit View Graphics Debug Desktop Window Help
Stack: Base
No valid plots for: x(1,1)
x <100x100 single>

	1	2	3	4	5	6	7	8	9
1	0.7949	0.5847	0.7338	0.0392	0.5467	0.2984	0.5613	0.3873	0.534
2	0.6652	0.1361	0.9549	0.4402	0.4913	0.8886	0.3606	0.6788	0.817
3	0.9599	0.0292	0.9835	0.5039	0.0461	0.5324	0.3558	0.2261	0.848
4	0.1380	0.3344	0.8377	0.3350	0.3693	0.2731	0.1712	0.6213	0.194
5	0.8425	0.4861	0.2833	0.2603	0.8247	0.3682	0.3999	0.2843	0.8191
6	0.3683	0.4582	0.8770	0.7832	0.2972	0.4107	0.6016	0.5614	0.750
7	0.2614	0.4892	0.9842	0.6969	0.0715	0.0504	0.0338	0.0072	0.545
8	0.8517	0.2426	0.9994	0.6423	0.6353	0.0987	0.1509	0.2185	0.321
9	0.6615	0.9588	0.9413	0.5924	0.7793	0.3271	0.6206	0.2433	0.968
10	0.9791	0.3607	0.7457	0.7082	0.4833	0.6006	0.8216	0.2550	0.6611
11	0.0359	0.8972	0.6896	0.8243	0.3121	0.3284	0.1096	0.4207	0.401
12	0.2633	0.9829	0.5666	0.5332	0.6813	0.2440	0.0046	0.2714	0.0601
13	0.6305	0.1585	0.7875	0.0611	0.1220	0.2777	0.6820	0.9360	0.419
14	0.9485	0.1293	0.2707	0.5104	0.1994	0.5936	0.7687	0.2790	0.154
15	0.0958	0.5737	0.2852	0.2358	0.0721	0.1377	0.0721	0.2142	0.2191
16	0.3720	0.1301	0.0551	0.8091	0.9890	0.1939	0.4023	0.8556	0.477
17	0.6906	0.0053	0.3784	0.2039	0.4055	0.3064	0.2345	0.2342	0.336
18	0.4646	0.4363	0.2697	0.5682	0.5757	0.6004	0.6740	0.4879	0.649

Figure 9

Again, this isn't a major problem, but it would be good if it was flagged up in the docs; a point that we - once our collective heart rate returned to normal - emphasised to Accelereyes.

Horses for courses

The performance kicker that Jacket delivers to MATLAB is so impressive that the natural inclination is to Jacketise all your code and try to do everything on the GPU. In practice, things aren't so straightforward. Simply chucking everthing onto the GPU doesn't work; small calculations or inherently serial operations are more efficiently performed on the CPU. A further consideration (see "*What you can do, what you can't*" above) is of course whether Jacket supports the functions or operations you are trying to use.

Having said all that, if you have large datasets to mince in MATLAB, then Jacket is excellent value. As our review shows, the performance boost obviously depends upon what you are doing, but so many of Jacket's strongest suits (large matrix manipulation etc) relate to tasks commonplace for Automated Trader readers. If Jacket covers your MATLAB function bases and you're otherwise considering filling a data centre with a CPU-based cluster, then this is your chance to save the planet.

Is it worth it?

If the preceding paragraph describes the sort of work you do, then the answer is a resounding yes. A standard license allows you to run Jacket on a single GPU and costs USD1750. Even after you factor in the cost of CUDA-enabled hardware - which you may already have anyway - that looks a serious bargain. Adding licenses for additional GPUs on the same machine costs USD750 a pop up to a total of four GPUs, though if you want to make the most of these additional units in MATLAB you'll also need to buy the MATLAB PCT. Real speed freaks will probably want to cut straight to the chase with the 8 GPU HPC Cluster License, which costs USD7250 and also includes support for the MATLAB DCS.

The thing that probably most impressed us about Jacket (apart from the raw speed) was the simplicity. The creation of GPU data types for MATLAB was a masterstroke because of the transparency it delivers to the end user. In many cases, just changing a handful of variables from MATLAB to Jacket data types can boost performance dramatically. And if loops are your thing, gfor/gend simply rocks.

The road ahead looks promising too; Accelereyes have steamed ahead in expanding their MATLAB function support and the prospects of combining Jacket with NVIDIA's Fermi technology when it hopefully lands in late Q1/early Q2 are frankly mouth watering.

So, what's our view? Put it this way; in the world of MATLAB 'big numbers', conventional CPUs can only take you so far; Jacket takes you the rest of the way – and then some...

Hat tips

In addition to Accelereyes support, three people were instrumental in helping the Automated Trader team put together this review:

- Pete Dhillon of 3D Computers - www.3d-computers.co.uk – for providing one of 3D's HPC workstations for this review and getting NVIDIA involved when their distributor wasn't.
- Mike Walsh of NVIDIA – for turning out at very short notice, at great inconvenience, to sort out something in minutes that NVIDIA's distributor had failed to sort in weeks.
- Torben Larsen of Aalborg University – for sharing his Jacket/MATLAB expertise with us and for preventing us from making assorted dumb mistakes.

Gentlemen - many thanks.