Parallel High Performance Computing

With Emphasis on Jacket Based GPU Computing

Jacket Introduction

Torben Larsen
Aalborg University

Outline

Jacket Teaser

Jacket Teaser

What is Jacket? Why use it?

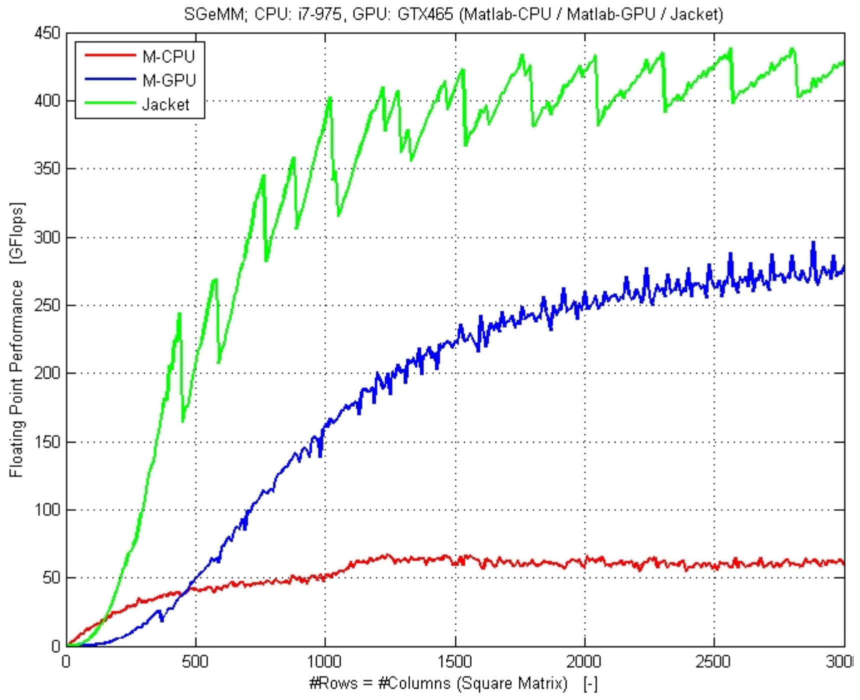
- **Jacket is an almost transparent overlay to MATLAB enabling easy access to GPUs.**
- **Jacket supports:**
  - Single and multiple GPUs in a single computer/workstation.
  - Single and multiple GPUs in a cluster configuration.
  - Small selection of reserved constructs to enable GPUs.
  - Supports MATLAB PCT (Parallel Computing Toolbox) and DCS (Distributed Computing Server).
  - Compiled stand alone code taking advantage of the GPU.

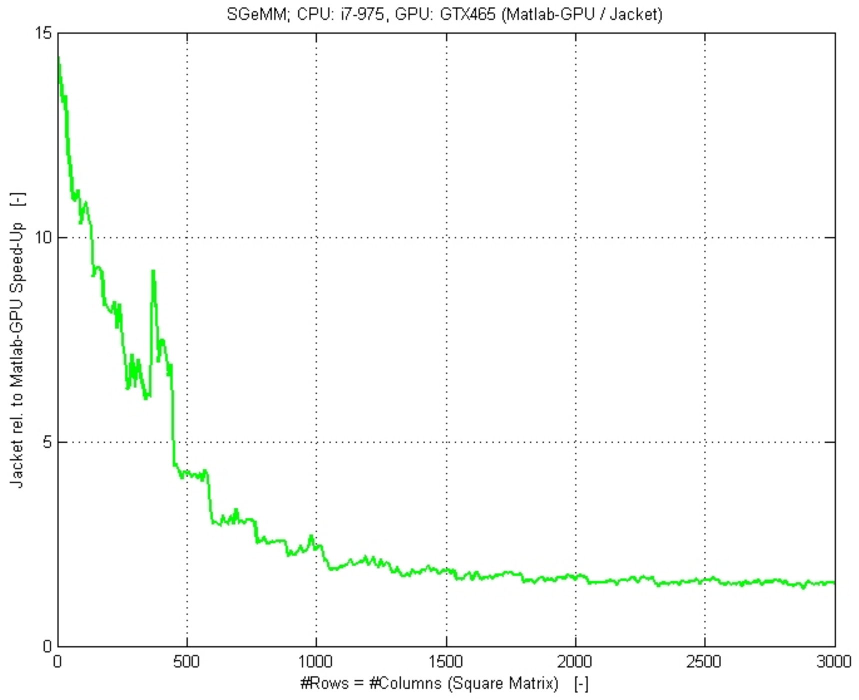
- **But why use Jacket now that MATLAB supports GPU computations (via gpuArray)?**
  - MATLAB not always high performance.
  - MATLAB has very, very modest support for computations on the GPU.
  - MATLAB requires at least CUDA 1.3 capable devices (Tesla C1060 etc.).
  - No lazy execution (inefficient without it – more later but it allows Jacket to collect computations, which is more efficient).
  - No support for multi-GPUs and clusters.
  - No support for double precision linear algebra package.
  - No support for compiled code.

- **But let's have a look at the raw floating point performance …**

Jacket Teaser
CPU vs. GPU Performance

- **MATLAB-GPU and Jacket single precision floating point performance versus square matrix size (complexity) on a GeForce GTX465 GPGPU:**



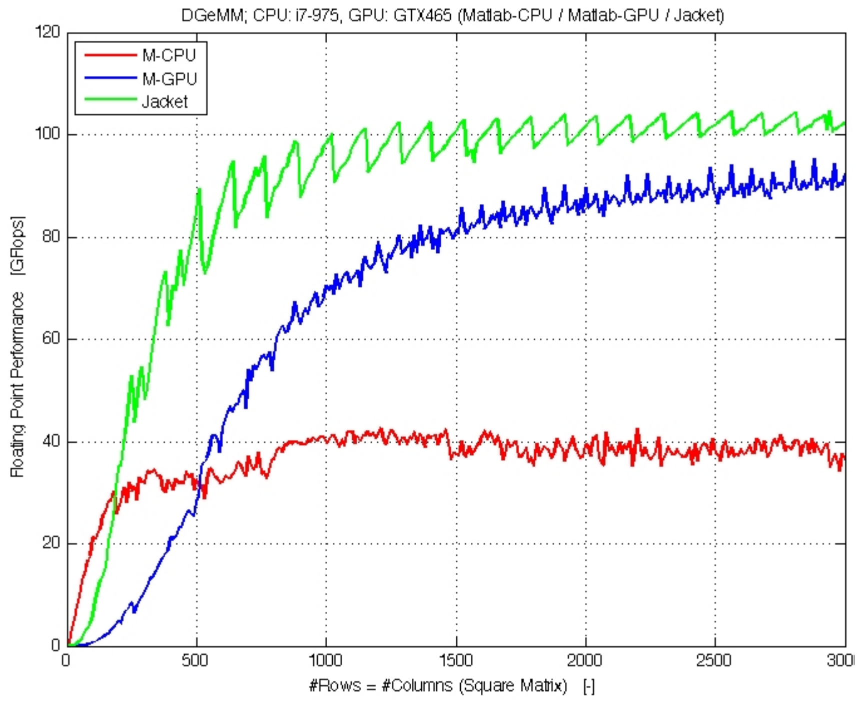


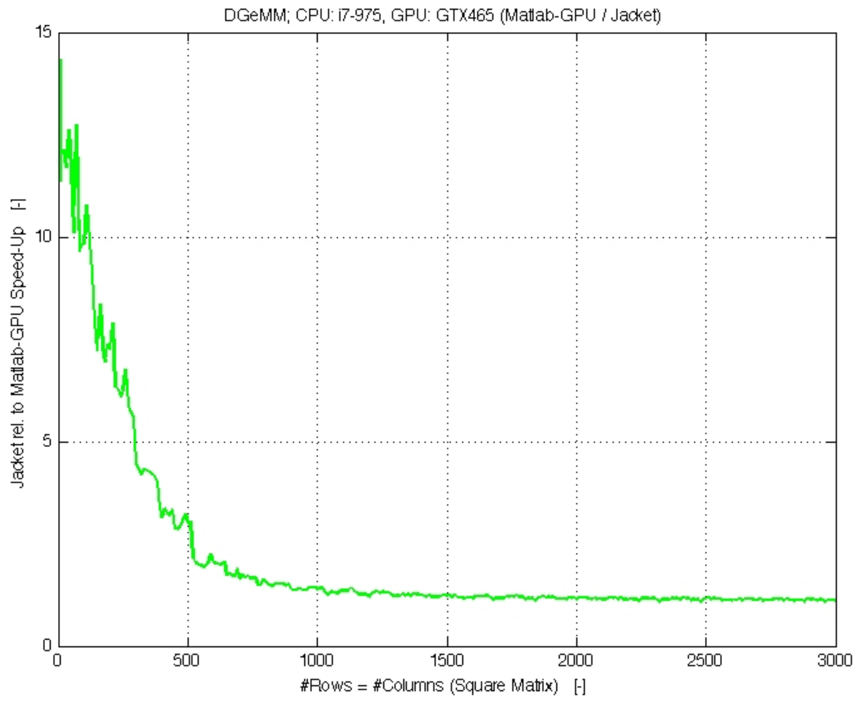
- **The GTX465 is a cheap Fermi based GPU for the mass/gaming market – price is approximately $ 250. Relatively strong in single precision GFLOPS**

Jacket Teaser

CPU vs. GPU Performance

- **MATLAB-GPU and Jacket double precision floating point performance versus square matrix size (complexity) on a GeForce GTX465 GPGPU:**



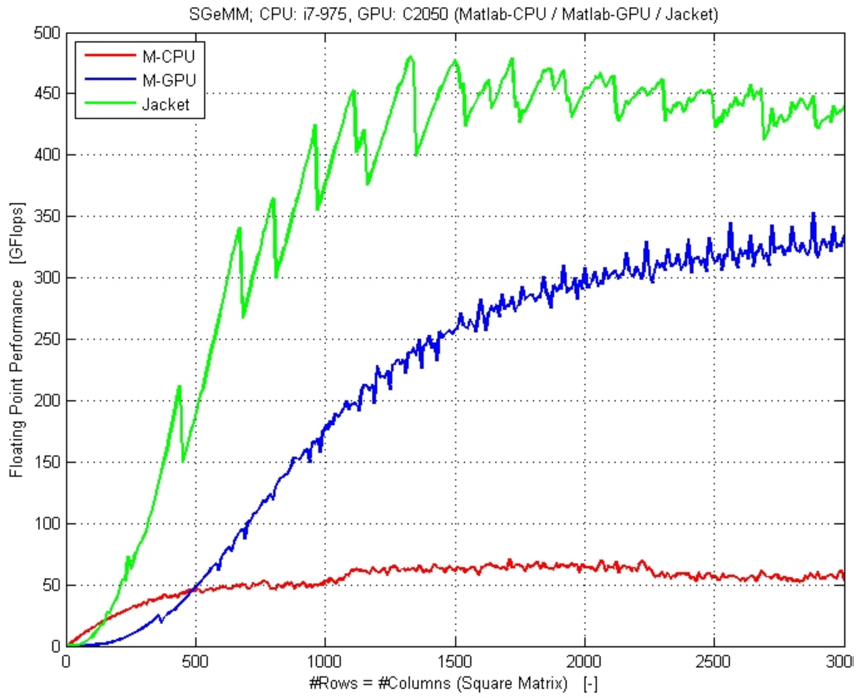


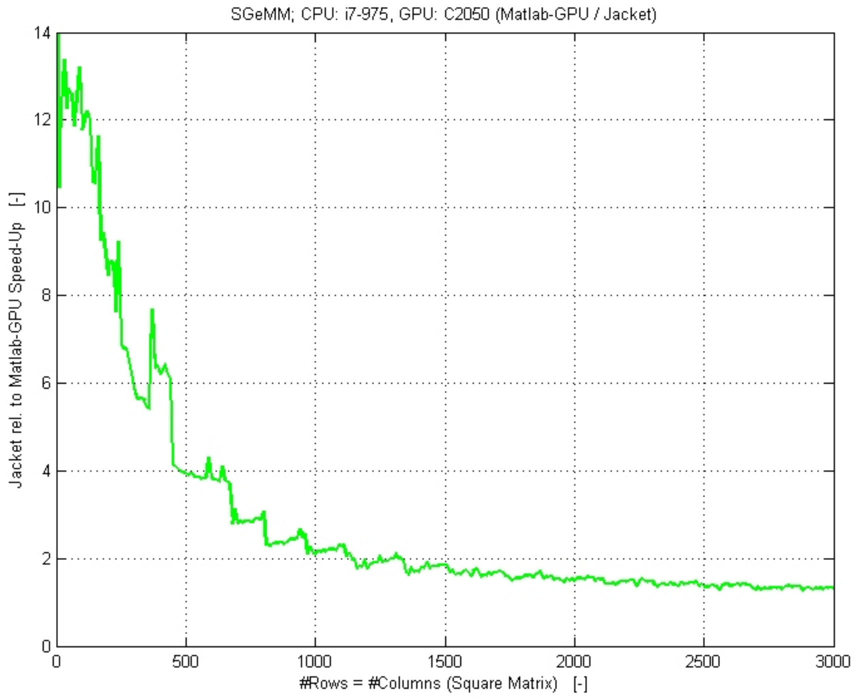
- **As expected NVIDIA GeForce models for the mass/gaming market are weak in double precision performance – double precision isn't of much use in gaming after all**

Jacket Teaser

CPU vs. GPU Performance

- MATLAB-GPU and Jacket single precision floating point performance versus square matrix size (complexity) on a Tesla C2050 GPGPU:



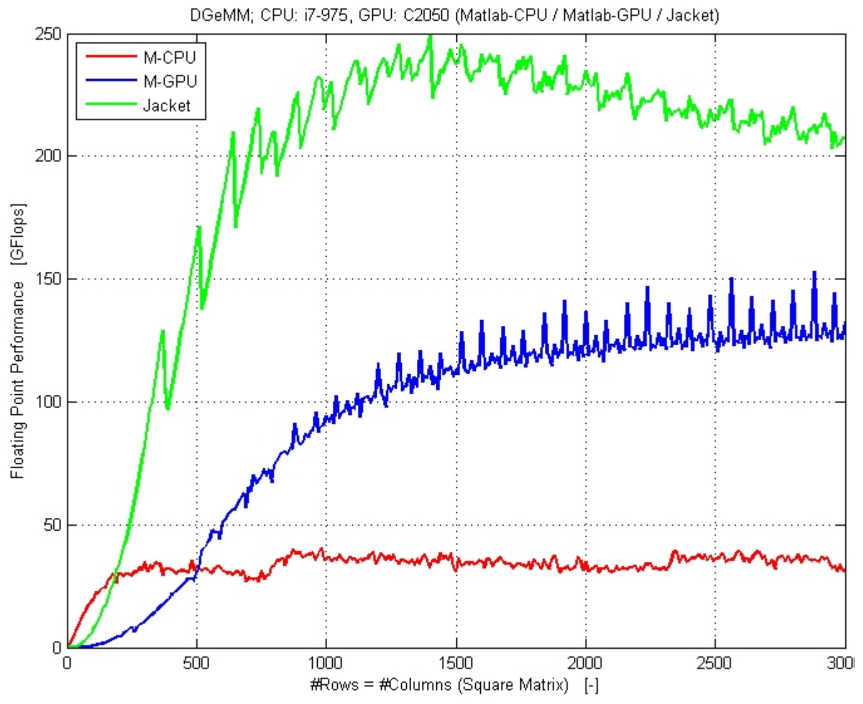


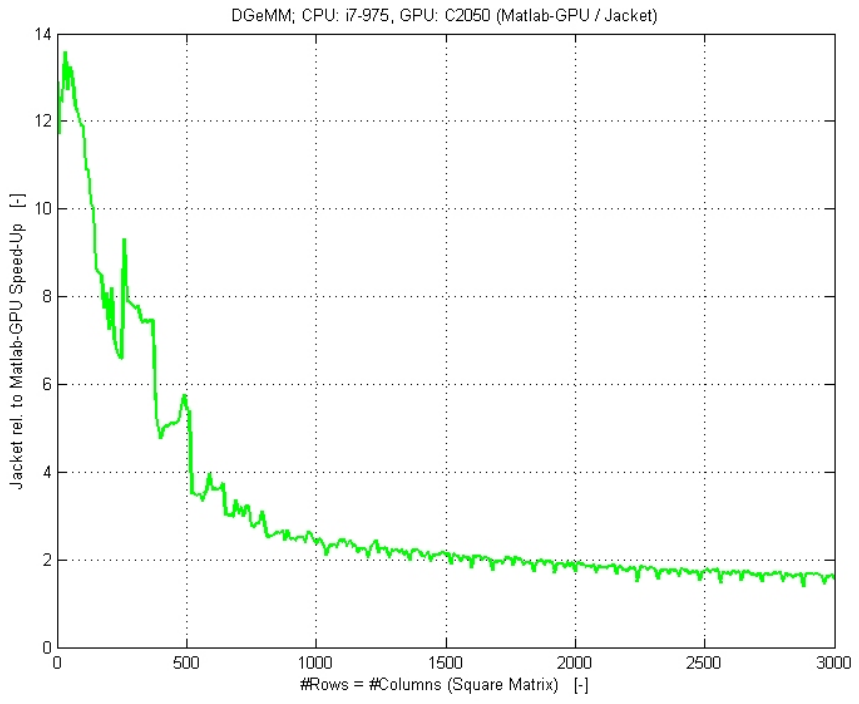
- Jacket clearly superior to MATLAB-GPU for all matrix sizes
- MATLAB-GPU relies on NVIDIAs implementation, which is part of the CUDA SDK for matrix computations while Jacket uses its own implementation
- The cheap GTX465 is not far behind

Jacket Teaser

CPU vs. GPU Performance

- **MATLAB-GPU and Jacket single precision floating point performance versus square matrix size (complexity) on a Tesla C2050 GPGPU:**





- **Here the GTX465 is left in the dust. The C2050 delivers about a factor 2.5 of the GTX465:**
  - **GTX465** price: approx. **$ 250**
  - **C2050** price: approx. **$ 3,000**

Jacket Teaser

CPU vs. GPU Performance

- **A number of different functions have been compared by real life measurements using MATLAB on the CPU versus Jacket using the GPU**
- **Hardware platform:**
  - Motherboard: Asus P6T7 Supercomputer
  - Memory: 6 x 4 GB, 1333 MHz DDR3 RAM
  - CPU: Intel Core i7-975, 3.33 GHz (4 cores)
  - GPU: NVIDIA Tesla C2050 (approx. 2.5 GB memory using ECC, 448 cores)
  - Disks: 5 x Seagate 15,000 RPM
- **Software:**
  - OS: Microsoft Windows 7 x64 Enterprise
  - MATLAB: R2010b (7.11.0.584)
  - Jacket: 1.5.0 (build 8250)
- **Array sizes:**
  - Matrix size: 2000 x 2000 (a 2^N would have been favorable to Jacket)
  - Vector size: 4.00E6 x 1 (a 2^N would have been favorable to Jacket)
  - MATLAB is being allowed to use all CPU cores
  - For svd the matrix size was 200 x 200 and the vector size was 200 x 1

Jacket Teaser

CPU vs. GPU Performance

- **Speed-up of Jacket relative to MATLAB (CPU) measure as "Time for MATLAB divided by time to Jacket":**

Function	Matrix Single	Matrix Double	Vector Single	Vector Double
all	4.60	4.94	9.43	8.06
any	4.57	5.04	9.26	8.38
asinh	50.49	11.70	49.59	11.83
atan2	326.82	93.26	323.51	93.08
atan	39.94	7.41	40.70	7.40
chol	38.79	1.92	---	---
conv2	2.50	---	---	---
cos	27.43	13.50	27.39	13.86
det	2.30	2.14	---	---
exp	46.96	17.02	47.31	17.00

Jacket Teaser

CPU vs. GPU Performance

- **Speed-up of Jacket relative to MATLAB (CPU) measure as "Time for MATLAB divided by time to Jacket":**

Function	Matrix Single	Matrix Double	Vector Single	Vector Double
fft	17.74	3.16	37.21	4.72
find	20.23	19.17	20.11	19.29
ifft	9.26	3.49	27.17	4.72
interp1	---	---	204.53	156.81
interp2	415.02	---	---	---
inv	---	1.85	---	---
load	0.95	1.00	0.95	0.95
log	35.13	12.37	34.57	12.41
lu	2.59	2.44	0.47	0.53
max	1.38	2.50	2.31	2.99
min	1.38	2.51	2.28	3.29

Jacket Teaser

CPU vs. GPU Performance

- **Speed-up of Jacket relative to MATLAB (CPU) measure as "Time for MATLAB divided by time to Jacket":**

Function	Matrix Single	Matrix Double	Vector Single	Vector Double
minus	19.84	10.27	20.22	10.15
mldivide	2.80	2.06	---	---
norm	0.54	0.90	5.24	50.84
plus	19.60	10.09	20.06	10.21
power	43.45	9.63	43.24	9.22
rand	48.15	45.15	47.97	45.40
randn	31.21	18.25	31.10	18.26
rdivide	11.91	6.38	12.05	6.41
save	0.99	0.99	0.99	0.99
sort	5.78	4.67	10.06	3.11

Jacket Teaser
CPU vs. GPU Performance

- **Speed-up of Jacket relative to MATLAB (CPU) measure as "Time for MATLAB divided by time to Jacket":**

Function	Matrix Single	Matrix Double	Vector Single	Vector Double
subsasgn	0.10	0.10	0.01	0.01
sum	1.37	2.52	2.10	3.29
svd	0.06	0.09	0.02	0.04
times	24.41	10.23	24.38	10.18
trapz	2.38	21.48	1.09	1.25

- As seen above, **Jacket is often much, much faster than plain MATLAB**
- **The situations where Jacket is less efficient is in some reductions and where arrays need to be manipulated (e.g. subsasgn)**

Jacket Teaser
MATLAB/CUDA/Jacket

- **MATLAB vs. Jacket vs. CUDA – why not CUDA then? Summing elements in a vector …**

```
%% MATLAB
Acpu = rand(n,1,single?);
sum(Acpu)
```

```
%% Jacket
Acpu = rand(n,1,single?);
Agpu = gsingle(Acpu);
sum(Agpu)
```

```
/* CUDA */
__global__
void kernel(int n, float *d_in, float *d_out)
{
    int tid = threadIdx.x;
    int grid = 128 * gridDim.x;
    int i = 128 * blockIdx.x + tid;
    float sum = 0;
    while (i < n) { sum += d_in[i]; i += grid; }
    __shared__ float e[128];
    e[tid] = sum;
    __syncthreads();
    if (tid < 64) { e[tid] += e[tid + 64]; }
    __syncthreads();
    if (tid < 32) { e[tid] += e[tid + 32]; e[tid] += e[tid + 16];
        e[tid] += e[tid + 8]; e[tid] += e[tid + 4];
        e[tid] += e[tid + 2]; e[tid] += e[tid + 1]; }
    if (tid == 0) { d_out[blockIdx.x] = *e;
}
```

Core i7-920 & Tesla C1060 (5E6 elements):

MATLAB:	1.57 ms
Jacket:	0.337 ms
CUDA:	0.276 ms

CUDA gives a 20% speedup in this case at hugely increased complexity

Jacket Teaser
MATLAB/CUDA/Jacket

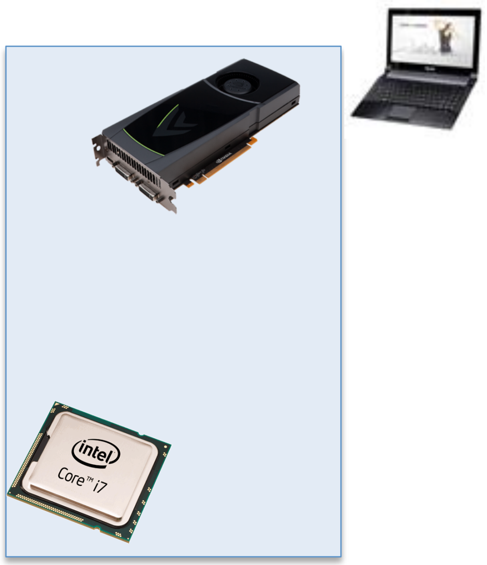
- **Conclusions …**
  - **Jacket is far superior to MATLAB R2010b and R2011a's support of GPUs in all aspects**
    - Far better support of functions, which are GPU enabled
    - Far less restrictions (not necessary to have PCT (Parallel Computing Toolbox) for just a single GPU)
    - Much better performance
  - **CUDA**: Poor CUDA code is way, way slower than Jacket; **Well designed CUDA code may be around 20% faster than Jacket** (depends heavily on the application though). **Poor CUDA code may perform at only 5% of what well designed code delivers**.
  - **Jacket supports a wide range of hardware and configurations:**
    - Single GPU platforms – as long as they have a CUDA enabled GPU (also including laptops)
    - Multi GPUs single platform (requires MATLAB PCT (Parallel Computing Toolbox))
    - Multi GPUs multi platforms (requires MATLAB PCT and DCS (Distributed Computing Server))
    - Support for single as well as double precision
    - Compiled code for stand alone execution

NVIDIA GPU Computing CUDA + jacket = **Near CUDA performance at slightly more than MATLAB complexity**

Jacket Framework

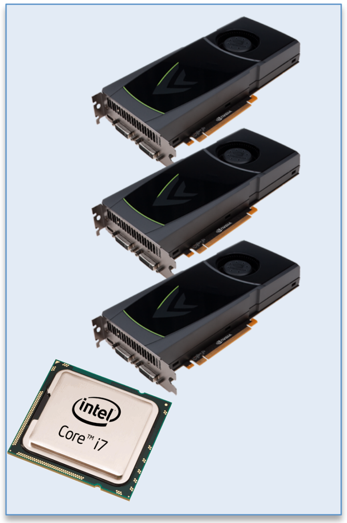
Jacket Framework
Configurations

- **Different Jacket configurations:**

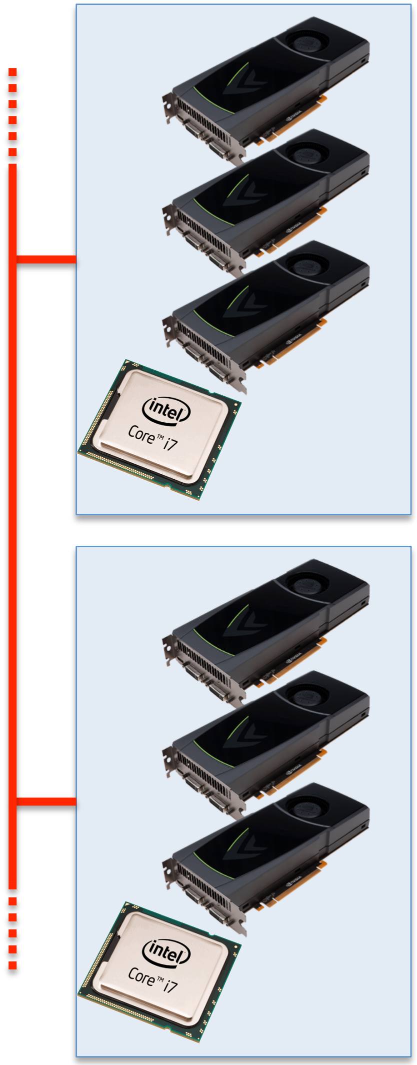


Single-GPU environment
This ranges from small workstations to laptops with CUDA capable NVIDIA GPUs

Multi-GPU environments
The largest single unit known is a Colfax CXT8000, which holds up to 8 Tesla C2070



Jacket for clustered environments



Jacket Framework
Product Types

- **Jacket …**
  - 1 GPU
  - 2-8 GPUs (known as MGL = Multi GPU License)
  - >8 GPUs (known as HPC = High Performance Computing)
  - DLA: Double precision Linear Algebra support. Extra addition to Jacket providing double precision support for e.g. INV, EIG, SVD, QR, LU, and MLDIVIDE. It is based on the premium version of CULA.
  - SLA: Sparse Linear Algebra support.
  - SDK Interface: Software Development Kit
  - JMC Deployment
- **libJacket:**
  - Deployment of Jacket via C/C++ as a library to create stand alone applications without MATLAB
- **Jacket Mobile NDK**:
  - GPU based signal processing on mobile devices (Android at the moment)

Jacket Framework
Platform Design

- **Minimum number of CPU cores:**

$$\#_{\text{cpu cores}} \geq \#_{\text{gpu devices}}$$

- **Minimum amount of CPU memory:**

$$\mathcal{M}_{\text{cpu}} > \sum_{i=1}^{\#_{\text{gpu devices}}} \mathcal{M}_{\text{gpu},i} + \mathcal{M}_{\text{os}}$$

- Is the sum of total GPU memory plus some amount for the operating system and what might be needed for the MATLAB arrays. I would even say that a factor 2 should be multiplied on the summation as we can easily expect that we need memory for both what we have on the GPUs but also for what modified arrays we get back to the MATLAB workspace.
- Therefore, a **conservative memory requirement** could be:

$$\mathcal{M}_{\text{cpu conservative}} > 2 \sum_{i=1}^{\#_{\text{gpu devices}}} \mathcal{M}_{\text{gpu},i} + \mathcal{M}_{\text{os}}$$

- **As an example** 3 x Quadro 4000 (each with 2 GB memory) and we wish to reserve 4 GB for the operating system … here we need 10 GB for the standard requirement and 16 GB for the conservative requirement

Jacket Framework
Platform Design

- **If possible use a separate GPU for handling the display – no computations on that GPU!**
  If you use the display GPU for computations anyway then reduce the display resolution to free GPU memory

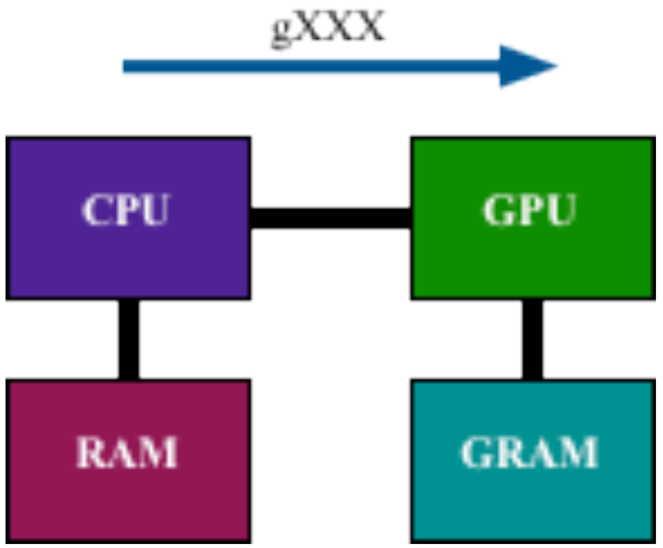
- **Design wise:**
  - **aim for balance;** don't use an expensive and very fast CPU and low performing GPUs
  - Take care of noise and heat in multi-GPU systems – it easily becomes a problem with 3 x 250 Watt GPUs plus the rest
  - **When selecting the GPU consider memory in particular**.
    If you have hard memory requirements it may pay off to buy a version with more memory: for example the GTX580 is made in a 1.5 and a 3 GB version
  - **For the motherboard ensure it supports the fastest PCIe possible** (matching the GPUs) and allows for the needed CPU, memory, and number of GPUs
  - **As for disks there do not appear to be any advantage of using SSD compared to standard rotational disks**

Jacket Fundamentals

Jacket Fundamentals

Data Types / Casting Variables

- **Jacket is essentially an overlay to MATLAB, which consists of a selection of functions to cast MATLAB variables to the GPU**
  - Once variables are on the GPU all subsequent computations are done on the GPU
  - The casting functions are named "gXXX" – e.g. gsingle(x) to cast the variable x from the CPU workspace to the GPU



Jacket Fundamentals
Data Types / Casting Variables

- **Jacket core functions to cast or create data on the GPU:**

Jacket function	Description	Example
gsingle	Casts a MATLAB array to a single precision array on the GPU.	A = gsingle(B);
gdouble	Casts a MATLAB array to a double precision array on the GPU.	A = gdouble(B);

glogical	Casts a MATLAB binary array to a GPU binary array. All non-zero values are set to “1”. The input array can be a CPU or GPU type of array.	A = glogical(B);
		A = glogical(0:4);

gint8, guint8
gint32, guint32
	Cast a MATLAB array to a signed/unsigned 8-bit or 32-bit integer GPU array	A = gint8(B); A = guint8(B);
		A = gint32(B); A = guint32(B);

gzeros	Creates an array on the GPU analogous to “zero”.	A = gzeros(5); A = gzeros(100,10);
gones	Creates an array on the GPU analogous to “ones”.	A = gones(5); A = gones([2,8]);
geye	Creates an identity matrix on the GPU analogous to “eye”	A = geye(5);
grand	Creates a matrix on the GPU analogous to “randn”	A = grand(10,20);
grandn	Creates a matrix on the GPU analogous to “randn”	A = grandn(3);

Jacket Fundamentals
Data Types / Casting Variables

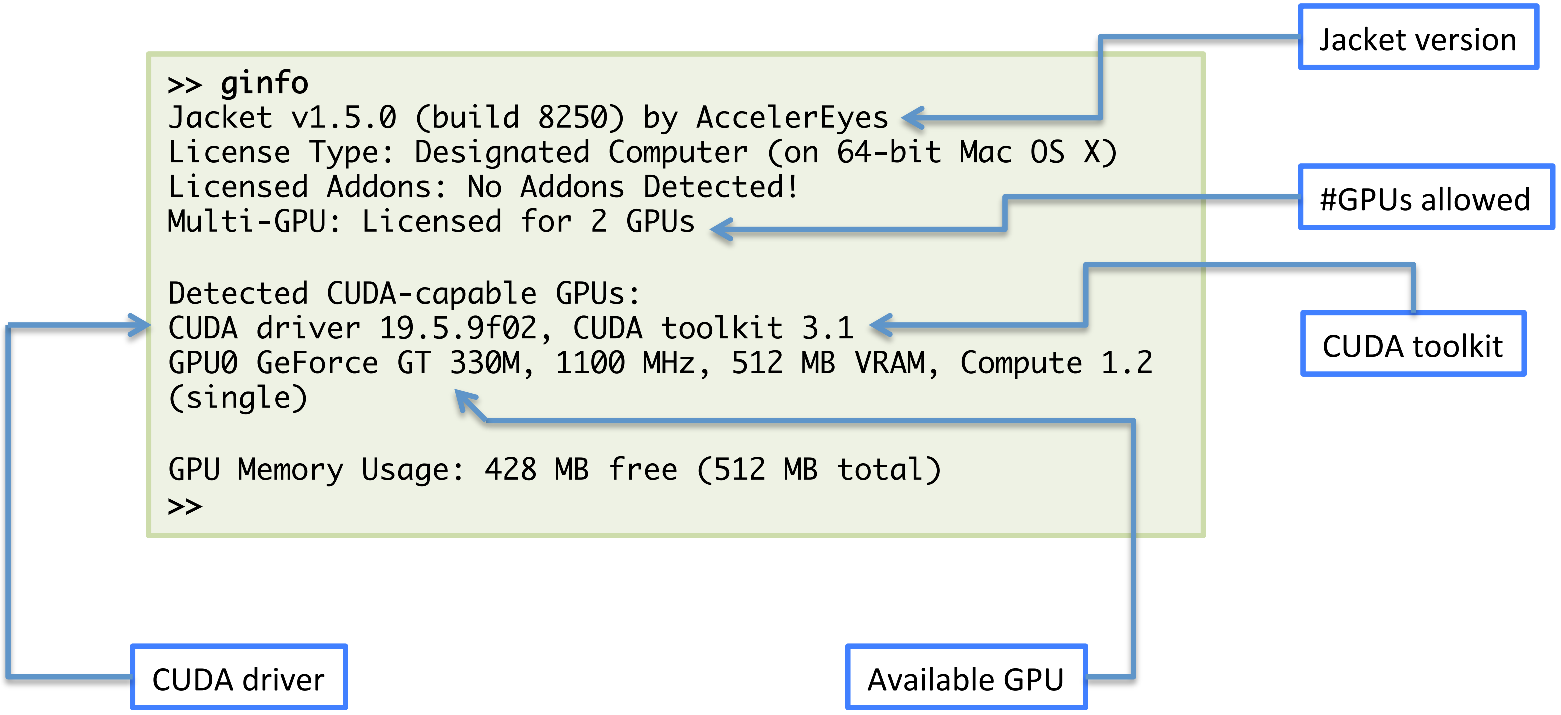
- **Example:** Generate random data on the CPU, move these to the GPU and do some computations on them. Afterwards move the data back to the CPU:

```
>> Acpu = randn(10,10);                                              % Create data on CPU
>> Agpu = gsingle(Acpu);                                             % Copy data to the GPU
>> whos                                                              % Check data types

  Name          Size                Bytes  Class        Attributes

  Acpu         10x10                  800  double
  Agpu         10x10                 1472  gsingle

>> Bcpu = single(Agpu);                                           % Copy data back to the CPU
>> whos                                                           % Check data types

  Name          Size                Bytes  Class        Attributes

  Acpu         10x10                  800  double
  Agpu         10x10                  568  gsingle
  Bcpu         10x10                  400  single

>> Agpu_direct = grandn(10,10);                          % Generate data directly on the GPU
```

Jacket Fundamentals
Data Types / Casting variables

- **Jacket basically contains functions to cast MATLAB variables to the GPU and then all functions applied to those GPU data are automatically performed on the GPU**

- **Example:**



```
>> ginfo
Jacket v1.5.0 (build 8250) by AccelerEyes
License Type: Designated Computer (on 64-bit Mac OS X)
Licensed Addons: No Addons Detected!
Multi-GPU: Licensed for 2 GPUs

Detected CUDA-capable GPUs:
CUDA driver 19.5.9f02, CUDA toolkit 3.1
GPU0 GeForce GT 330M, 1100 MHz, 512 MB VRAM, Compute 1.2
(single)

GPU Memory Usage: 428 MB free (512 MB total)
>>
```

Jacket Fundamentals
Data Types / Casting Variables

- **In case we have a hardware platform with multiple GPUs**
  - **we must select which one to perform the computations**

```
>> ginfo
Jacket v1.7.1 (build 58de35b) by AccelerEyes
License Type: Designated Computer (on 64-bit Windows)
Licensed Addons: DLA
Multi-GPU: Licensed for 4 GPUs

Detected CUDA-capable GPUs:
CUDA driver 263.06, CUDA toolkit 3.2
GPU0 Quadro 2000, 1251 MHz, 994 MB VRAM, Compute 2.1 (single,double) (in use)
GPU1 Quadro 4000, 810 MHz, 2015 MB VRAM, Compute 2.0 (single,double)
GPU2 Quadro 4000, 810 MHz, 2015 MB VRAM, Compute 2.0 (single,double)
GPU3 Quadro 4000, 810 MHz, 2015 MB VRAM, Compute 2.0 (single,double)
Display Device: GPU3 Quadro 4000

GPU Memory Usage: 879 MB free (994 MB total)
```

Jacket Fundamentals
Data Types / Casting Variables

- `gselect(gpu#)` **selects the gpu to use in the MATLAB workspace – GPUs are numbered like GPU0, GPU1, …, GPU*N* in case there are *N*+1 GPUs:**

Select GPU3

```
>> gselect(3)
>> ginfo
Jacket v1.7.1 (build 58de35b) by AccelerEyes
License Type: Designated Computer (on 64-bit Windows)
Licensed Addons: DLA
Multi-GPU: Licensed for 4 GPUs

Detected CUDA-capable GPUs:
CUDA driver 263.06, CUDA toolkit 3.2
GPU0 Quadro 2000, 1251 MHz, 994 MB VRAM, Compute 2.1 (single,double)
GPU1 Quadro 4000, 810 MHz, 2015 MB VRAM, Compute 2.0 (single,double)
GPU2 Quadro 4000, 810 MHz, 2015 MB VRAM, Compute 2.0 (single,double)
GPU3 Quadro 4000, 810 MHz, 2015 MB VRAM, Compute 2.0 (single,double) (in
use)
Display Device: GPU3 Quadro 4000

GPU Memory Usage: 1977 MB free (2015 MB total)
```

Jacket Fundamentals
Data Types / Casting Variables

- `gselect(gpu#)` **must be applied before any Jacket computation** – otherwise you get an error
  - Once a GPU has been put to work you need to close down MATLAB to select another one (or to use SPMD if you have the parallel computing toolbox – described later)

```
>> grandn(2,1).*grandn(2,1)

ans =

    0.6366
    0.1419

>> gselect(2)
??? Error using ==> gpu_entry
gselect: must be called before any computation

Error in ==> C:\Program Files\AccelerEyes\Jacket\engine\gselect.p>gselect at 3

%    Select or query which GPU is in use
```

Jacket Fundamentals
Pass-by-Value Semantics

- **MATLAB uses pass-by-value semantics when transferring data to functions –** in a slightly modified way though
- **Pass-by-value in itself implies that the function makes a local copy of the input data but this doesn't always happen:**

```
function [ A ] = pass1( A )

end
```

```
function [ A ] = pass2( A )
  A(1,1) = 1111;
end
```

- If we time these two functions with the input matrix being very large (20000×20000) we get:

```
N = 20000;   RPT = 20;
A = randn(N,N,'single');
tic;  for ii=1:RPT,  R1 = pass1(A);
end;  toc
```

```
N = 20000;   RPT = 20;
A = randn(N,N,'single');
tic; for ii=1:RPT,  R2 = pass2(A);  end;
toc
```

```
>> master_pass1
Elapsed time is 0.061896 seconds.
>>
```

```
>> master_pass2
Elapsed time is 15.355820 seconds.
>>
```

- As also indicated by the performance numbers, a local copy is only made if changes are made to the variable. For structures and cell arrays a copying scheme is used where only modified entries are copied to enhance performance*

*: http://www.mathworks.com/support/solutions/en/data/1-15SO4/index.html?solution=1-15SO4

Jacket Fundamentals
Warm-Up Jacket and MATLAB

- **Warming up Jacket:**
  - We need to warm up MATLAB and Jacket to reach reliable and reproducible timing results
  - The need for this is caused by instruction and data caching effects
- **Example:**

```
>> A=grandn(2000,2000); B=grandn(2000,2000);
>> gsync; tic; R=cos(A).*sin(B); geval(R); gsync; toc
Elapsed time is 0.260963 seconds.

>> gsync; tic; R=cos(A).*sin(B); geval(R); gsync; toc
Elapsed time is 0.010465 seconds.

>> gsync; tic; R=cos(A).*sin(B); geval(R); gsync; toc
Elapsed time is 0.010050 seconds.

>> gsync; tic; R=cos(A).*sin(B); geval(R); gsync; toc
Elapsed time is 0.010113 seconds.
```

- **As seen above the first run takes 261 ms, whereas the second, third, … only takes around 10 ms**
- **The safe way to warming up the CPU/GPU is to perform precisely the same computations, which are set to be timed**

Jacket Fundamentals
Lazy Computation

- **Lazy computation**
  - Jacket uses a lazy computation scheme where it collects several commands and tries to optimize this for least copying etc.

MATLAB

```
>> A = randn(10,10); % A is computed
>> B = A.^3 + 1;     % B is computed
>> B;                % No action due to ?;?
>> B(1,1)            % B(1,1) is displayed
```

Jacket

```
>> A = grandn(10,10); % A is not computed
>> B = A.^3 + 1;      % B is not computed
>> B;                 % No action due to ?;?
>> B(1,1)             % B is comp. and displ.
```

- **As seen in this example, Jacket first computes when the result is requested**
  - In the interactive world this of course could be more of a burden but in the important situation where it is part of a longer computation this is important.
  - If we removed the "B(1,1)" line in the small example above, Jacket would not compute anything at all.
  - This is extremely important when benchmarking – we might expect Jacket to behave as MATLAB – but it doesn't.
  - Jacket has the commands "gsync" and "geval" to force computation – typically used in benchmarking.

Jacket Fundamentals
Initial Benchmarking Considerations

- **Consequences of ignoring the lazy execution paradigm when benchmarking:**

INCORRECT – Ignoring lazy effect

```
>> N = 4000;
>> A = grandn(N,N,'single');
>> B = grandn(N,N,'single');
>> tic; R=A*B; toc

>> tic; R=A*B; toc
Elapsed time is 0.000657 seconds.

>> tic; R=A*B; toc
Elapsed time is 0.000439 seconds.
```

CORRECT – Respecting lazy effect

```
>> N = 4000;
>> A = grandn(N,N,'single');
>> B = grandn(N,N,'single');
>> gsync;tic; R=A*B; geval(R); gsync;toc

>> gsync; tic; R=A*B; geval(R); gsync; toc
Elapsed time is 3.162283 seconds.

>> gsync; tic; R=A*B; geval(R); gsync; toc
Elapsed time is 3.169414 seconds.
```

This is by far **the most typical error made among new users of Jacket** – just see the Jacket forums at: http://forums.accelereyes.com/forums/viewforum.php?f=7&sid=611c0f2c0d82950883e382d2ce462ed7

Jacket Fundamentals

Initial Benchmarking Considerations

- **IMPORTANT REMARK:** Due to the lazy computations paradigm, which is used in Jacket, **there is a difference between benchmarking CPU computations and benchmarking GPU computations:**

MATLAB

```
%% CPU COMPUTATIONS
% Generate data
Ac = randn(1000,1000,'single');


% Warm-up MATLAB
Rc = Ac.^2;
Rc = Ac.^2;

% Start timer
tStart = tic;

% Perform computation
R = Ac.^2;


% Measure elapsed time
Tcpu = toc(tStart);
```

Jacket

```
%% GPU COMPUTATIONS:
% Generate data and move to the GPU
A = randn(1000,1000,'single');
Ag = gsingle(A);

% Warm-up GPU
R = Ag.^2; geval(R);
R = Ag.^2; geval(R);

% Synchronize and start timer
gsync; tStart=tic;

% Perform computation and force evaluation
Rg = Ag.^2;
geval(R);

% Synchronize and measure elapsed time
gsync; Tgpu= toc(tStart);
```

Jacket Fundamentals
Initial Benchmarking Considerations

- **THE GPU BENCHMARKING:**
  - **The data must be transferred to the GPU memory before doing any GPU computations on it** – during the benchmarking the transfer time is not measured, since we want to compare the pure GPU computation time vs. CPU computation time.
  - Both the **CPU and, even more importantly, the GPU must be warmed up before the benchmarking**. As described earlier this is best done by performing the exact same operation as you intend to time – including repetitions if any. To save time it is possible sometimes to just get the CPU into action but this may not help is some register synchronization for example needs to be done.
  - **The command gsync must be run before the timer start and stops**. This command maintains the GPU activity (finalize the GPU pending or finalize the GPU activity depending on the current GPU state) and synchronize the CPU and the GPU. But it does NOT finalize any computations waiting due to lazy execution – it relates entirely to synchronization.
  - **The command `geval()` must be executed as the last command in the GPU computations block to force Jacket to make computations**. Without this command, Jacket tries to optimize computations – it for example tries to create as many threads as possible to keep the GPU busy. But sometimes you just need a result – most often during benchmarking. Otherwise an end-2-end code automatically yields the final result you need. In benchmarking, the timer is normally stopped after the `geval()` command.

Jacket Fundamentals

Case Study: Transfer of Data Between CPU and GPU Memory

- Suppose we need to transfer a bunch of vectors to the GPU from the MATLAB environment
- For simplicity say we have a matrix A with 1000 columns and 1000 rows – our algorithm works on rows:

$$\begin{aligned}\mathbf{a}_1 &= [a_{1,1}, \ldots, a_{N,1}]^{\mathrm{T}} \\ &\vdots \\ \mathbf{a}_N &= [a_{1,N}, \ldots, a_{N,N}]^{\mathrm{T}}\end{aligned}$$

- We could make this transfer in two ways:

$$\mathbf{a}_1, \ldots, \mathbf{a}_N$$

Transfer N vectors each of size 1000×1

$$\mathbf{A} = [\mathbf{a}_1; \cdots; \mathbf{a}_N]$$

Pack N vectors into one matrix A and transfer this size 1000×1000 matrix in one go.

Jacket Fundamentals

Case Study: Transfer of Data Between CPU and GPU Memory

- **Code for transfer between CPU and GPU memory transfer when moving one matrix**:

```
%% REFERENCE MATRICES
N = 1000;
RPT = 100;
Ac_ref = randn(N,N,'single');

%% CPU TO GPU
for ii=1:RPT
  Ag = gsingle(Ac_ref);
  geval(Ag);
end
gsync; ts=tic;
for ii=1:RPT
  Ag = gsingle(Ac_ref);
  geval(Ag);
end
gsync; tc2g = toc(ts)/RPT;
R_c2g_GBps = 4*N^2/(tc2g*1E9)
```

```
%% GPU TO CPU
Ag_ref = gsingle(Ac_ref);
geval(Ag_ref);
for ii=1:RPT
  Ac = single(Ag_ref);
end
gsync; ts=tic;
for ii=1:RPT
  Ac = single(Ag_ref);
end
gsync; tg2c = toc(ts)/RPT;
R_g2c_GBps = 4*N^2/(tg2c*1E9)
```

Jacket Fundamentals

Case Study: Transfer of Data Between CPU and GPU Memory

- **Code for transfer between CPU and GPU memory transfer when moving vectors:**

```
%% REFERENCE MATRICES
N = 1000;
RPT = 10;
Ac_ref = randn(N,N,'single');

%% CPU TO GPU
for ii=1:RPT
  Ag = gzeros(N,N,'single');
  for col=1:N,  Ag(:,col) = gsingle
(Ac_ref(:,col));  end
  geval(Ag);
end
gsync; ts=tic;
for ii=1:RPT
  Ag = gzeros(N,N,'single');
  for col=1:N,  Ag(:,col) = gsingle
(Ac_ref(:,col));  end
  geval(Ag);
end
gsync; tc2g = toc(ts)/RPT;
R_c2g_MBps = 4*N^2/(tc2g*1E6)
```

```
%% GPU TO CPU
Ag_ref = gsingle(Ac_ref); geval(Ag_ref);
for ii=1:RPT
  Ac = zeros(N,N,'single');
  for col=1:N,  Ac(:,col) = single
(Ag_ref(:,col));  end
end
gsync; ts=tic;
for ii=1:RPT
  Ac = zeros(N,N,'single');
  for col=1:N,  Ac(:,col) = single
(Ag_ref(:,col));  end
end
gsync; tg2c = toc(ts)/RPT;
R_g2c_MBps = 4*N^2/(tg2c*1E6)
```

Code includes warm-up in both directions and averages across RPT full transfers to yield reproducible results. Total transferred data in bytes is: 4*N^2*RPT, which in this case is 400 MB.

Jacket Fundamentals
Case Study: Transfer of Data Between CPU and GPU Memory

- **Measured results:**
  - Dual Intel Xeon X5570 CPU with 48 GB memory and an NVIDIA Tesla C2070; Jacket 1.7.

```
>> matrix_transfer

R_c2g_GBps =  4.2887
R_g2c_GBps =  1.6963

>>
```

```
>> vector_transfer

R_c2g_MBps =  41.9080
R_g2c_MBps =  31.8386

>>
```

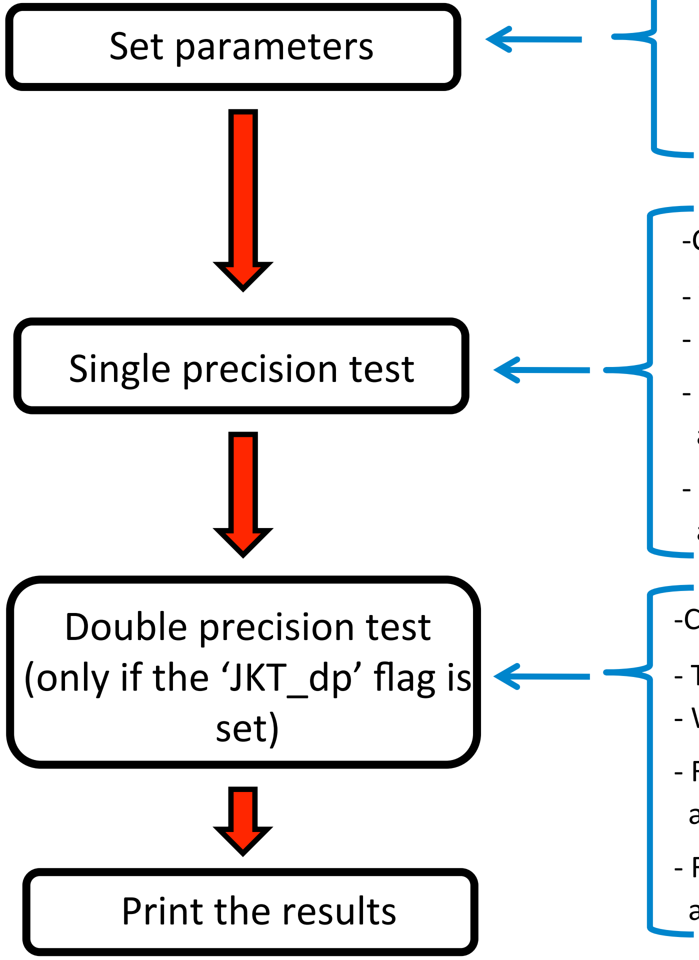
- **Observe that the results for matrix transfer is given in GB/s where the rate is given in MB/s for vector transfer**
  - For CPU>>GPU transfer the packed matrix method is more than 100 times faster than moving column by column (vector based transfer)
  - For GPU>>CPU transfer the packed matrix method is more than 50 times faster than moving column by column (vector based transfer)
- **We will investigate this issue later** … it turns out that the transfer rate is basically the same no matter the size of the data. BUT it turns out that we must pay the initialization cost once for each transfer – meaning that initialization cost is 1000 times higher for the vector based method than for the matrix based method

- So: **PACK THE DATA BEFORE TRANSFERRING IT TO/FROM THE GPU.**

Some Examples

Examples
Standardized benchmarking file

- **All examples has the following structure:**



Set parameters:

- Set size of the data
- Set the number of CPU iterations [REPSG]
- Set the number of GPU iterations [REPSC]
- Set double precision test flag [JKT_dp]

Single precision test:

-Create/Load data with a <u>single precision</u> (in a CPU memory)
- Transfer the data to the GPU memory
- Warm up the Matlab
- Run Matlab computations (REPSC times) and measure the average Matlab single precision computation time
- Run Jacket computations (REPSG times) and measure the average GPU single precision computation time

Double precision test:

-Create/Load data with a <u>double precision</u> (in a CPU memory)
- Transfer the data to the GPU memory
- Warm up the Matlab
- Run Matlab computations (REPSC times) and measure the average Matlab double precision computation time
- Run Jacket computations (REPSG times) and measure the average GPU double precision computation time

Example 1: Matrix/Matrix Multiplication

- **The initialization + CPU computations + GPU computations (single precision):**

```
%% SETUP COMPUTATION
% Matrix size
N = 2000;

% Number of repetitions
REPS = 100;

% Set to 1 if double precision is available for Jacket
JKT_dp = 1;

% Create matrices in CPU
Aref = rand(N,N,'double');
Bref = rand(N,N,'double');
```

Set number of repetition loop to ensure we measure over sufficient time to reduce influence by the operating system etc.

Define reference matrices

Examples
Example 1: Matrix/Matrix Multiplication

- **First we measure the times to do the MATLAB computations in single and double precision:**

```
%% MATLAB (CPU) SINGLE PRECISION
% Load matrices in correct format
A = single(Aref);
B = single(Bref);

% Warm-up MATLAB
R = A*B;
R = A*B;

% Time multiplication event
tStart = tic;
for jj=1:REPS
    R = A*B;
end
Tcpu_sp = toc(tStart)/REPS;
```

```
%% MATLAB (CPU) DOUBLE PRECISION
% Only perform the comp if DP is avail.
if JKT_dp==1
    % Load matrices in correct format
    A = double(Aref);
    B = double(Bref);

    % Warm-up MATLAB
    R = A*B;
    R = A*B;

    % Time multiplication event
    tStart = tic;
    for jj=1:REPS
        R = A*B;
    end
    Tcpu_dp = toc(tStart)/REPS;
end
```

- **These are trivial computations;**
  - First ensure that the correct casting is done (single/double)
  - Then warm-up MATLAB – almost (but only almost) just as important as for Jacket
  - Then do the measurement using the number of repetitions needed. **The loop in itself has such a small overhead that it doesn't matter when the run time is in the seconds range**

Examples
Example 1: Matrix/Matrix Multiplication

- **Next we move to the Jacket part, which is slightly different from MATLAB:**

```
%% JACKET (GPU) SINGLE PRECISION
% Load matrices in correct format
A = gsingle(Aref);
B = gsingle(Bref);

% Warm-up Jacket
R = A*B;   geval(R);
R = A*B;   geval(R);

% Time multiplication event
gsync;
tStart = tic;
for jj=1:REPS
    R = A*B;
    geval(R);
end
gsync;
Tgpu_sp = toc(tStart)/REPS;
```

```
%% JACKET (GPU) DOUBLE PRECISION
% Only perform the comp if DP is avail.
if JKT_dp==1
    % Load matrices in correct format
    A = gdouble(Aref);
    B = gdouble(Bref);

    % Warm-up Jacket
    R = A*B;   geval(R);
    R = A*B;   geval(R)

    % Time multiplication event
    gsync;
    tStart = tic;
    for jj=1:REPS
        R = A*B;
        geval(R);
    end
    gsync;
    Tgpu_dp = toc(tStart)/REPS;
end
```

- **Important aspects here:**
  - Use of `gsingle/gdouble` to move data to the GPU
  - Use of `geval` and `gsync` to force computations and to synchronize CPU and GPU, respectively

Examples

Example 1: Matrix/Matrix Multiplication

- **And finally print the results:**

```
%% PRINT RESULTS
fprintf('======== | BENCHMARK - Matrix multiplication: | ========\n');
fprintf('CPU run time (single prec.):     %6.3f [s]\n', Tcpu_sp);
fprintf('GPU run time (Single prec.):     %6.3f [s]\n', Tgpu_sp);
fprintf('Speed-up (single prec.):         %6.3f [-]\n', Tcpu_sp/Tgpu_sp);
if JKT_dp==1
    fprintf('CPU run time (double prec.):     %6.3f [s]\n', Tcpu_dp);
    fprintf('GPU run time (double prec.):     %6.3f [s]\n', Tgpu_dp);
    fprintf('Speed-up (Double prec.):         %6.3f [-]\n', Tcpu_dp/Tgpu_dp);
end
fprintf('==========================================================\n');
```

- **Important to note that `tic-toc` is ALWAYS used like:**

```
gsync;   tStart=tic;
...
gsync;   tStop=toc(tStart);
```

- **And ALWAYS use `geval` to force computation before stopping the timer:**

```
gsync;   tStart=tic;
R = A*B;
geval(R);
gsync;   tStop=toc(tStart);
```

Examples
Example 1: Matrix/Matrix Multiplication

- The code was tested on the machine: **Dual Intel Xeon X5570 2.93 GHz, 48 GB RAM; GPU: NVIDIA Tesla C2070, Jacket v1.7, MATLAB R2010b (build 7.11.0.584); OS: Ubuntu 10.04 LTS**

- **The result of the code on the above machine:**

```
======== | BENCHMARK - Matrix multiplication: | ========
CPU run time (single prec.):      0.189 [s]
GPU run time (Single prec.):      0.031 [s]
Speed-up (single prec.):          6.157 [-]
CPU run time (double prec.):      0.383 [s]
GPU run time (double prec.):      0.056 [s]
Speed-up (Double prec.):          6.872 [-]
=========================================================
```

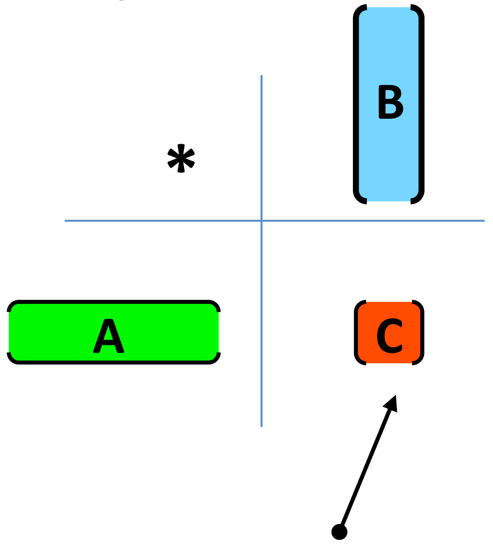
- **The single precision computations are three times faster than the CPU computations**
  – in the more complicated examples, which will be shown later, the speed gain will be even bigger
- **Quite as expected, this GPU is mainly for single precision computations**
  – in double precision it is slower than the CPU

Examples
Example 2 & 3: Vector/Vector multiplication

- Example 2 and 3 are concerned on, at the first sight, very similar problems:
  - Example 2: multiple horizontal vector and vertical vector
  - Example 3: multiple vertical vector and horizontal vector

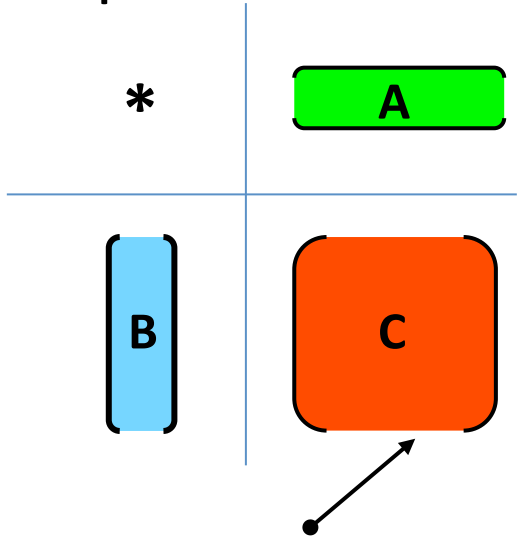
  Vector multiplying is **not commutative.** Result of multiplication from the example 2 is one number, while the result of multiplication from the example 3 is a matrix:

Example 2:



The result of the example 2 is one number – it is highly non parallizable problem.

Example 3:



The result of the example 3 is a matrix. All elements of these matrix can be computed in parallel.

Examples
Example 2 & 3: Vector/Vector multiplication

- Example 2 and 3 are very similar to the example 1, only the initial data is different – there are two vectors. Additionally, there is <u>only one</u> difference between the example 2 and the example 3 – the initialization of a data:

```
%% CREATE THE REFERENCE VECTORS
A_ref = randn(1,N,'double');
B_ref = randn(N,1,'double');
```

Vecmul2_master: Lines 28 – 31

```
%% CREATE THE REFERENCE VECTORS
A_ref = randn(N,1,'double');
B_ref = randn(1,N,'double');
```

Vecmul3_master: Lines 28 – 31

Dear careful student, where is the difference?? ;-)

- You can find the code for example 2 and example 3 in **examples/ex2** and **example/ex3** respectively.

Examples
Example 2 & 3: Vector/Vector multiplication

- Although the difference between example 2 and 3 is very tiny, the difference between benchmarking result is tremendous. The code was run on the **Intel Core i7 2.67 GHz, 12GB RAM, GPU: GeForce GTX 260 (1348 MHz), 870 MB RAM, Jacket v1.7.1 (build 58de35b), MATLAB R2010b (build 7.11.0.584), OS: Windows 7 Enterprise :**

- **Example 2:**

```
======== | BENCHMARK - vector multiplication (v1): | ========
CPU run time (single prec.):     0.137 [ms]
GPU run time (single prec.):     1.638 [ms]
Speed-up (single prec.):         0.1 [-]
CPU run time (double prec.):     0.969 [ms]
GPU run time (double prec.):     1.745 [ms]
Speed-up (double prec.):         0.6 [-]
========================================================
```

- **Example 3:**

```
======== | BENCHMARK - vector multiplication (v2): | ========
CPU run time (single prec.):     7.942 [ms]
GPU run time (single prec.):     0.297 [ms]
Speed-up (single prec.):         26.8 [-]
CPU run time (double prec.):     16.025 [ms]
GPU run time (double prec.):     0.605 [ms]
Speed-up (double prec.):         26.5 [-]
========================================================
```

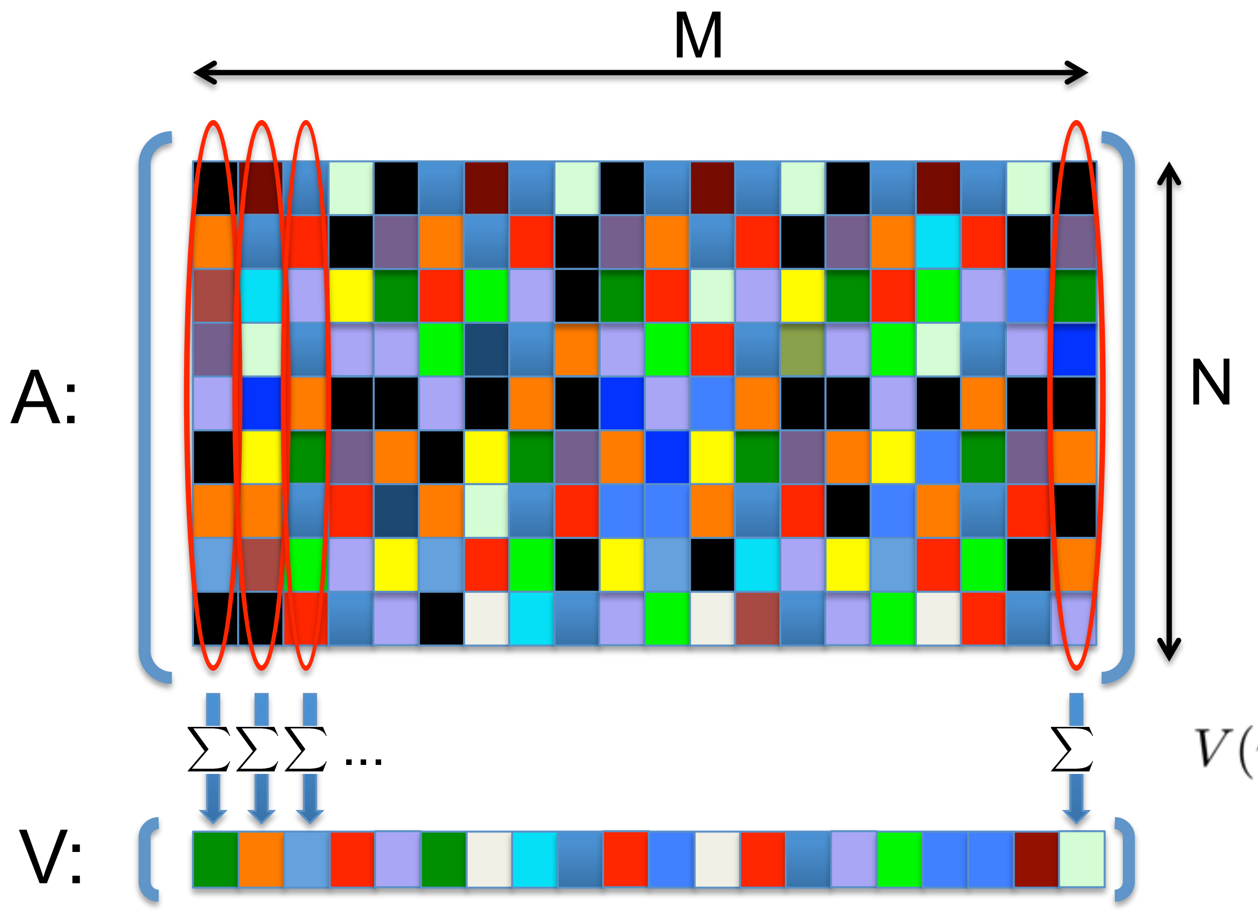
Not all types of computations are faster on a GPU.

Before porting code to JACKET try to figure out if it is possible to perform the code in parallel.

Examples

Example 4: Sum of matrix columns

- Let matrix A has N rows and M columns. The problem is to compute vector V; ith – element of this vector is a sum of ith – column of the matrix A.



This operation is highly parallelizable, sum of every column can be computed in parallel.

$$V(i) = \sum_{j=1}^{N} A(j,i)$$

Examples

Example 4: Sum of matrix columns

- In this example there are separate functions for CPU and GPU computations. There is a for loop over all columns , and internal loop which add all elements from the current column:

```
function Cdat = comp_sumcC(ac,nc,nr)

    % Preallocate vector for computation results
    Cdat = zeros(nc,1);

    % Loop over all columns of the matrix
    for inxC=1:nc

        % Loop over all elemnts in one column
        for inxR=1:nr
            Cdat(inxC) = ac(inxR,inxC) + Cdat(inxC);
        end
    end
end
```

```
function Gdat = comp_sumcG(ac,nc,nr)

    % Preallocate vector for computation results
    Gdat = gzeros(nc,1);

    % Loop over all columns of the matrix
    gfor inxC=1:nc

        % Loop over all elemnts in one column
        for inxR=1:nr
            Gdat(inxC) = ac(inxR,inxC) + Gdat(inxC);
        end
    gend
end
```

There are two differences between the CPU and the GPU code:

- Preallocation of the computation results is created using **`zeros`** function in the CPU code and **`gzeros`** in the GPU code.

- The loop over all columns of the matrix is implemented using the **`for`** loop in the CPU code and **`gfor`** loop in the GPU code.

- Obviously, the GPU function must operate on the GPU data.

Examples

Example 4: Sum of matrix columns

- You can find the code in the **examples/ex4** directory. The code was tested on the machine: **Intel Xeon 2.8 GHz, 32GB RAM, GPU: Tesla C1060 (1296 MHz), 4096 MB RAM, Jacket v1.7.1 (build 58de35b), MATLAB R2010b (build 7.11.0.584), OS: CentOS 5.5 (final)**

- The result of the code on the above machine:

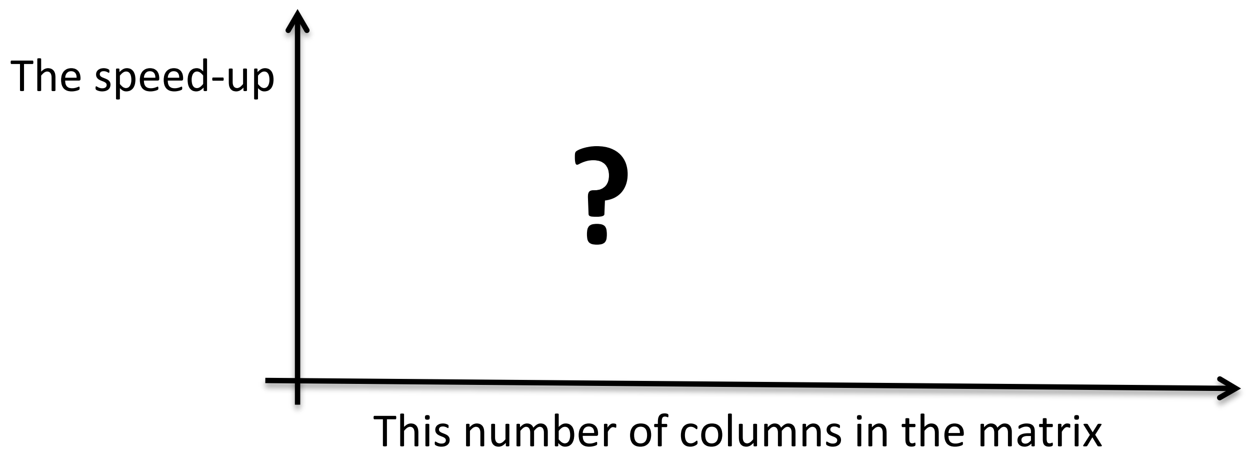
```
==== | BENCHMARK - sum of matrix columns: | ====
CPU run time (single prec.):     129.808 [ms]
GPU run time (single prec.):     10.059 [ms]
Speed-up (single prec.):           12.90 [-]

CPU run time (double prec.):     125.331 [ms]
GPU run time (double prec.):     10.706 [ms]
Speed-up (double prec.):           11.71 [-]
```

- In the above example there are 2^5 rows and 2^17 columns. What would be the speed up for smaller matrix – higher or lower.

Examples

Example 5: Sum of matrix columns – speed up vs. # of columns

- In the last example (example 4), there is a different speed up dependently on the number of columns in a processed matrix. Example 5 is concerned on measuring the speed up for the different number of columns in the matrix.



The set up for this benchmark is it:

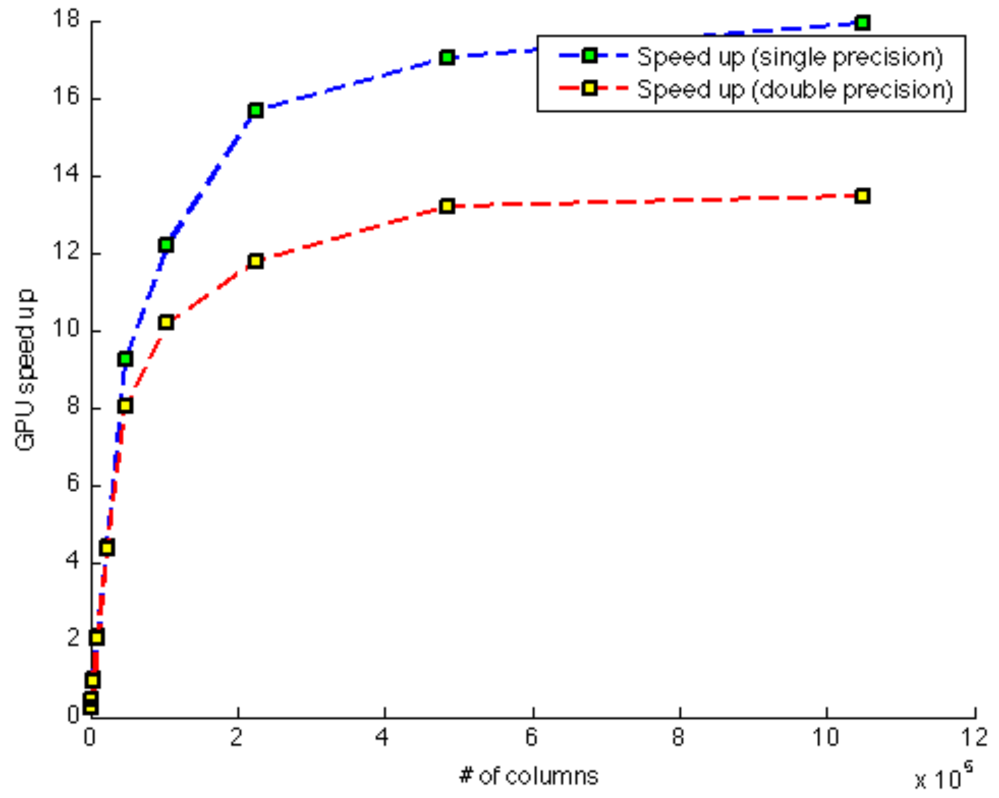
- the number of test points (sizes of matrix in the benchmark)
- the size of the matrix in the first point
- the size of the matrix in the last test point

Examples

Example 5: Sum of matrix columns – speed up vs. # of columns

- You can find the code in the **examples/ex5** directory. The code was tested on the machine: **Intel Xeon 2.8 GHz, 32GB RAM, GPU: Tesla C1060 (1296 MHz), 4096 MB RAM, Jacket v1.7.1 (build 58de35b), MATLAB R2010b (build 7.11.0.584), OS: CentOS 5.5 (final)**

  The result of the code on the above machine is shown below:

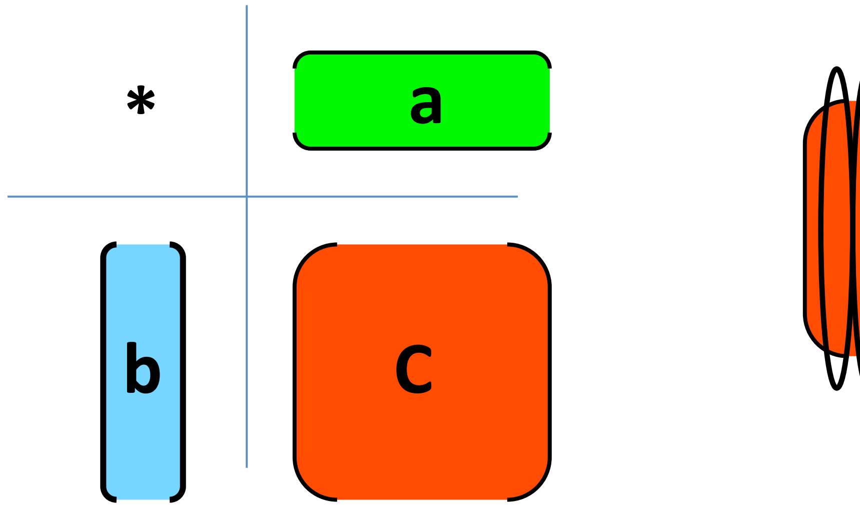


As it can be seen, the speed up increases if size of a matrix increase.

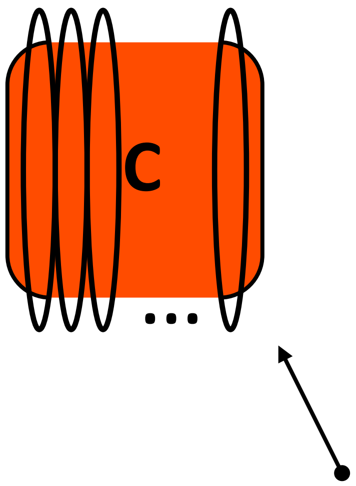
Examples

Example 6: Vector/Vector Multiplication With Normalization

- **This example measures the computational time**, which is needed to multiply two vectors (row/column) and normalize all columns of that matrix.
  - Two reference vectors (row vector a & column vector b) are created
  - Vectors are multiplied, matrix C is a product: C = a * b
  - All columns in the matrix C are normalized



1. Matrix C is computed as a product of two vectors multiplication.



2. Every column in the matrix C is treated as a vector and normalized.

Examples

Example 6: Vector/Vector Multiplication With Normalization

- **The result matrix is given by:**

$$C \leftarrow \begin{bmatrix} \dfrac{C_{1,1}}{\sqrt{\sum_{m=1}^{M} \left|C_{m,1}\right|^2}} & \cdots & \dfrac{C_{1,N}}{\sqrt{\sum_{m=1}^{M} \left|C_{m,N}\right|^2}} \\ \vdots & & \vdots \\ \dfrac{C_{N,1}}{\sqrt{\sum_{m=1}^{M} \left|C_{m,1}\right|^2}} & \cdots & \dfrac{C_{N,N}}{\sqrt{\sum_{m=1}^{M} \left|C_{m,N}\right|^2}} \end{bmatrix}$$

- **It is possible to compute this effectively by use of** “`bsxfun`” but here we will just use a loop to normalize the matrix.

Examples
Example 6: Vector/Vector Multiplication With Normalization

- **Computationally we use the following strategy:**
  - Define a CPU based function being called with a row vector, a column vector, and number of repetitions to use when doing the timing.
  - Define a GPU based function being called with a row vector, a column vector, and number of repetitions to use when doing the timing.
    - This must ensure the forced computations are done.
  - A master script to setup the data and display the results

Examples

Example 6: Vector/Vector Multiplication With Normalization

- **Core function (CPU)** – must be defined in a file name "`vvmulnormC.m`":

```
%% Vectors/vector multiplication and normalization (CPU)
function Cdat = comp_vvmulnormC(a,b)

    % Multiply vectors
    Cdat = a*b;

    % Normalize columns
    SQ = sqrt(sum(abs(Cdat).^2));
    for inxCol=1:size(Cdat,2)
        Cdat(:,inxCol) = Cdat(:,inxCol) / SQ(inxCol);
    end
end
```

- **Core function (GPU)** – must be defined in a file name "`vvmulnormG.m`":

```
% Vectors/vector multiplication and normalization (GPU)
function Gdat = comp_vvmulnormG(a,b)

    % Multiply vectors
    Gdat = a*b;

    % Normalize columns
    SQ = sqrt(sum(abs(Gdat).^2));
    gfor inxCol=1:size(Gdat,2)
        Gdat(:,inxCol) = Gdat(:,inxCol) / SQ(inxCol);
    gend
end
```

Important note:
Here we use `gfor/gend` rather than `for/end`
This has a severe impact on timing results as seen on the results slide

Examples
Example 6: Vector/Vector Multiplication With Normalization

- **Computations setup** – script file named “`vvmulnorm_master.m`”:

```
%% SETUP COMPUTATION
% Vector size
N = 4000;

% Number of repetitions
REPSG = 20;          % GPU repetitions
REPSC = 100;         % CPU repetitions

% Set to 1 to compare the CPU data with the GPU data.
SPDP_cmp = 1;

%% Check if a GPU double precision computation is available on
this GPU
gpu_info=gpu_entry(13);
if gpu_info.compute > 1.2 % double precision available
    JKT_dp = 1;
else
    JKT_dp = 0;
end

%% CREATE THE REFERENCE VECTORS
A_ref = randn(N,1,'double');
B_ref = randn(1,N,'double');
```

Examples

Example 6: Vector/Vector Multiplication With Normalization

- **Single precision computations** – (file: "`vvmulnorm_master.m`")

```
%% MATLAB (CPU) SINGLE PRECISION

% Print info...
fprintf('CPU at work (single precision)\n');

% Load vectors in the correct format (single
prec. CPU)
Ac = single(A_ref);
Bc = single(B_ref);

% Warm-up MATLAB
comp_vvmulnormC(Ac,Bc);
comp_vvmulnormC(Ac,Bc);

% Start the time measurements
tStart = tic;

% Perform CPU computations REPSC times
for jj=1:REPSC
    Cdat_sp = comp_vvmulnormC(Ac,Bc);
end

% Stop the timer
Tcpu_sp = toc(tStart);

% Calculate the average time
Tcpu_sp = Tcpu_sp/REPSC;
```

```
%% JACKET (GPU) SINGLE PRECISION

% Print info...
fprintf('GPU at work (single precision)...\n');

% Load matrix in correct format (sing. prec. GPU)
Ag = gsingle(A_ref);
Bg = gsingle(B_ref);

% Warm-up JACKET
Gdat_sp = comp_vvmulnormG(Ag,Bg); geval(Gdat_sp);
Gdat_sp = comp_vvmulnormG(Ag,Bg); geval(Gdat_sp);

% Start the timer
gsync; tStart = tic;

% Perform GPU computations REPSG times
for jj=1:REPSG
    Gdat_sp = comp_vvmulnormG(Ag,Bg);
    geval(Gdat_sp);
end

% Stop the timer
gsync; Tgpu_sp = toc(tStart);

% Calculate the average time
Tgpu_sp = Tgpu_sp/REPSG;
```

Examples

Example 6: Vector/Vector Multiplication With Normalization

- **Double precision computations** – (file: “vvmulnorm_master.m”)

```
%% MATLAB (CPU) DOUBLE PRECISION

% Print info...
fprintf('CPU at work (double precision)\n');

% Load vectors in correct format (dbl. pr.)
Ac = double(A_ref);
Bc = double(B_ref);

% Warm-up MATLAB
comp_vvmulnormC(Ac,Bc);
comp_vvmulnormC(Ac,Bc);

% Start the timer
tStart = tic;

% Perform CPU computations REPSC times
for jj=1:REPSC
    Cdat_dp = comp_vvmulnormC(Ac,Bc);
end

% Stop the timer
Tcpu_dp = toc(tStart);

% Calculate the average time
Tcpu_dp = Tcpu_dp/REPSC;
```

```
%% JACKET (GPU) DOUBLE PRECISION

% Print info...
fprintf('GPU at work (double precision)\n');

% Load matrix in correct format (dbl. pr.)
Ag = gdouble(A_ref);
Bg = gdouble(B_ref);

% Warm-up JACKET
Gdat_dp = comp_vvmulnormG(Ag,Bg); geval
(Gdat_dp);
Gdat_dp = comp_vvmulnormG(Ag,Bg); geval
(Gdat_dp);

% Start the timer
gsync; tStart = tic;

% Perform GPU computations REPSG times
for jj=1:REPSG
    Gdat_dp = comp_vvmulnormG(Ag,Bg); geval
(Gdat_dp);
end

% Stop the timer
gsync; Tgpu_dp = toc(tStart);

% Calculate the average time
Tgpu_dp = Tgpu_dp/REPSG;
```

Examples

Example 6: Vector/Vector Multiplication With Normalization

- **Tests performed on an Apple MacBook Pro with the following characteristics:**
  - Core i7-620 CPU with 8 GB memory
  - NVIDIA GeForce GT330 GPU

- Now using the above core functions we can measure the time for CPU and GPU computations

USING FOR/END FOR JACKET

```
%% VECTOR SIZE IS N=1000
>> vvmulnorm_master
Speed-up sp:  0.03
```

USING GFOR/GEND FOR JACKET

```
%% VECTOR SIZE IS N=1000
>> vvmulnorm_master
Speed-up sp:  1.55
```

- This use of `gfor/gend` has just been included as a teaser
- `GFOR/GEND` **handles the internal of the construct in parallel when possible** –
  - in this case it is obviously a huge advantage
- The use of `GFOR/GEND` **provides a speed-up in excess of 50 when comparing with the standard** `FOR/END` construct

Abbreviations

- **Abbreviations:**
  - **Cache**: Memory close to the CPU (usually on the CPU die) – often comes in several levels where level 1 is a small amount of very fast memory, level 3 is usually much larger (MB) but somewhat slower to access and level 2 is something between level 1 and 3.
  - **CUDA**: Compute Unified Device Architecture
  - **ALU**: Arithmetic Logical Unit
  - **MATLAB**: MATrix LABoratory
  - **GPU**: Graphics Processing Unit
  - **GPGPU**: General Purpose Graphics Processing Unit
  - **HPC**: High Performance Computing (used to describe problems which either take too long time to compute on large desktop computers or problems which simply can't fit on such computers)
  - **IB**: InfiniBand (high speed / low latency bus to connect computers in a cluster – most often to share memory data)
  - **LAN**: Local Area Network
  - **FLOPS**: FLOating Point operations per Second (usually measured by matrix multiplication / LINPACK or something else which is arithmetically dense)

Further Reading

1) AccelerEyes: Torben’s Corner. http://wiki.accelereyes.com/wiki/index.php/Torben’s_Corner
2) Torben Larsen, Gallagher Pryor, and James Malcolm: “Jacket: GPU Powered MATLAB Acceleration”, In “GPU Computing Gems”, Morgan-Kauffman, July 2011.
3) NVIDIA: NVIDIA CUDA C Programming Guide, 2010.
4) Jason Sanders and Edward Kandrot: “CUDA By Example – An Introduction to General-Purpose GPU Programming”, Addison-Wesley, (Boston, MA, USA), 2011.