Parallel High Performance Computing

With Emphasis on Jacket Based GPU Computing

Basic Programming

Torben Larsen
Aalborg University

Outline

Programming Methodology

Programming Methodology
Core Principles

- **The big question:**

? ?

MATLAB Programming = Jacket programming

? ?

- **The answer is very short**: **NO !**
  - The CPU has few cores **><** The GPU has many
  - The CPU has tons of memory **><** The GPU hasn't
  - The CPU memory access is fairly slow **><** The GPU generally has very fast memory access
  - The single/double performance ratio is sometimes better for CPUs than for GPUs
  - The GPU is attached via a latency affected and relatively slow bus
  - …

Algorithms and coding should match the architecture available

- **A very typical procedure for Jacket coding is the following:**
  - Ah … GPUs are fast and very trendy. Let's give them a try
  - We have some existing MATLAB code let's just do:

```
gsingle( … );
gdouble( … );
grandn( … );
```

- In rare cases it may even run correctly … and if we are very, very lucky we may even get a performance improvement

- **The problem is that the algorithm and code is not really developed for the hardware we have**

We need a different procedure to gain the advantage Jacket has to bring

Programming Methodology
Core Principles

- **Overall:**
  - Identify the hardware you expect to use for the code
  - Describe computational task – identify likely bottlenecks; describe variables incl. size and type;
  - Formulate tasks as modules – see how stop-resume procedure can be applied
  - Make initial code with one single focus: to make the code provide the correct results. Use for loops and all the stuff we know is slow (while / if / …)
  - Jacket enable the code and optimize for performance. This is a multi-step procedure.
    - **Keep copies of what you try and what results you achieve. Ensure reproducibility**

- **Recommended practice:**
  - **Functions should have a well defined objective** – use sub-functions within the same overall function if the sub-functions are only used locally.
  - **For the computational heavy part only use standard types** (not structures etc. which tends to slow things down and be more difficult to read).
  - **Use your time where it makes a difference**. Don't use 90% of your time to optimize a function using only 5% of the execution time.
  - **Use the tools, which fit the job**. (Too) many use advanced GPU programming to solve non-time critical problems taking for example 1-2 minutes to compute. It is usually not worth the trouble. **Estimate the total time based on all runs of different variables etc**

- **When considering how much time to use on code/algorithm optimization consider for example:**
  - Is the code only made for yourself? If so try to outline how many executions you will need once the code is ready – typically you need to run the code across many variables.

    Make a lifetime assessment of the code – how many times in total does it need to run

  - Is the code for a toolbox, which is internationally circulated? In this case you need to provide solid, fast and well tested and documented code.

- **Based on the problem you should have an idea of worst case array sizes etc.**, use this to estimate how much time the most critical parts take.

Programming Methodology
Core Principles

Hardware platform
GPU(s); Performance: memory, transfer, disk, CPU, functional, floating point; Architecture; …

Computational Problem
Identify functional blocks; Identify time critical parts; Worst and typical case array sizes; Make some preliminary timing tests of critical functions; …

Algorithm
Describe the algorithm to solve the problem for the platform available; Define functions and sub-functions; Define input and output variables; Describe functional behavior; Define validation cases including expected outputs; …

Initial Code
Implement version 1 of code, where focus is to get correct results; Test all functions wrt. functionality and execution time; …

Optimize Code
Step 1: All functions should allow Jacket as well as MATLAB variables; Step 2: Optimize for low execution time given the hardware constraints; Test across intended platforms; Test for correct functionality; …

Maintaining CPU and GPU Code in 1 File

Maintaining CPU and GPU Code in 1 File
Objective and Method

- Traditionally, functions are made in two versions:
  1) a MATLAB version; and **2) a Jacket version**
- This is obviously not good and it is an excellent possibility for making errors
- The objective of this part is to see what can be done to just have one file, which automatically adapts to the type of input. This objective can be met – to some degree – and we will se how and what limitations we have.
- **Two core concepts:**

CLASS – returns the class of an object

```
>> Agpu = grand(3,1,'single');
>> Acpu = rand(3,1,'single');
>> Rgpu = randn(10,1,class(Agpu));
>> Rcpu = rand(4,1,class(Acpu));
>> whos
 Name      Size     Bytes   Class

  Acpu      3x1      12      single
  Agpu      3x1      536     gsingle
  Rcpu      4x1      16      single
  Rgpu      10x1     536     gsingle
```

ISA – is object of a given class

```
>> Agpu = grand(3,1,'single');
>> Acpu = rand(3,1,'single');

>> isa(Agpu, 'gsingle')
ans=1
>> isa(Agpu, 'garray')
ans=1
>> isa(Agpu, 'single')
ans=0
>> isa(Acpu, 'gsingle')
ans=0
```

Maintaining CPU and GPU Code in 1 File
Objective and Method

- **SUPERIORCLASS** can be used to determine the class **(to know if we have Jacket or MATLAB arrays)** by using it like:

```
>> a=randn(3,1); b=grandn(3,1,'single');

>> cls = superiorfloat(a,b)

cls =

   Empty array: 0-by-0

>> whos a b cls
  Name          Size                Bytes  Class      Attributes

  a             3x1                    24     double
  b             3x1                   536    gsingle
  cls           0x0                   536    gsingle

>>
```

Case Study: Class Dependent Computations

- Let's take a **small example** to show how this can be used in practice.
- Say we need a function to perform the following input-output relation:

$$r_i = a_i \exp\left[\sin\{k\, b_i\, d_i\}\right] + c$$

- where:

$$\begin{aligned}
\mathbf{r} &= [r_1, \ldots, r_N]^{\mathrm{T}} \in \mathcal{C}^{N\times 1} && \text{(Response)} && (1)\\
\mathbf{a} &= [a_1, \ldots, a_N]^{\mathrm{T}} \in \mathcal{C}^{N\times 1} && \text{(Input)} && (2)\\
\mathbf{b} &= [b_1, \ldots, b_N]^{\mathrm{T}} \in \mathcal{C}^{N\times 1} && \text{(Random)} && (3)\\
\mathbf{d} &= [0, 1, \ldots, N-1]^{\mathrm{T}} \in \mathcal{R}^{N\times 1} && && (4)
\end{aligned}$$

- **In general particularly the d-vector is difficult to handle in functions, which should be both MATLAB and Jacket enabled. It is possible to handle it like:**

```
cls = class(a);
zero = zeros(cls);
d = (zero : N-1).';
```

By using this approach we ensure that the $\mathbf{d}$-vector follows the class of the input vector $\mathbf{a}$. **This improves speed quite a lot**. This type of colon is very important and is investigated later.

Maintaining CPU and GPU Code in 1 File
Case Study: Class Dependent Computations

- `foo.m` **function file, which is both MATLAB and Jacket enabled**

```
function [ r ] = foo( a, k, c )
  % foo Computes the output r for both MATLAB and Jacket variables.

  % Determine the common class
  cls = class(a);

  % Define scalar '0' used in colon expansion
  zero = zeros(cls);

  % Create random vector b; use grandn if a is a garray type,
  % and randn otherwise
  b = randn(length(a),1,cls);

  % Create d vector; will type wise follow the a and b vectors
  d = (zero:length(a)-1).';

  % Compute the output; no need to multiply c with
  % ones(N,1) as a scalar here is automatically handled
  r = a .* exp(sin( k * b .* d )) + c;
end
```

Maintaining CPU and GPU Code in 1 File
Case Study: Class Dependent Computations

- master_foo.m **script file which controls the test of the foo.m function:**

```
% Script file to test the foo.m function. The objective is to show how one
% function file by use of class inheritance can be used for both MATLAB and
% Jacket. A computational function is used to illustrate the concepts.

%% SINGLE
a=randn(1E7,1,'single'); k=single(0.1); c=single(2);
t1=tic;
for ii=1:5
  x_single_matlab=foo(a,k,c);
end;
t_single_matlab=toc(t1)/5

a=grandn(1E7,1,'single'); k=gsingle(0.1); c=gsingle(2); geval(a,k,c);
gsync;
t1=tic;
for ii=1:20
  x_single_jacket=foo(a,k,c);
  geval(x_single_jacket);
end;
gsync;
t_single_jacket=toc(t1)/20
Speedup_single = t_single_matlab / t_single_jacket
```

Maintaining CPU and GPU Code in 1 File

Case Study: Class Dependent Computations

```
%% DOUBLE
gver = gpu_entry(13);
if gver.compute > 1.2
  a=randn(1E7,1,'double'); k=double(0.1); c=double(2);
  t1=tic;
  for ii=1:5,   x_double_matlab=foo(a,k,c);   end;
  t_double_matlab=toc(t1)/5

  a=grandn(1E7,1,'double'); k=gdouble(0.1); c=gdouble(2);
  geval(a,k,c);
  gsync;  t1=tic;
  for ii=1:20,   x_double_jacket=foo(a,k,c);
                 geval(x_double_jacket);  end;
  gsync;  t_double_jacket=toc(t1)/20
  Speedup_double = t_double_matlab / t_double_jacket
end

%% TYPES
if gver.compute > 1.2
   whos x_single_matlab x_single_jacket ...
        x_double_matlab x_double_jacket
else
  whos x_single_matlab x_single_jacket
end
```

The test is also performed if the GPU is double precision enabled.

Show the types of the variables. The byte count is incorrect for Jacket variables. No solution to this at the moment.

Maintaining CPU and GPU Code in 1 File

Case Study: Class Dependent Computations

- **Results:**
  - **Core i7-970 with 24 GB memory**, and **NVIDIA Quadro 4000**

```
>> master_foo

t_single_matlab    =  0.2471
t_single_jacket    =  0.0107
Speedup_single     = 23.1687

t_double_matlab    =  0.2732
t_double_jacket    =  0.0096
Speedup_double     = 28.5032

  Name                         Size             Bytes            Class

  x_double_jacket      10000000x1               824            gsingle
  x_double_matlab      10000000x1          80000000            double
  x_single_jacket      10000000x1               824            gsingle
  x_single_matlab      10000000x1          40000000            single
```

Maintaining CPU and GPU Code in 1 File
FOR and GFOR

- **With GFOR things are more complicated – no obvious solution exist as part of Jacket**
- **The ISA function may be used to select between methods: - requires Jacket installed:**

```
function [ A ] = computefun( A, b )
  N = size(A,2);

  if isa(A,'garray')
    gfor ii=1:N                 % Jacket
      A(:,ii) = A(:,ii) .* b;
    gend
  else
    for ii=1:N                  % MATLAB
      A(:,ii) = A(:,ii) .* b;
    end
  end
end
```

Master file

```
N = 10E3;
K = 2E3;

Ac = randn(N,K,'single');
Ag = gsingle(Ac);
bc = randn(N,1,'single');
bg = gsingle(bc);

Rc = computefun( Ac, bc );
Rg = computefun( Ag, bg );

whos Rc Rg
```

```
>> loop
  Name          Size                Bytes       Class       Attributes

  Rc        10000x2000            80000000       single
  Rg        10000x2000              536          gsingle
```

Maintaining CPU and GPU Code in 1 File
FOR and GFOR

- **Possible to use try-catch also – use when Jacket is not installed:**

```
function [ y ] = examplefun( x )
  %% TEMPORARY VARIABLES
  N = length(x);
  try                        % Try with Jacket
    y = gzeros(N,N);
    tmp = grandn(N,N);
    gfor ii=1:N
      y(:,ii) = x(ii)*tmp(:,ii);
    gend
    fprintf('\nJacket computing ...\n');
  catch                      % and use MATLAB else
    y = zeros(N,N);
    tmp = randn(N,N);
    for ii=1:N
      y(:,ii) = x(ii)*tmp(:,ii);
    end
    fprintf('\nMATLAB computing ...\n');
  end
end
```

```
>> xc=examplefun(ones(3,1))

>> xg=examplefun(gones(3,1));


>> whos xc xg
```

But it is better to use the class dependence directly and not rely on MATLABs error handling mechanism

Efficient Array Indexing

Efficient Array Indexing
Analysis

- **Colon indexing in MATLAB is extremely powerful** – i.e. A(1:N,:) to access only parts of a matrix A.
- However, a recent discovery showed that in particular **Jacket is very sensitive to how we access the arrays** - http://wiki.accelereyes.com/wiki/index.php/Array_Indexing_In_Jacket.

- This study compares six different ways of array indexing:
  - **FOO1.m:** Indexing is done like **a(1:length(a)-1)**.
  - **FOO2.m:** Indexing done like **a(p1)** with **p1=one:N-1** and "one" is a class dependent scalar "1" – meaning one is a single for MATLAB and a **gsingle** for Jacket.
  - **FOO3.m:** Indexing done like **a(1:end-1)** with no class derived indexing.
  - **FOO4.m:** Similar to FOO2.m … but indexing done like **a(p1)** with **p1=one:one*(N-1)** and "one" is a class dependent scalar "1" – meaning one is a single for MATLAB and a **gsingle** for Jacket.
  - **FOO5.m:** Similar to FOO4.m … but where we refer to **a(p1)** by using a predefined **X=length(a)-n;** and then **p1=1:end-X.**
  - **FOO6.m:** Similar to FOO5.m … but here we use class dependent values for **X** and **p1**.

- **In the following we will look into the characteristics of these different indexing methods**

Efficient Array Indexing
Analysis

- FOO1.m:

```
function [ y ] = foo1( a, b )
  % foo1 Computes the output r for both MATLAB and
  % Jacket variables.

  % N is the length of the input vectors a and b; no
  % sanity check for ease
  N = length(a);

  % Determine the common class
  cls = superiorfloat(a,b);

  % Create d vector; type wise follow the
  % a and b vectors
  d = randn(N,1,cls);

  % Compute the output
  r = b(1:N-1) .* exp(sin( d(1:N-1) ./ d(2:N) ...
      .* a(1:N-1) ./ a(2:N) )) + b(2:N);
  y = sum(r(:));
end
```

Class taken from input arrays a and b.

Define random array by class dependence; uses **RANDN** for MATLAB and **GRANDN** for Jacket class

Output computed by colon operations

Efficient Array Indexing
Analysis

- FOO2.m:

```
function [ y ] = foo2( a, b )
  % foo2 Computes the output r for both MATLAB and Jacket
  % variables.

  % N is the length of the input vectors a and b; no
  % sanity check for ease
  N = length(a);

  % Determine the common class and class dependent '1'
  cls = superiorfloat(a,b);
  one = ones(cls);

  % Create d vector; type wise follow the a and b vectors
  d = randn(N,1,cls);

  % Indexing pointers following the class of the input vectors
  p1 = one:(N-1)*one;
  p2 = 2*one:N*one;

  % Compute the output;
  r = b(p1) .* exp(sin( d(p1)./d(p2) .* a(p1)./a(p2) )) ...
      + b(p2);
  y = sum(r(:));
end
```

Class taken from input arrays a and b. one defines "1" with class determined from a and b

Define random array by class dependence; uses **RANDN** for MATLAB and **GRANDN** for Jacket class.

Class inherited index pointers p1 and p2.

Output computed by colon operations.

Analysis

- FOO3.m:

```
function [ y ] = foo3( a, b )
  % foo3 Computes the output r for both MATLAB and
  % Jacket variables.

  % Vector length
  N = length(a);

  % Determine the common class
  cls = superiorfloat(a,b);

  % Create d vector; will type wise follow the a and b
  % vectors
  d = randn(N,1,cls);

  % Compute the output
  r = b(1:end-1) .* exp(sin( d(1:end-1) ./ d(2:end) ...
      .* a(1:end-1)./ a(2:end) )) + b(2:end);
  y = sum(r(:));
end
```

Class taken from input arrays a and b.

Define random array by class dependence; Uses **RANDN** for MATLAB and **GRANDN** for Jacket class.

Output computed by colon operations but here we refer to last elements by use of "**end**".

Analysis

- F004.m:

```
function [ y ] = foo4( a, b, N )
  % foo4 Computes the output r for both
  % MATLAB and Jacket variables.

  % Determine the common class
  cls = superiorfloat(a,b);

  % Define class dependent scalar '1' used in colon
  % expansion
  one = ones(cls);

  % Create d vector; will type wise follow the a and b
  % vectors
  d = randn(length(a),1,cls);

  % Compute the output
  p1 = one : N-1;
  p2 = 2*one : N;

  r = b(p1) .* exp(sin( d(p1)./d(p2) .* a(p1)./a(p2) )) ...
      + b(p2);
  y = sum(r(:));
end
```

Class taken from input arrays a and b.

one defines "1" with class determined from **a** and **b**.

Define random array by class dependence; Uses **RANDN** for MATLAB and **GRANDN** for Jacket class.

Class inherited index pointers p1 and p2.

Output computed by the index pointers.

Efficient Array Indexing
Analysis

- F005.m:

```
function [ y ] = foo5( a, b, N )
  % foo5 Computes the output r for both
  % MATLAB and Jacket variables.

  % Determine the common class
  cls = superiorfloat(a,b);

  % Create d vector; will type wise follow the a and b
  % vectors
  d = randn(length(a),1,cls);

  % Compute the output
  X = length(a) - N + 1;

  r = b(1:end-X-1) .* exp(sin(d(1:end-X-1) ./ d(2:end-X) ...
        .* a(1:end-X-1) ./ a(2:end-X))) + b(2:end-X);
  y = sum(r(:));
end
```

Class taken from input arrays a and b.

Define random array by class dependence; uses **RANDN** for MATLAB and **GRANDN** for Jacket class.

Subtraction value when using "**end**". Could just have been written as X=1 but this is done to facilitate a more general approach.

Output determined by "**end**" combined with a subtraction value.

Efficient Array Indexing
Analysis

- FOO6.m:

```
function [ y ] = foo6( a, b, N )
  % foo6 Computes the output r for both
  % MATLAB and Jacket variables.

  % Determine the common class
  cls = superiorfloat(a,b);
  one = ones(cls);
  two = 2*one;

  % Create d vector; type wise follow
  % the a and b vectors
  d = randn(length(a),1,cls);

  % Compute the output
  X = one*(length(a) - N + 1);

  r = b(one:end-X-1) .* exp( sin(d(one:end-X-1) ...
      ./ d(two:end-X) .* a(one:end-X-1) ...
      ./ a(two:end-X) )) + b(two:end-X);
  y = sum(r(:));
end
```

Class taken from input arrays a and b.
Defines class dependent "one" and "two"

Define random array by class dependence;
Uses **RANDN** for MATLAB and **GRANDN** for Jacket class.

Subtraction value when using end. Could just have been written as **X=1** but this is done to facilitate a more general approach.

Output via "**end**" and a class dependent subtraction value.

Results

- **Results:**
  - Dual Intel Xeon X5670 with 192 GB memory; **NVIDIA C2070 GPU;** **MATLAB R2010b**; **Jacket 1.7.**
  - Validity has been checked (not directly possible with the code shown due to random inputs changing from run to run).

Function	MATLAB [s]	Jacket [s]	Speed-up	Characteristics
FOO1	0.705737	0.856081	0.824381	a(1:length(a)-1)
FOO2	0.896946	0.025219	35.565758	a(p1), p1=one:N-1 (class inherited)
FOO3	0.493997	0.007911	62.445923	a(1:end-1)
FOO4	0.885673	0.024603	35.999250	a(p1), p1=one:one*(N-1) (class inh.)
FOO5	0.682283	0.007843	86.989167	a(1:end-X), X=length(a)-N+1
FOO6	0.681455	0.034046	20.015442	A(one:end-X), X=one*(length(a)-N+1)

Conclusions:

- There is not all that much performance difference for MATLAB; but huge difference for Jacket

Efficient Array Indexing
Results

- **Results:**
  - Intel Core i7-970 CPU with 24 GB memory; **NVIDIA Quadro 4000 GPU; MATLAB R2011a; Jacket 1.7.1**

Function	MATLAB [s]	Jacket [s]	Speed-up	Characteristics
FOO1	0.625742	0.692563	0.903517	a(1:length(a)-1)
FOO2	0.835107	0.053767	15.532015	a(p1), p1=one:N-1 (class inherited)
FOO3	0.458965	0.013276	34.569788	a(1:end-1)
FOO4	0.839691	0.048237	17.407463	a(p1), p1=one:one*(N-1) (class inh.)
FOO5	0.633865	0.013354	47.466337	a(1:end-X), X=length(a)-N+1
FOO6	0.631005	0.055384	11.393289	A(one:end-X), X=one*(length(a)-N+1)

Conclusions:
- Results similar to that for the X5670 + C2070 platform
- There is not all that much performance difference for MATLAB; but huge difference for Jacket

Efficient Array Indexing
Results

- Conclusions:

\+
```
a(dX:end-dY)
```

Best for both MATLAB and Jacket

\-
```
a(dX:length(a)-dY)
```

\-
```
p1=1:length(a);
a(p1)
```

\-
```
cls = class(a);
one = ones(cls);
a(one:one*length(a))
```

Acquiring System Information

Acquiring System Information
Objective

- **When performing various tests it is very useful to acquire some basic information such as:**
  - MATLAB version
  - Jacket version
  - Operating system
  - GPUs available
  - Active GPU
  - Active CPU
  - Compute capability of the GPU
  - GPU driver
  - GPU toolkit
  - …

Handled by calls to the Jacket kernel as well as MATLAB and operating system

Acquiring System Information
Implementation

- **Function to extract system information (Windows):**

```
function [ sys ] = sysinfo()
  %SYSINFO Returns information about the system, MATLAB, Jacket and GPU(s)

  % Operating system
  cver = evalc('ver');
  pn = regexp(cver,'\n');
  pb = strfind(cver,'Operating System:');
  ptr = find(pn>pb(1));
  sys.os = cver(pb(1)+18:pn(ptr(1))-1);

  % CPU info - windows only
  val = winqueryreg('HKEY_LOCAL_MACHINE', ...
                    'HARDWARE\DESCRIPTION\System\CentralProcessor\0', ...
                    'ProcessorNameString');
  sys.cpu = val;

  % MATLAB version
  cver = evalc('ver');
  pb = strfind(cver,'MATLAB Version');
  pe = strfind(cver,')');
  sys.matlab = cver(pb+15:pe(1));
```

Acquiring System Information

Implementation

```
% Jacket version
  gver = gpu_entry(13);
  sys.jacket = gver.version;

  % GPUs available
  sys.gpu0 = gpufinder('GPU0');
  if ~isempty(gpufinder('GPU1')), sys.gpu1 = gpufinder('GPU1'); end
  if ~isempty(gpufinder('GPU2')), sys.gpu2 = gpufinder('GPU2'); end
  if ~isempty(gpufinder('GPU3')), sys.gpu3 = gpufinder('GPU3'); end
  if ~isempty(gpufinder('GPU4')), sys.gpu4 = gpufinder('GPU4'); end
  if ~isempty(gpufinder('GPU5')), sys.gpu5 = gpufinder('GPU5'); end
  if ~isempty(gpufinder('GPU6')), sys.gpu6 = gpufinder('GPU6'); end
  if ~isempty(gpufinder('GPU7')), sys.gpu7 = gpufinder('GPU7'); end
  if ~isempty(gpufinder('GPU8')), sys.gpu8 = gpufinder('GPU8'); end

  % Active GPU
  gver = evalc('ginfo');
  inuse = regexp(gver,'(in use)');
  gpufields = regexp(gver,'GPU');
  gpus = find(gpufields<inuse);
  sys.active_gpu = gver(gpufields(gpus(end)):inuse-3);
```

Implementation

```
  % Display GPU
  gver = evalc('ginfo');
  dispdev = regexp(gver,'Display Device: ');
  cr = regexp(gver,'\n');
  ptr = find(cr>dispdev);
  sys.display_gpu =  gver(dispdev+16:cr(ptr(1))-1);

  % GPU driver
  gver = evalc('ginfo');
  pb = strfind(gver,'CUDA driver');
  pn = regexp(gver,',');
  ptr = find(pn>pb(1));
  sys.driver = gver(pb(1)+12:pn(ptr(1))-1);

  % CUDA compute level
  gver = gpu_entry(13);
  sys.cudalv = gver.compute;

  % CUDA toolkit version
  gver = gpu_entry(13);
  sys.cudatk = gver.toolkit;
end
```

Implementation

- **Local function gpufinder:**

```
function [ gtype ] = gpufinder( gpu )
  gver = evalc('ginfo');
  pb = strfind(gver,gpu);
  if isempty(pb), gtype=pb;
    return;
  end
  pn = regexp(gver,',');
  ptr = find(pn>pb(1));
  gtype = gver(pb(1)+5:pn(ptr(1))-1);
end
```

Acquiring System Information
Example

- **Example of output:**

```
>> simsystem = sysinfo()

          os: 'Microsoft Windows 7 Version 6.1 (Build 7600)'
         cpu: 'Intel(R) Core(TM) i7 CPU         970  @ 3.20GHz'
      matlab: '7.12.0.635 (R2011a)'
      jacket: '1.7.1 (build 58de35b)'
        gpu0: 'Quadro 2000'
        gpu1: 'Quadro 4000'
        gpu2: 'Quadro 4000'
        gpu3: 'Quadro 4000'
  active_gpu: 'GPU0 Quadro 2000, 1251 MHz, 994 MB VRAM, ...
               Compute 2.1 (single,double)'
 display_gpu: 'GPU3 Quadro 4000'
      driver: '263.06'
      cudalv: 2.1000
      cudatk: 3.2000
```

- **Save this information together with benchmarks.**

Acquiring System Information
Example

- Another example from **ged0.lab.es.aau.dk**:

```
>> simsystem = sysinfo_unix()

          os: 'Linux 2.6.18-238.9.1.el5 #1 SMP ...
               Tue Apr 12 18:10:13 EDT 2011 x86_64'
      matlab: '7.11.0.584 (R2010b)'
      jacket: '1.7.1 (build 58de35b)'
        gpu0: 'Tesla C1060'
        gpu1: 'Tesla C1060'
        gpu2: 'Tesla C1060'
        gpu3: 'Quadro FX 5800'
  active_gpu: 'GPU0 Tesla C1060, 1296 MHz, 4096 MB VRAM, ...
               Compute 1.3 (single,double)'
 display_gpu: 'GPU3 Quadro FX 5800'
      driver: '260.19.26'
      cudalv: 1.3000
      cudatk: 3.2000

>>
```

Benchmarking MATLAB and Jacket Functions

Benchmarking MATLAB and Jacket Functions
Introduction

- **So, why do we want to perform benchmarking?**

Compare different algorithms

…

Establish baseline for the performance of our code

Ensure that new versions of toolboxes perform at least as good as the previous one

Compare different implementations

Know performance of different functions on CPUs and GPUs

Compare CPU and GPU implementations

- **Dependencies:**

Process priority

Software platforms

Hardware platform

Power scheme

Problem size

…

Introduction

- **When performing benchmarking it is important to document the platform used – both in terms of hardware and software**
- **Hardware description includes:**
  - Model name (e.g. Colfax CXT2000).
  - Chassis (e.g. Antec P183) – might not seem relevant but since excessive heating may cause problems the chassis is also important.
  - Motherboard (e.g. ASUS P6T7 Supercomputer motherboard).
  - CPU (e.g. Intel Core i7-975 Extreme 3.33 GHz).
  - CPU memory (e.g. 6 x 4 GB DDR3 RAM, 1333 MHz).
  - GPU(s) (e.g. PNY NVIDIA Quadro 4000 and NVIDIA Tesla C2050).
  - Disk(s) (e.g. Intel X25-M 160 GB).
  - …
- **Software description includes:**
  - MATLAB version (Windows/Mac/Linux, 32/64 bits).
  - Jacket version and packages.
  - Operating system and kernel.
  - CUDA driver.
  - CUDA Toolkit.
  - …

Sometimes it is difficult to get all desired information but do as much as you can to comply with the principle of "Reproducible Research" [1]

Benchmarking MATLAB and Jacket Functions
Introduction

- **Benchmarking requirements:**
  - The results must be correct and reproducible.

- **How is that achieved?**
  - The hardware must be precisely described.
  - The software versions, drivers, toolkits etc. must be precisely documented.
  - Benchmarking code must be publically available and open to scientific scrutiny.
  - A fresh MATLAB must always be opened before a benchmark is performed.
  - Always measure for so long time durations that small artifacts from the operating system is insignificant.

Benchmarking MATLAB and Jacket Functions
Procedure

- **Principle:**
  - **Warm-up** – the best is to perform the exact same procedure as we later want to measure.
  - Estimate the number of repetitions we need to measure over a certain minimum time TMIN.
  - **While measurement time < TMIN**
    - Perform measurement.
    - Increase the number of repetitions.
  - **Measure the loop repetition time to compensate for this later – the time is short but we can still compensate for it.**

```
R = matrix of same size as result matrix
geval(R);

synchronize
Start timer >> t1=tic;
for ii=1:RPTloop
  for jj=1:RPT
    geval(R);
  end
end
synchronize
Stop timer >> looptime=toc(t1)/RPTloop;
```

Measurement of loop repetition time. This is done by repeating the loop repetition as this is very fast in itself. Be aware that RPTloop·RPT don't get too big.

Procedure

- **Ensuring minimum execution times**
  - Effectively combined with warming up when using two executions of the benchmark code
  - The second warming up execution is timed and used to estimate the number of repetitions needed to reach a certain minimum required measurement time
  - A loop ensures that the minimum required measurement time is actually met
- **Principle:**

```
Tmin = 2;
tend = -1;   count = 0;
while tend<Tmin
  count = count + 1;
  << Warming up >>
  t1 = tic;
  << Warming up >>
  t2 = toc(t1);
  RPT = round(1.4*count*Tmin/t2);
  gsync; ts = tic;
  for rpt=1:RPT
      Res = BENCHMARKFUN(in1,in2);
      geval(Res);
  end
  gsync; tend = toc(ts);
end
```

Having the warm-up procedure inside the while-end loop may be overdoing it. The while-end loop ensures that we meet the minimum measurement time and we update the repetition number via the count variable.

Benchmarking MATLAB and Jacket Functions
MATLAB Implementation

- **MATLAB timing:**

```
function [ TC, TELAPSC, RPTC ] = timefun( FUN, TMIN )
  % TIMEFUN Timing of MATLAB function

  % Generate error if FUN is not a function handle
  assert(isa(FUN, 'function_handle'))

  % Default time to average over
  if nargin < 2,   TMIN = 0.25;   end

  % Estimate time and number of repetitions
  FUN();   FUN();
  ts = tic;   FUN();   FUN();   tend = toc(ts)/2;
  RPTi = max(ceil(0.75*TMIN/tend),2);

  % Warm up and adjust RPTi estimate
  ts = tic;
  for rpt=1:RPTi,
    FUN();
  end;
  tend = toc(ts)/RPTi;
  RPTi = max(ceil(TMIN/tend),2);
```

Set default measurement time.

Initial warm-up and estimate number of repetitions to obey minimum measurement time.

Full warm-up and more accurate estimation of the number of repetitions to obey minimum measurement time.

Benchmarking MATLAB and Jacket Functions
MATLAB Implementation

```
  % Measure time
  TELAPSC = -1;
  while TELAPSC < TMIN
    ts = tic;
    for rpt=1:RPTi,
      FUN();
    end;
    TELAPSC = toc(ts);
    RPTC = RPTi;
    RPTi = ceil(1.25*RPTi);
  end

  % Compensate for loop time
  y = FUN();
  ts = tic;
  for ii=1:ceil(1E6/RPTC)
    for rpt=1:RPTC
      y;
    end
  end
  tloop = toc(ts)/ceil(1E6/RPTC);
  TC = (TELAPSC-tloop)/RPTC;
end
```

Perform the actual measurement.

Loop time compensation. Here it is done such that the product of the number of measurement repetitions and the number of loop time measurement repetitions are around 1E6. This can be modified if requested.

Jacket Implementation

- **Jacket timing:**

```
function [ TG, TELAPSG, RPTG ] = gtimefun( FUN, TMIN )
  % GTIMEFUN Timing of Jacket function

  % Generate error if FUN is not a function handle
  assert(isa(FUN, 'function_handle'))

  % Default time to average over
  if nargin < 2,  TMIN = 0.25;  end

  % Estimate time and number of repetitions
  geval(FUN());  geval(FUN());
  gsync; ts = tic;
    geval(FUN());  geval(FUN());
  gsync; tend = toc(ts)/2;
  RPTi = max(ceil(0.75*TMIN/tend),2);

  % Warm up and adjust RPTi estimate
  gsync, ts = tic;
  for rpt=1:RPTi,  geval(FUN());  end
  gsync; tend = toc(ts)/RPTi;
  RPTi = max(ceil(TMIN/tend),2);
```

Set default measurement time.

Initial warm-up and estimate number of repetitions to obey minimum measurement time. Observe use of GEVAL.

Full warm-up and more accurate estimation of the number of repetitions to obey minimum measurement time. Again observe use of GEVAL.

Jacket Implementation

```matlab
  % Measure time
  TELAPSG = -1;
  while TELAPSG < TMIN
    gsync; ts = tic;
    for rpt=1:RPTi
      geval(FUN());
    end
    gsync; TELAPSG = toc(ts);
    RPTG = RPTi;
    RPTi = ceil(1.25*RPTi);
  end

  % Compensate for loop time
  y = FUN();
  gsync, ts = tic;
  for ii=1:ceil(1E6/RPTG)
    for rpt=1:RPTG
      geval(y);
    end
  end
  gsync; tloop = toc(ts)/ceil(1E6/RPTG);
  TG = (TELAPSG-tloop)/RPTG;
end
```

Perform the actual measurement. Evaluate the call to the function handle.

Loop time compensation. Here it is done such that the product of the number of measurement repetitions and the number of loop time measurement repetitions are around 1E6. This can be modified if requested. Inside the loop we evaluate the result – but we obviously don't compute it!

Benchmarking MATLAB and Jacket Functions
Example 1

- **Results:**
  - Dual Intel Xeon X5670 with 192 GB memory**; NVIDIA Tesla C2070;** **Jacket 1.7**; **MATLAB R2010b**

```
>> N = 10000;
>> Ac = randn(N,N,'single');
>> t_matlab = timefun(@() exp(tan(sin(cos(Ac)))), 5)

     t_matlab = 0.6221

>> t_matlab = timefun(@() exp(tan(sin(cos(Ac)))), 5)

     t_matlab = 0.6203
```

```
>> N = 10000;
>> Ag = grandn(N,N,'single');
>> t_jacket = gtimefun(@() exp(tan(sin(cos(Ag)))), 5)

     t_jacket = 0.0341

>> t_jacket = gtimefun(@() exp(tan(sin(cos(Ag)))), 5)

     t_jacket = 0.0341
```

Benchmarking MATLAB and Jacket Functions
Example 2

- **Results for measuring floating point performance by matrix multiplication:**
  - Dual Intel Xeon X5670 with 192 GB memory**; NVIDIA Tesla C2070;** **Jacket 1.7**; **MATLAB R2010b.**

```
>> N = 1000;
>> Ac = randn(N,N,'single');
>> Bc = randn(N,N,'single');
>> matlab_gflops = N^2*(2*N-1)/(1E9*timefun(@() Ac*Bc, 5))

   matlab_gflops = 204.0606

>> matlab_gflops = N^2*(2*N-1)/(1E9*timefun(@() Ac*Bc, 5))

   matlab_gflops = 114.1704
```

```
>> N = 1000;
>> Ag = grandn(N,N,'single');
>> Bg = grandn(N,N,'single');
>> jacket_gflops = N^2*(2*N-1)/(1E9*gtimefun(@() Ag*Bg, 5))

   jacket_gflops = 481.9003

>> jacket_gflops = N^2*(2*N-1)/(1E9*gtimefun(@() Ag*Bg, 5))

   jacket_gflops = 481.8823
```

Notice that we here second time clearly have been allocated less resources (cores) for the MATLAB computation than first time. This is controlled by the operating system and the CPU – the only thing we might be able to do is to set our MATLAB process to high priority (is possible).

For Jacket we don't see these differences since even one core is more than sufficient to keep the GPU occupied.

Jacket Code on Non-Jacket Enabled MATLAB Installations

Jacket Code on Non-Jacket Enabled MATLAB Installations
Objective

- **As long as we only use the Jacket enabled code we make on our own computers it is not a problem**
- **But suppose we distribute Jacket based toolbox to users without Jacket …**

if we don't do something the code crashes …

- A plain MATLAB does not know "`gsingle`, `ginfo`" etc.

Jacket library defining the special Jacket constructs such that they revoke to plain MATLAB

```
gsingle >> single
gdouble >> double
        .
        .
        .
```

Jacket Code on Non-Jacket Enabled MATLAB Installations
Implementation

- **The virtual_jacket library defines:**

gdouble	geval	geye	ginfo	gint8
gint32	glogical	gones	grand	grandn
gsingle	gsync	guint8	guint32	gzeros

- **Example of code:**

```
function [ out ] = grandn( varargin )
  out = randn(varargin{:});
end
```

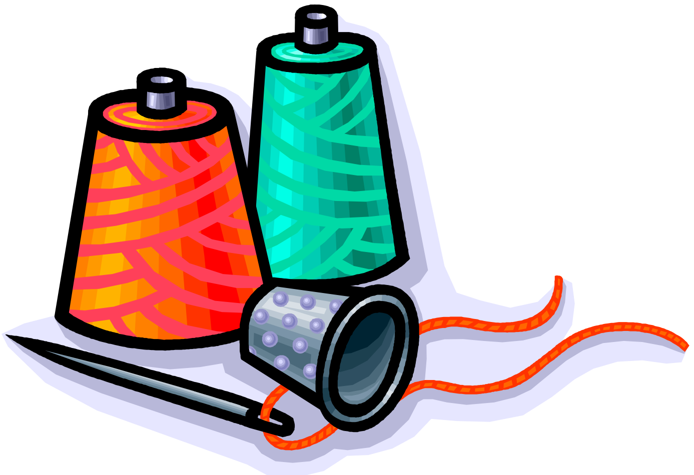
```
function gsync
end
```

```
function [ out ] = gones( varargin )
  out = ones(varargin{:});
end
```

```
function [ out ] = gint32( varargin )
  out = int32(varargin{:});
end
```

- **When distributing a Jacket based toolbox (or other Jacket based code) just include the virtual_jacket library found in the folder "Toolboxes".**
  - **The path to virtual_jacket must be included when Jacket is not installed.**

Computational Threads



Introduction

- **Intel CPUs which are the most widespread CPUs for computations at the moment have some features which are important to know:**
  - They contain multiple cores (e.g. a Core i7-970 has 6 physical cores).
  - They have Hyper Threading (with twice as many threads as physical cores).
  - They have Turbo Speed – the possibility to work above the usual clock rate. However, note that this turbo speed can usually only be used for one core – if you use e.g. all cores then the traditional clock is the limit. In theoretical estimations of GFLOPS for example you can't multiply the maximum number of cores with the maximum turbo frequency – this can't happen. The reason is power/ thermal issues.
- **MATLAB has a possibility to set the maximum number of computational threads used in the MATLAB session – maxNumCompThreads(#threads) where #threads is the number of threads we want to use.**
- **Some MATLAB functions take advantage of the multi-core possibility** – matrix multiplication for example. But how many cores we actually get access to is beyond user control and is decided between MATLAB, the operating system, and the CPU. We can only set the maximum.
- Beware that threads is a virtual (software) concept – this is used to better keep the CPU occupied. One thread being executed on a physical core and one thread waiting until the core has resources to use on the waiting thread.

Computational Threads
Test

- **Code:**

```
%% USER INPUT
MAXTHREADS = 16;
N = 3000;

%% SINGLE
Ac = randn(N,N,'single');   Bc = randn(N,N,'single');
Ag = gsingle(Ac);   Bg = gsingle(Bc);   geval(Ag,Bg);
matlab_SP = zeros(MAXTHREADS,1);
jacket_SP = zeros(MAXTHREADS,1);

% Perform test
for jj=1:MAXTHREADS
  maxNumCompThreads(jj);
  matlab_SP(jj) = N^2*(2*N-1)/(1E9*timefun(@() Ac*Bc, 2));
  jacket_SP(jj) = N^2*(2*N-1)/(1E9*gtimefun(@() Ag*Bg, 2));
end

%% DOUBLE
sys = gpu_entry(13);
if sys.compute > 1.2
  % Same principle as for single
end
```

A possibly used dual CPU system has 2×4=8 physical cores and 16 threads available. We measure the floating point performance via matrix multiplication for both MATLAB and Jacket with threads from 1 to the maximum 16. Obviously we adjust the code compared to the available maximum number of threads. We do the test for both single and double precision.

Computational Threads
Results

- **GeMM performance versus max. number of computational threads**
  - Colfax CXT2000i: Intel Core i7-970 and **NVIDIA Quadro 4000, Windows 7 x64 Enterprise, Jacket 1.7.1.**

	MATLAB		Jacket	
# Threads	Single GFlops	Double GFlops	Single GFlops	Double GFlops
1	24.6	12.3	206.7	116.3
2	48.5	24.4	205.9	116.4
3	48.7	34.4	206.7	116.3
4	92.7	46.2	207.6	116.3
5	91.6	49.9	206.4	116.3
6	90.3	49.7	206.8	116.3
7	102.7	47.0	207.2	116.3
8	107.6	46.3	207.8	116.4
9	109.0	53.3	209.9	116.3
10	110.7	50.4	206.7	116.3
11	103.0	54.3	206.3	116.3
12	98.4	48.2	206.2	116.3

MATLAB: the performance clearly depends on the max. number of computational threads. Increasing #Threads does not always increase performance – this can't be controlled by the user unfortunately. As expected we get roughly the same result for #Threads from 6-12 – there is only 6 physical cores. There is some variation as we can only set the max. number of threads.

JACKET: Even one core can easily keep matrix multiplication running at full speed, which is hardly surprising. The entire task is offloaded to the GPU and is kept there until done.

Computational Threads
Results

- **GeMM performance versus max. number of computational threads**
  - Colfax GPU HPC: Dual Intel Xeon X5570 **and NVIDIA Tesla C2070**, **Ubuntu Linux**, **Jacket 1.7.**

# Threads	MATLAB Single GFlops	MATLAB Double GFlops	Jacket Single GFlops	Jacket Double GFlops
1	25.4	12.7	602.1	267.3
2	50.4	25.2	602.2	267.4
3	50.5	37.4	602.2	267.3
4	100.1	49.9	602.4	267.2
5	99.3	59.3	602.4	267.3
6	142.6	71.2	602.3	267.4
7	142.6	81.5	602.3	267.4
8	184.5	91.8	602.4	267.4
9	184.6	92.7	602.3	267.4
10	184.6	92.5	602.3	267.3
11	184.6	92.2	602.4	267.3
12	184.4	92.3	602.4	267.4
13	184.6	92.8	602.3	267.3
14	184.5	92.2	602.5	267.3
15	184.6	92.1	602.4	267.3
16	184.5	92.5	602.4	267.3

MATLAB: the performance clearly depends on the max. number of computational threads. Increasing #Threads does not always increase performance – this can't be controlled by the user unfortunately. As expected we get the same result for #Threads from 9-16 – there is only 8 physical cores and here we can easily keep them busy. Therefore nothing to gain above #Threads=8.

JACKET: Even one core can easily keep GeMM running at full speed which is hardly surprising. The entire task is offloaded to the GPU and is kept there until done.

Results

- **GeMM performance versus max. number of computational threads**
  - Colfax GPU HPC: Dual Intel Xeon X5670 **and NVIDIA Tesla C2070, Ubuntu Linux, Jacket 1.7.**

# Threads	MATLAB Single GFlops	MATLAB Double GFlops	Jacket Single GFlops	Jacket Double GFlops
1	25.4	12.5	602.1	267.3
2	50.3	24.8	602.1	267.4
3	50.3	36.9	602.3	267.3
4	99.6	49.1	602.1	267.3
5	95.6	58.6	602.2	267.3
6	141.4	70.0	602.3	267.3
7	141.2	80.1	602.1	267.3
8	180.1	92.5	602.1	267.2
9	160.6	92.1	602.1	267.3
10	105.8	52.4	602.2	267.3
11	106.8	52.4	602.3	267.3
12	124.6	135.5	602.3	267.4

MATLAB: the performance clearly depends on the max. number of computational threads. Increasing #Threads does not always increase performance – this can't be controlled by the user unfortunately. As expected we get the same result for #Threads from 8-16 – there is only 8 physical cores and here we can easily keep them busy. Therefore nothing to gain above #Threads=8.

JACKET: Even one core can easily keep GeMM running at full speed which is hardly surprising. The entire task is offloaded to the GPU and is kept there until done.

Results

- **GeMM performance versus max. number of computational threads**
  - Colfax GPU HPC: Dual Intel Xeon X5670 **and NVIDIA Tesla C2070;** **Ubuntu Linux**, **Jacket 1.7.**

# Threads	MATLAB Single GFlops	MATLAB Double GFlops	Jacket Single GFlops	Jacket Double GFlops
13	124.6	135.5	602.2	267.2
14	124.8	135.7	602.0	267.4
15	126.4	135.7	602.3	267.3
16	126.0	135.7	602.3	267.3
17	126.2	135.8	602.2	267.4
18	125.8	135.7	602.3	267.3
19	125.9	62.8	602.1	267.3
20	126.4	135.4	602.2	267.3
21	125.8	135.5	602.3	267.3
22	126.1	135.9	602.3	267.3
23	125.8	135.7	602.2	267.3
24	126.0	135.6	602.2	267.3

MATLAB: the performance clearly depends on the max. number of computational threads. Increasing #Threads does not always increase performance – this can't be controlled by the user unfortunately. As expected we get the same result for #Threads from 13-24 – there is only 12 physical cores and here we can easily keep them busy. Therefore nothing to gain above #Threads=12.

JACKET: Even one core can easily keep GeMM running at full speed which is hardly surprising. The entire task is offloaded to the GPU and is kept there until done.

Handling Scalars in Jacket

x1 = single(3.245356412) ?

x3 = double(32.3435996) ?

x2 = gsingle(11.4356201) ?

x4 = gdouble(53.1326412)

Handling Scalars in Jacket

- **Normally we need to care about how arrays are described to take take advantage of Jacket.**
- **But for scalars things are a bit different.**
- **The following shows some examples and the results –** the important lesson to learn is that it should be checked if one or the other definition of scalars is the best.

Handling Scalars in Jacket

- **Code `computefun.m`:**

```
function [ R ] = computefun( a, k1, k2 )
  RPT = 2500;
  Rt = zeros(RPT,1,class(a));

  for rpt=1:RPT
    Rx = k1*k2*a - k2*k1.^2*a.^2 ...
         + k1*k2.^2*a.^3 - k2*k1.^3*a.^4 ...
         + k1*k2.^4*a.^5 - k2*k1.^5*a.^6 ...
         + k1*k2.^7*a.^7 - k2*k1.^8*a.^8 ...
         + k1*k2.^9*a.^9 - k2*k1.^10*a.^10 ...
         + k1*k2.^11*a.^11 - k2*k1.^12*a.^12 ...
         + k1*k2.^13*a.^13 - k2*k1.^14*a.^14 ...
         + k1*k2.^15*a.^15 - k2*k1.^16*a.^16 ...
         + k1*k2.^17*a.^17 - k2*k1.^18*a.^18 ...
         + k1*k2.^19*a.^19 - k2*k1.^20*a.^20;
    Rt(rpt) = sum(abs(fft(Rx)).^2)/(length(a)^2);
  end
  R = sum(Rt);
end
```

`computefun.m` is a mix of various computations involving the input vector a and the two constants k1 and k2.

When called from the master program, "a" is always a Jacket type, and "k1" and "k2" are various combinations of MATLAB and Jacket variables.

Handling Scalars in Jacket

- **Key part of the `master_computefun.m` script:**

```
% User must set the length of the vector
LEN = 2^21; % Or whatever

% Create reference data
aref = randn(LEN,1); a = gdouble(aref);
k1ref = pi/4;  k2ref = pi/5;

%% CONSTANT-1: CPU, CONSTANT-2: CPU
k1 = double(k1ref);   k2 = double(k2ref);
[R] = computefun(a, k1, k2); geval(R);
gsync; tstart = tic;
[R] = computefun(a, k1, k2); geval(R);
gsync; T_cpu_cpu = toc(tstart);

%% CONSTANT-1: GPU, CONSTANT-2: CPU
k1 = gdouble(k1ref);  k2 = double(k2ref);
[R] = computefun(a, k1, k2); geval(R);
gsync; tstart = tic;
[R] = computefun(a, k1, k2); geval(R);
gsync; T_gpu_cpu = toc(tstart);
```

Handling Scalars in Jacket

- **Key part of the `master_computefun.m` script (cont'd):**

```
%% CONSTANT-1: CPU, CONSTANT-2: GPU
k1 = double(k1ref);   k2 = gdouble(k2ref);
[R] = computefun(a, k1, k2); geval(R);
gsync; tstart = tic;
[R] = computefun(a, k1, k2); geval(R);
gsync; T_cpu_gpu = toc(tstart);

%% CONSTANT-1: GPU, CONSTANT-2: GPU
k1 = gdouble(k1ref);   k2 = gdouble(k2ref);
[R] = computefun(a, k1, k2); geval(R);
gsync; tstart = tic;
[R] = computefun(a, k1, k2); geval(R);
gsync; T_gpu_gpu = toc(tstart);

%% CONSTANT-1: Jacket, CONSTANT-2: Jacket
[R] = computefun(a, k1ref, k2ref); geval(R);
gsync; tstart = tic;
[R] = computefun(a, k1ref, k2ref); geval(R);
gsync; T_jkt_jkt = toc(tstart);
```

Handling Scalars in Jacket

- **Results:**
  - Intel Xeon X5570 with 48 GB of memory**, NVIDIA Tesla C2070; Ubuntu Linux, Jacket 1.7.1.**

```
>> master_computefun

SINGLE PRECISION:
======================================
k1: CPU,  k2: CPU   >>      5.046 [ms]
k1: CPU,  k2: GPU   >>      6.430 [ms]
k1: GPU,  k2: CPU   >>      6.363 [ms]
k1: GPU,  k2: GPU   >>      6.634 [ms]
k1: JKT,  k2: JKT   >>      4.428 [ms]

JKT = Jacket decides
======================================
```

```
>> master_computefun

DOUBLE PRECISION:
======================================
k1: CPU,  k2: CPU   >>      4.755 [ms]
k1: CPU,  k2: GPU   >>      5.427 [ms]
k1: GPU,  k2: CPU   >>      5.436 [ms]
k1: GPU,  k2: GPU   >>      6.019 [ms]
k1: JKT,  k2: JKT   >>      4.754 [ms]

JKT = Jacket decides
======================================
```

- **Conclusion #1: so this means that we should let Jacket decide – i.e. let the scalars initially just be MATLAB defined and let Jacket convert internally if it wants to.**

- **Conclusion #2: Quite surprising we experience quite reduced performance of we force the variables to be of Jacket type.**

Is Jacket Column or Row Major?

$$\begin{array}{cc} & ? \\ ? & \begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & & \vdots \\ a_{m,1} & \cdots & a_{m,n} \end{bmatrix} \end{array}$$

Is Jacket Column or Row Major?

- In terms of performance it is important to know how data is organized to traverse memory the most efficient way
- **MATLAB is known to be column major** – but the question is if Jacket is the same. Also it is interesting to see how much we loose if we access data in the "wrong" way
- **As a small test we perform the following:**

sum(A,2) = transpose(sum(transpose(A,1)))

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & & \vdots \\ a_{n,1} & \cdots & a_{n,n} \end{bmatrix}$$

sum(A,1) = transpose(sum(transpose(A,2)))

- The transpose approach above leads to the same results but computationally the 4 methods are different and will most likely lead to different execution times.

Is Jacket Column or Row Major?

- **Key parts of the code (major.m):**

```
%% Define matrix dimensions and repetitions
N = 4000;   TMIN = 2;

%% Define reference matrix for CPU and GPU
Ac = randn(N,N,'double'); Ag = gdouble(Ac); geval(Ag);

%% Here we do the sum down all colums to produce a row vector
Tc_cols1 = 1E3*timefun(@() sum(Ac,1), TMIN);
Tg_cols1 = 1E3*gtimefun(@() sum(Ag,1), TMIN);

%% Here we sum along all rows to produce a column vector
Tc_rows1 = 1E3*timefun(@() sum(Ac,2), TMIN);
Tg_rows1 = 1E3*gtimefun(@() sum(Ag,2), TMIN);

%% Here we transpose, sum across rows, and transpose again
Tc_cols2 = 1E3*timefun(@() transpose(sum(transpose(Ac),2)), TMIN);
Tg_cols2 = 1E3*gtimefun(@() transpose(sum(transpose(Ag),2)), TMIN);

%% Here we transpose, sum down columns, and transpose again
Tc_rows2 = 1E3*timefun(@() transpose(sum(transpose(Ac),1)), TMIN);
Tg_rows2 = 1E3*gtimefun(@() transpose(sum(transpose(Ag),1)), TMIN);
```

N is the square matrix size; TMIN is the minimum time over which the measurements are performed.

The TIMEFUN and GTIMEFUN functions are used to time the operations.

All times are in ms.

Is Jacket Column or Row Major?

- **Some data for the used CPUs and GPUs in the test:**

GPU type	Cores [—]	Mem. [MB]	Mem. BW [GB/s]	Power [W]	CUDA [—]
Core i7—975	4	12 GB	25.4	130	—
Core i7—970	6	24 GB	25.4	130	—
Xeon X5570	4	48 GB	32.0	95	—
Xeon X5670	6	192 GB	32.0	95	—
GeForce GTX465	352	1024	102.6	200	2.0
GeForce GTX580	512	1536	192.4	244	2.0
Quadro FX3800	192	1024	51.2	108	1.3
Quadro 2000	192	1024	41.6	62	2.1
Quadro 4000	256	2560	89.6	142	2.0
Tesla C1060	240	4096	102.0	188	1.3
Tesla C2070	448	6144	144.0	238	2.0

Is Jacket Column or Row Major?

- **Result of test (major.m) with N=4000 and double precision matrix:**
  - Colfax GPU HPC: dual Intel Xeon X5570 with 48 GB memory, **NVIDIA Tesla C2070; Ubuntu Linux, Jacket 1.7.1.**

```
>> major

Col sum: Sum down columns:
>> CPU:                  3.44455 [ms]
>> GPU:                  2.09927 [ms]
Col sum: Sum of transpose across rows and transposed again:
>> CPU:                 51.78325 [ms]
>> GPU:                 12.67366 [ms]

Row sum: Sum along rows:
>> CPU:                  6.31172 [ms]
>> GPU:                 10.02469 [ms]
Row sum: Sum of transpose down columns and transposed again:
>> CPU:                 46.44008 [ms]
>> GPU:                  4.76021 [ms]
```

First of all notice that Jacket AND MATLAB are both **column major**. It is significantly faster to do operations down columns than across rows.

Second, MATLAB is quite affected – 3.4 ms versus 6.3 ms.

Third, Jacket is way faster than MATLAB. But Jacket is also very sensitive to how data is access. Jacket can do the sum down columns in 2.1 ms but it takes 12.7 ms to sum across rows. If we use the alternative approach and transpose, sum across columns, and transpose yet again, we only use 4.8 ms using Jacket (direct row operations costs 10.1 ms).

Is Jacket Column or Row Major?

- **Result of test (major.m) with N=4000, and double precision matrix:**
  - Colfax GPU HPC: Intel Core i7-970 with 24 GB memory, **NVIDIA Quadro 4000; Windows 7 x64, Jacket 1.7.1.**

```
>> major

Col sum: Sum down columns:
>> CPU:                 7.17746 [ms]
>> GPU:                 4.82869 [ms]
Col sum: Sum of transpose across rows and transposed again:
>> CPU:                82.13320 [ms]
>> GPU:                27.89012 [ms]

Row sum: Sum along rows:
>> CPU:                 8.45769 [ms]
>> GPU:                23.23092 [ms]
Row sum: Sum of transpose down columns and transposed again:
>> CPU:                81.74223 [ms]
>> GPU:                 9.40878 [ms]
```

Also here we note that both Jacket AND MATLAB are both **column major. It is significantly faster to do operations down columns than across rows.**

Second, MATLAB is not that much affected – 7.2 ms versus 8.4 ms for column and row operations, respectively.

Third, Jacket is substantially faster than MATLAB. But where MATLAB only has a small difference between row/column operations the difference is a factor 5 (sum across rows is a factor 6 slower than sum down columns). It takes 4.4 ms to sum down columns and it takes 26.2 ms across rows. If we use the alternative approach and transpose, sum across columns, and transpose yet again, we only use 8.9 ms using Jacket (direct row operations costs 21.7 ms). The extra operations are easily paid for due to column sum savings.

Further Reading

1) Patrick Vandewalle, Jelena Kovacevic, and Martin Vetterli: "Reproducible Research in Signal Processing [What, why, and how]". IEEE Signal Processing Magazine, May 2009, pp. 37-47.