Parallel High Performance Computing

With Emphasis on Jacket Based GPU Computing

Advanced Programming

Torben Larsen
Aalborg University

Outline

1) GPU characteristics
   - Architecture
   - Strengths/Weaknesses in Relation to Computing

2) Stop-Resume Paradigm
   - Objective
   - Procedure
   - Example

3) GFOR Loops
   - Objectives
   - Working Principle
   - Characteristics
   - Case Study: Computational Function
   - Memory

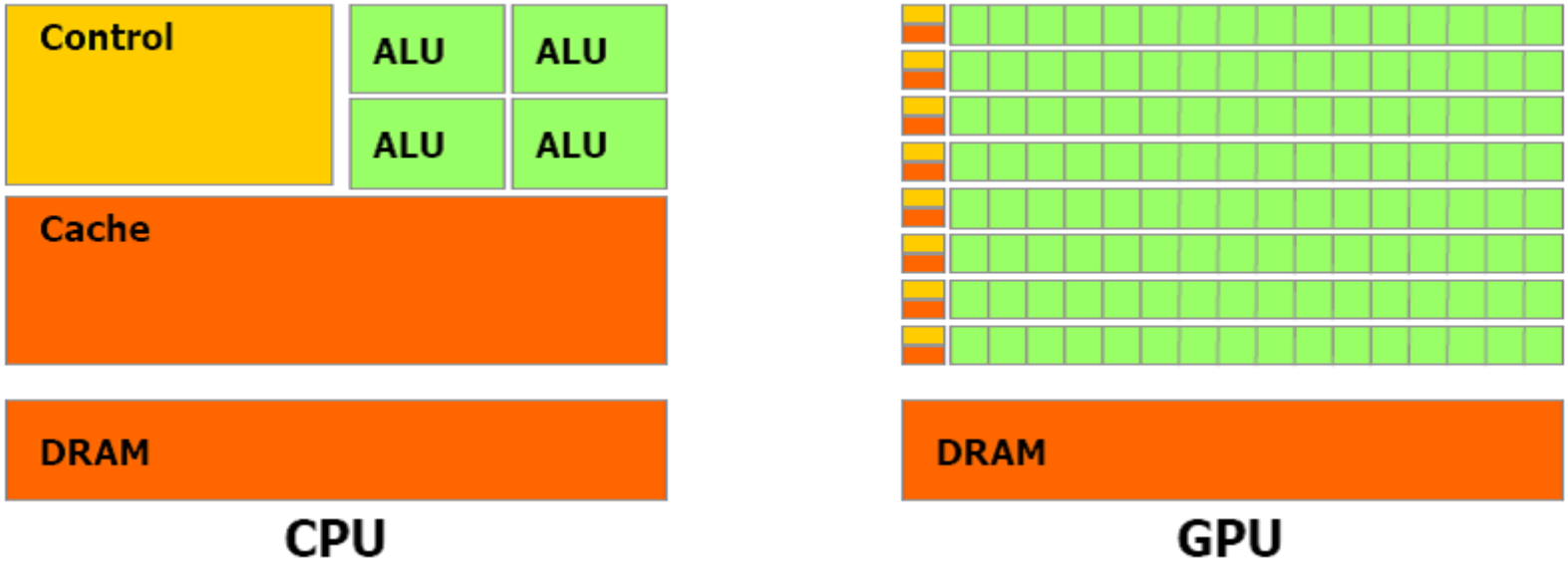
4) Handling Large Amounts of Data
   - Problem and Methodology
   - Conclusions

5) Case Study: Jacobi and Hessian Computation
   - The Problem
   - Implementation #1
   - Results #1
   - Possible Improvements
   - Implementation #2
   - Results #2

GPU Characteristics

GPU Characteristics
Architecture

- **A short reminder with a rough sketch of CPU and GPU architectures:**



- **GPUs:**
  - **Many cores – several hundreds.**
  - **Newest technologies with same relative single/double precision capability as CPUs.**
  - **Simpler control logics than for CPUs.**
  - **Less memory (register/cache) per core than for CPUs.**
  - **Needs a host (CPU) system – connected via a PCIe bus as things are now.**

GPU Characteristics

Strengths/Weaknesses in Relation to Computation

GPUs:

- Very fast access to memory (best case: 150 GB/s).
- High number of parallel computational units (>500 cores).
- Little register/cache memory.
- Normally way less memory than the CPU counterpart.
- Latency and transfer rates to move data across the PCIe bus.
- Limited control logic.
- Still immature software support.
- ~5-7 GFLOPS/W.

CPUs:

- Reasonably access to memory (best case: 20-30 GB/s).
- Still only limited number of cores (up to 6).
- Plenty of register and different level cache memory.
- Almost as much memory as you can afford.
- No PCIe bus involved and hence no overhead for this.
- Advanced control logic allows fancy operations.
- Mature software – at least for single core applications.
- ~1 GFLOPS/W.

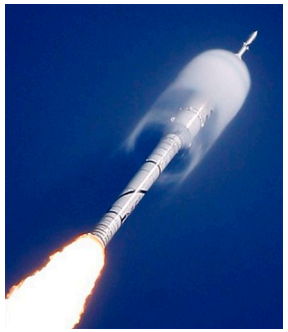
GPU Characteristics
Strengths/Weaknesses in Relation to Computation

- **Some important aspects to GPU usage in a HPC framework:**
  - **Data access** – moving data from CPU memory to GPU memory calls for use of the PCI bus with max. theoretical transfer rates of 8 GB/s. **In reality expect around 100-200 μs latency and 3-4 GB/s for CPU-GPU transfers and 2-2.5 GB/s for GPU-CPU transfers.**
  - **Extremely fast access to GPU memory (100-150 GB/s)** – but only recent GPUs have cache memory worth mentioning.
  - **Many computational cores** – generally not advanced control logic.
  - Generally much, much faster doing single precision computations than double precision – all is IEEE-754 though, so it's not bad at all. Recent Fermi architecture GPUs show the same ½ relation between double and single precision floating point operations as we also see for CPUs.
  - **Limited amount of memory** – typically in the range 1-6 GB for more powerful GPUs.

Stop-Resume Paradigm

- **Characteristics of HPC programming:**

Large and expensive equipment



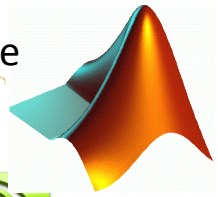
High complexity HW/SW

Large amounts of data

Failure prone

Huge execution time

Involved code



Stop-Resume Paradigm
Objective

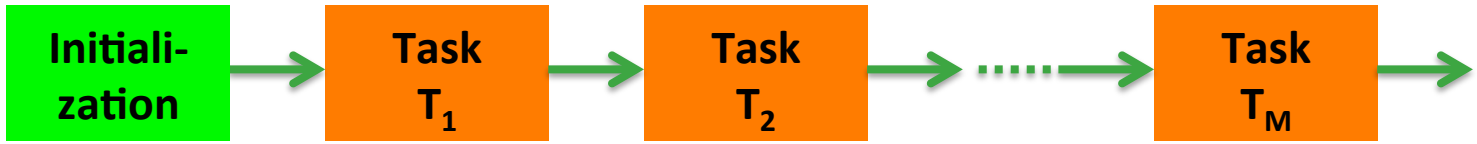
- **Because of …**
  - High complexity in software and hardware …
  - Advanced and state-of-the-art HPC facility …
  - Software still under development …
  - Long execution time …
  - And so on …

- **You should be prepared for:**

Therefore: make the code in such a way that you can restart/resume the program in case of an unexpected crash – either caused by hardware or software.
It does happen, you know.

Stop-Resume Paradigm
Procedure

- The first thing to clarify is to identify what tasks are done over time – it could for example be a loop. **We describe the problem as a number of tasks $T_1,\ldots,T_M$.**
- The structure is typically:



- It is important to recognize that the tasks are not at all required to be parallel or anything like that. In fact they are typically formulated as sequential operations – but some can hopefully be computed in arbitrary order.

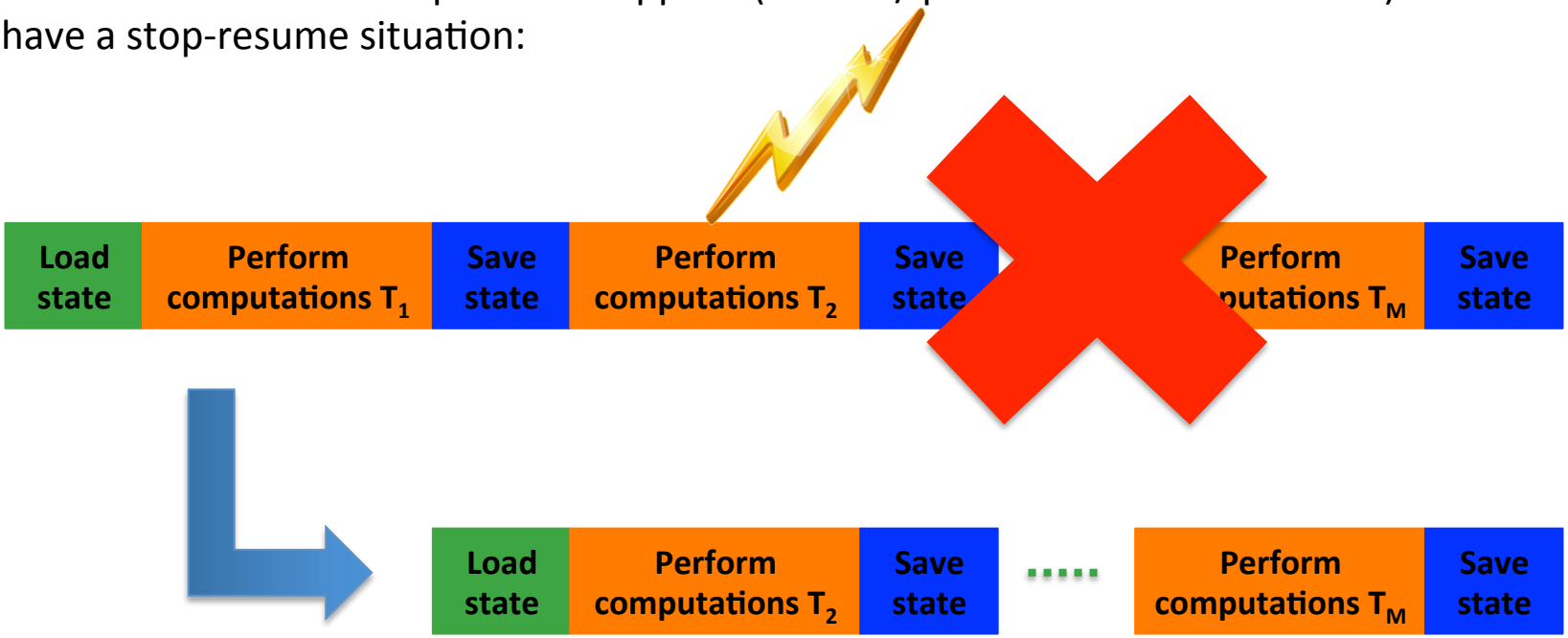
- To facilitate a stop-resume paradigm we need a more general approach to defining tasks. We need a task m=1,…,M to be defined as:
  - Read state defining variables.
  - Compute the task.
  - Save the state of variables to disk (or whatever) when all task m computations are done.

- If everything works fine we obviously can just continue from task m to task m+1 without reading the state after task m (since this is already exist in the workspace). But we obviously must save all necessary state variables from task m.

Stop-Resume Paradigm
Procedure

- In the case where everything works as planned and no stops are made, the computation scheme follows the structure:

- And if some unforeseen problem happens (reboot / power failure or whatever) we have a stop-resume situation:



Stop-Resume Paradigm
Procedure

- **So how can this be done?**
- **One very, very typical situation is that you work with loops … Let's take a simple example:**
  - Say we generate random data in a vector, make an FFT, and add the magnitude squared values.
  - We need to do this N times – where N may be e.g. 1E16.
- **So in the event of a crash happening for N=1E15 we really loose big time.**

```
% Initialization of variables
# GetState >> n1, N, R, PRNGstate from
# file if exist
# else from function

% Compute
for n=n1:N
    R = R + whatever;
    # pushState(n,R,PRNGstate) every 100 n
    # >> save data in file
end

% Print result
fprintf('Result: %6.3g\n', R);
delete(file)
```

1. We get initialization data from a file if this exist (if it exist it is because something went wrong during an execution); else from a function.
2. We compute the loop. Note the start value for n is n1. So in case of normal operation, n1=1 and if read from file it depends on when the code stopped.
3. For every 100 (or whatever) n-values we push the state of all relevant variables to the disk. Too often reduces performance; too rare means we loose more. It is an insurance question basically.
4. Delete the file – its mission is now completed. When we are here everything is done.

Stop-Resume Paradigm
Example

- **Functions to push state and to fetch state:**

```
function [ ] = pushState(Fname, n, N, A, R)
  if rem(n,3)~=0
    return;
  else
    % Save state of n, N, A and R
    save(Fname, 'n', 'N', 'A', 'R');
  end
end
```

For every 3 n-values we push the state to a file. How often is a question of how much we are willing to loose in case of a crash.

We save the state of the stream used to generate pseudo random number values. This is more tricky than in earlier versions and we must be careful not to make mistakes here.

```
function [N1,N,A,R] = fetchState( Fname )
  if ~exist(Fname, 'file')
    N = 12;
    N1 = 1;
    R = zeros(N,1);
    reset(RandStream.getDefaultStream);
    A = randn(N,N);
  else
    load(Fname);
    N1 = n+1;
  end
end
```

Here we fetch the state. When the file Fname does not exist (when we start first time) we set N, N1 and R to the initial values.

When we are then requested the state based on an existing Fname file, we load the file and restore the state, and we set N1 according to how far into the n-values we know R.

Stop-Resume Paradigm
Example

- **To maintain an easy to read code, two functions are defined:**
  - One which pushes data to a file every 3 iterations, and
  - One which sets the desired data or reads data from the earlier generated file

master.m

```
% Name of data file
Fname = 'data.mat';

% Define variables from scratch or file
[N1,N,A,R] = fetchState(Fname);

% Initialize data
for n=N1:N
  % Compute
  R(n) = sum(A(:,n),1);
  pause(0.3);
  fprintf('n >> %6.0f     %16.6f\n', n,R(n));

  % Save result and state every 3 iterations
  pushState(Fname, n, N, A, R);
end

% Print result and dele file (mission completed)
fprintf('Result completed: %12.6f\n', sum(R(:)));
delete(Fname);
```

Provide name for data file.
Get variables by calling the fetchState function. This function either reads data from disk (if an error happened last time – shown by the files existence) or reads data from the function.

Inside the loop, the state of variables is pushed via the pushState function, which writes data to disk. This is done for every 3 n-values in this case.

We print the result and delete the data file. The data file must be deleted not to disturb the next run – if the file exist the next run will incorrectly use this data file. A more advanced technique may of course be used.

Stop-Resume Paradigm
Example

- **Results:**
  - MacBook Air with Intel Core 2 Duo 2.13 GHz and 4 GB memory; MATLAB R2011a; Jacket 1.7.1.

```
>> master
n >>      1              7.927858
n >>      2              7.175391
n >>      3             -2.189328
n >>      4             -0.914683
n >>      5              1.107893
n >>      6              1.414232
n >>      7              0.717868
n >>      8              1.318465
n >>      9             -3.808732
n >>     10             -2.022484
n >>     11              1.605775
n >>     12             -3.183660
Result completed:     9.148595
```

Top: Un-interrupted execution leading to an expected result.

Right: Interrupt of execution with CTRL-C – and then a resume. After the first "master" command the file **data.mat** is generated. And the state is read from this in the second master run.

```
>> master
n >>      1              7.927858
n >>      2              7.175391
n >>      3             -2.189328
n >>      4             -0.914683
n >>      5              1.107893
n >>      6              1.414232
n >>      7              0.717868
n >>      8              1.318465
??? Operation terminated by user
during ==> master at 12

>> master
n >>      7              0.717868
n >>      8              1.318465
n >>      9             -3.808732
n >>     10             -2.022484
n >>     11              1.605775
n >>     12             -3.183660
Result completed:     9.148595
```

GFOR Loops

GFOR Loops
Objectives

- **Loops is probably the single most important possibility in programming – it supports what computers do best: repeat-repeat-repeat-… with modified inputs.**
- FOR-loops:

```
% Pre-allocate output vector
N = 10;
R = zeros(N,1,'single');

for ii=1:N
  R(ii) = 100*rand();
end
```


This loop is done like:

Time step 1: ii=1 >> R(1)
Time step 2: ii=2 >> R(2)
.
.
.
Time step N: ii=N >> R(N)

- **Characteristics:**
  - Computation is done in the order ii=1,2,…,N
  - Computation time is roughly N times the time it takes to perform one of R(1),…,R(N)
  - Registers etc. can be reused so memory consumption (besides for R) does not increase with N

GFOR Loops
Working Principle

- **GFOR is a Jacket loop construct and only works with Jacket variables.**
- **GFOR-GEND works significantly different from MATLABs FOR-END construct.**

```
% Pre-allocate output vector
N = 10;
R = gzeros(N,1,'single');

gfor ii=1:N
  R(ii) = 100*grand();
gend
```

This loop is in principle (enough cores) done like:

Time step 1: ii=1 >> R(1), … ii=N >> R(N)

So all ii's are issued for computation at the same time. What happens is that N threads are formed and all are submitted to the GPU scheduler for computation.

This is data-parallel computing – multiple cores working on each its part of the dataset.

- **Characteristics**:
  - Computation is in principle done simultaneously – each ii value makes a copy of the problem "R(1) =100*rand();" … "R(N)=100*rand();".
  - Since the problem is "copied" N times we need N times the memory required for doing one ii computation.
  - Computation time may be significantly reduced – it does depend on the computation, number of computational cores etc. If all cores are need for one ii then the gain is obviously non-existent.

GFOR Loops
Working Principle

- Let's see the result first in the two cases:

```
>> for_example

R =

   81.4724
   90.5792
   12.6987
   91.3376
   63.2359
    9.7540

>>
```

The standard for loop produces the expected result where the values in the R vector are different.

```
>> gfor_example

R =

    8.9191
    8.9191
    8.9191
    8.9191
    8.9191
    8.9191

>>
```

Here we see one of the things to take care of. As all ii's are issued independently and at the same time, they have the same state of the rand generator. Therefore all of them produce the same result.

GFOR Loops
Characteristics

- We always need random arrays predefined before the GFOR-GEND loop:

```
% Pre-allocate output vector
N = 10;
R = gzeros(N,1,'single');

a = grand(N,1,'single');
gfor ii=1:N
  R(ii) = 100*a(ii);
gend
```

```
>> gfor_example_2

R =

   41.7847
   74.2069
    9.2483
   88.5878
   77.4584
   73.4144

>>
```

- It doesn't even help to initialize the pseudo random number generator inside the loop since the loop body is only covered once.

GFOR Loops
Characteristics

- **GFOR restrictions:**
  - Iterator independence – all loop bodies must be able to run independently from all the other ones. This means that R(1),…,R(N) must be able to be computed independently. So R(1) can't depend on e.g. R(N).

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

  - No conditional statements – i.e. no if-elseif-else-end statements inside the GFOR body. Use multiplication of glogicals to cheat this restrictions. Dividing the loop into two GFOR's where each corresponds to one condition may also be a solution.

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

  - Nested GFOR-GFOR loops are not supported. It is possible to use GFOR loops inside for loops though (and sometimes also the other way around).

```
for k = 1:n
  gfor j = 1:m  % good
  % ...
  gend
end
```

  - The loop iterator is not available outside the GFOR loop.

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

GFOR Loops
Characteristics

- **GFOR restrictions:**
  - In terms of memory the problem with **GFOR** is that it takes the full amount of memory for EACH loop iterator value. So a loop iterator ii=1:N requires N times the memory of just one ii value.

```
% INSUFFICIENT MEMORY
gfor k = 1:400
  B = A(:,k);
  C(:,:,k) = B*B'; %  - mem.
gend
```

  - The solution in case of insufficient memory is to apply a for loop around the gfor to reduce the memory load.

```
% REDUCED MEMORY
for kk = 1:100:400
  gfor k = kk:kk+100-1 % four
    B = A(:,k);
    C(:,:,k) = B*B'; %  + mem.
  gend
end
```

  - The iterator must be uniformly spaced.

```
gfor i = 1:n          % good
gfor i = m:n          % good
gfor i = 5:2:100      % good
gfor i = 1:2:n        % good
gfor i = [1 4 2 3]    % bad
```

GFOR Loops
Characteristics

- **GFOR restrictions:**
  - Colon expressions are not allowed in GFOR loops. Only in the loop iterator itself. The solution is to use vector arithmetic to access the indexing.

```
A = gones(100,100);
gfor k = 5:95
  A(k-4:k+4,k-4:k-4) = ... % -
gend
```

```
A = gones(100,100);
idx = gsingle(-4:4);
gfor k = 5:95
  A(k+idx,k+idx) = ...     % +
gend
```

GFOR Loops

Case Study: Computational Function

- **Let's take a case study and compare the performance in a simple case.**
- **We have a simple computational code in 3 versions:**
  - Version 1: This is using MATLAB variables and a standard FOR loop.
  - Version 2: This is using Jacket variables and a standard FOR loop.
  - Version 3: This is using Jacket variables and a GFOR loop.

```
% Version 1: MATLAB variables & FOR loop
function [ T, R ] = cpufor(m,PAGES,RPT)
  R = zeros(PAGES,1);
  t1 = tic;
  for jj=1:RPT
    for ii=1:PAGES
      a = m(:,:,ii);
      dd1 = a(1,1)*(a(2,2)*a(3,3) - a(1,3)*a(2,2)*a(3,1));
      dd2 = a(1,2)*(a(2,3)*a(3,1) - a(1,1)*a(2,3)*a(3,2));
      dd3 = a(1,3)*(a(2,1)*a(3,2) - a(1,2)*a(2,1)*a(3,3));
      R(ii) = exp(0.13*sin(dd1 + dd2 + dd3));
    end
  end
  T = toc(t1)/RPT;
end
```

GFOR Loops

Case Study: Computational Function

- **And version 2 of the computational code:**

```
% Version 2: Jacket variables & FOR loop
function [ T, R ] = gpufor(m,PAGES,RPT)
  R = gzeros(PAGES,1);
  gsync; t1 = tic;
  for jj=1:RPT
    for ii=1:PAGES
      a = m(:,:,ii);
      dd1 = a(1,1)*(a(2,2)*a(3,3) - a(1,3)*a(2,2)*a(3,1));
      dd2 = a(1,2)*(a(2,3)*a(3,1) - a(1,1)*a(2,3)*a(3,2));
      dd3 = a(1,3)*(a(2,1)*a(3,2) - a(1,2)*a(2,1)*a(3,3));
      R(ii) = exp(0.13*sin(dd1 + dd2 + dd3));
    end
    geval(R); % GEVAL here; matrix first ready now
  end
  gsync; T = toc(t1)/RPT;
end
```

The code is made for the GPU. Version 2 uses FOR (and version 3 uses GFOR). Remember GEVAL inside the repetition loop.

GFOR Loops

Case Study: Computational Function

- **Version 3 is here:**

```
% Version 3: Jacket variables & GFOR loop
function [ T, R ] = gpugfor(m,PAGES,RPT)
  R = gzeros(PAGES,1);
  gsync; t1 = tic;
  for jj=1:RPT
    gfor ii=1:PAGES
      a = m(:,:,ii);
      dd1 = a(1,1)*(a(2,2)*a(3,3) - a(1,3)*a(2,2)*a(3,1));
      dd2 = a(1,2)*(a(2,3)*a(3,1) - a(1,1)*a(2,3)*a(3,2));
      dd3 = a(1,3)*(a(2,1)*a(3,2) - a(1,2)*a(2,1)*a(3,3));
      R(ii) = exp(0.13*sin(dd1 + dd2 + dd3));
    gend
    geval(R); % GEVAL here; matrix first ready now
  end
  gsync; T = toc(t1)/RPT;
end
```

The code is made for the GPU. Version 3 using GFOR (where version 2 uses plain FOR). Remember GEVAL inside the repetition loop.

GFOR Loops

Case Study: Computational Function

```
function [ ] = gfor_speedtest(PAGES,RPT)
  % Define 3-D array
  ff = 10*rand(3,3,PAGES,'single'); ffg = gsingle(ff);

  %% CPU - FOR
  [ Tcpufor, Rcpufor ] = cpufor(ff,PAGES,RPT(1));

  %% GPU – FOR
  [ Tgpufor, Rgpufor ] = gpufor(ffg,PAGES,RPT(2));

  %% GPU – GFOR
  [ Tgpugfor, Rgpugfor ] = gpugfor(ffg,PAGES,RPT(3));

  %% PRINT RESULTS
  x = abs((Rcpufor - Rgpufor)./Rcpufor)*100;     maxErrGPUFOR = max(x(:));
  x = abs((Rcpufor - Rgpugfor)./Rcpufor)*100;   maxErrGPUGFOR = max(x(:));
  fprintf('FOR/MATLAB measurement time:      %12.5f\n', RPT(1)*Tcpufor);
  fprintf('FOR/Jacket measurement time:      %12.5f\n', RPT(2)*Tgpufor);
  fprintf('GFOR/Jacket measurement time:     %12.5f\n', RPT(3)*Tgpugfor);
  fprintf('Speedup CPU-FOR >> GPU-FOR:       %12.5f\n', Tcpufor/Tgpufor);
  fprintf('Speedup CPU-FOR >> GPU-GFOR:      %12.5f\n', Tcpufor/Tgpugfor);
  fprintf('Speedup GPU-FOR >> GPU-GFOR:      %12.5f\n', Tgpufor/Tgpugfor);
  fprintf('Max. percent error GPU-FOR:       %12.5f\n',  maxErrGPUFOR);
  fprintf('Max. percent error GPU-GFOR:      %12.5f\n',  maxErrGPUGFOR);
end
```

Function to perform the test of the three versions:

1. **MATLAB FOR.**
2. **Jacket FOR.**
3. **Jacket GFOR.**

The input to the function is number of pages (PAGES) and a vector containing number of repetitions for the three versions (RPT).

GFOR Loops

Case Study: Computational Function

- **Results from the GFOR_SPEEDTEST function**
  - Intel Xeon X5570 with 48 GB memory; NVIDIA Tesla C2070.
  - Ubuntu 10.04 LTS; Jacket 1.7 (4ba10c6); MATLAB 7.11.0.584 (R2010b); Driver: 260.19.44.

```
>> gfor_speedtest(11000,[1,1,1])
FOR/MATLAB measurement time:          0.02110
FOR/Jacket measurement time:         16.45428
GFOR/Jacket measurement time:         0.00960
Speedup CPU-FOR >> GPU-FOR:           0.00128
Speedup CPU-FOR >> GPU-GFOR:          2.19779
Speedup GPU-FOR >> GPU-GFOR:       1713.80856
Max. percent error GPU-FOR:           0.01225
Max. percent error GPU-GFOR:          0.01225
>> gfor_speedtest(11000,[200,1,1000])
FOR/MATLAB measurement time:          2.42066
FOR/Jacket measurement time:         16.49162
GFOR/Jacket measurement time:         2.53498
Speedup CPU-FOR >> GPU-FOR:           0.00073
Speedup CPU-FOR >> GPU-GFOR:          4.77451
Speedup GPU-FOR >> GPU-GFOR:       6505.62312
Max. percent error GPU-FOR:           0.01388
Max. percent error GPU-GFOR:          0.01388
>>
```

FOR combined with Jacket is very slow compared to a standard MATLAB type of variable.

When combining Jacket with GFOR we see a huge speedup. Now Jacket/GFOR is close to 5 times faster than MATLAB/FOR. For more computationally challenging sub-functions the speedup may easily be much larger.

When looking at FOR versus GFOR for Jacket types, the speedup of using GFOR versus FOR is more than 6,500 times – when using the more reliable run with sufficient repetition (green). Quite impressive. Comparing Jacket GFOR with MATLAB FOR is less convincing though – but there is a speed improvement. The down-side is increased memory consumption though.

GFOR Loops
Memory

- Let's consider a typical type of GFOR loop:

```
…
A = grandn(N,N,'double');
A = grandn(N,N,'double');
R = gzeros(N,N,'double');
GFOR ii=1:N
  R(:,ii) = A(:,ii) .* B(:,ii);
GEND
```

- The memory usage for the matrices **A**, **B** and **R** are:

$$\mathcal{M}_{\mathbf{A}} = 8 \cdot N^2 \qquad \mathcal{M}_{\mathbf{B}} = 8 \cdot N^2 \qquad \mathcal{M}_{\mathbf{R}} = 8 \cdot N^2$$

- Since the GFOR loop copies all N ii-values in one go it request memory according to:

$$\begin{aligned} \mathcal{M}_{\mathbf{GFOR}} &= N \cdot \{\mathcal{M}_{\mathbf{A}} + \mathcal{M}_{\mathbf{B}} + \mathcal{M}_{\mathbf{R}}\} \\ &= N \cdot \{3 \cdot 8 \cdot N^2\} \\ &= 24 \cdot N^3 \end{aligned}$$

- For N=1000 this means that we need 24 GB of memory – no GPU has that much memory and even if it did all the copying costs in performance. **Use GFOR with care!**

Handling Large Amounts of Data

Handling Large Amounts of Data
Problem and Methodology

- As mentioned already, typically we have around **1-6 GB of memory available in one GPU.**
- The typical way to handle this is the obvious solution by splitting the input data in segments where each segment can be computed.

$$\mathbf{R} = f(d)$$

d is the huge input data
f is the operation applied to d
$\mathbf{R}$ is the result of the operation

- The trick is to reformulate the problem such that we get the same result but in a different way than before:

$$\mathbf{R} = h\{g(d_1) + \cdots + g(d_N)\}$$

- **A few things are important to notice:**
  - The d input data is not necessarily identical to the sum of d_1,…, d_N. The individual d_1,…, d_N just needs to fit in the memory we have available.
  - Also note that the operations described by g and h are not necessarily identical to f. Of course it may often be so that h is a no-op (doing nothing) and g is identical to f – but it does not have to be like that.

- **Let's take a look at an example.** Suppose we have an input matrix, which we can write as a sum of column vectors:

$$\mathbf{D} = \begin{bmatrix} d_{1,1} & \cdots & d_{1,N} \\ \vdots & & \vdots \\ d_{M,1} & \cdots & d_{M,N} \end{bmatrix} = \mathbf{d}_1 + \cdots + \mathbf{d}_N; \quad \mathbf{d}_n = \begin{bmatrix} d_{1,n} \\ \vdots \\ d_{M,n} \end{bmatrix}$$

- The elements of the $\mathbf{d}$-vector are given by:

$$d_{m,n} = 5\, A_n \cos\left[2\pi \frac{f_n}{f_s} m\right] + \xi(m)$$

- The variables are chosen as:
  - A_n is a randomly chosen amplitude from a normal distribution (only positive values used and max. value 10) with standard deviation 1 and zero mean.
  - f_n is a randomly chosen frequency taken from a normal distribution (only positive values used and max. value f_{max}=5) with standard deviation 1 and zero mean.
  - $\xi(n)$ is a noise component made as a random value taken from a normal distribution with standard deviation 1 and zero mean.
  - f_s is a sampling frequency chosen as f_s=10 f_{max}.

- Suppose the D-matrix is single precision, M=2^22=4194304 and N=2000, we use approx. 8.4 GB of memory just to hold the matrix.

- In terms of processing we will apply the following to all $\mathbf{d}_n$:
  - First compute the FFT (Fast Fourier Transform).
  - Compute the mean of absolute squared value of the FFT($\mathbf{d}_n$) vector.

- The output from the processing is thus:

$$\begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix} = \begin{bmatrix} \text{mean}\left\{|\text{FFT}\{\mathbf{d}_1\}|^2\right\} \\ \vdots \\ \text{mean}\left\{|\text{FFT}\{\mathbf{d}_N\}|^2\right\} \end{bmatrix}$$

- We want the implementation to have the following features:
  - Create the total D-matrix in the MATLAB environment.
  - Make a function to perform the computing.

- Suppose we use a GFOR-loop to do the processing. The memory needed for one $\mathbf{d}$-vector and its processing is likely (at least):
  - $p \cdot M$ bytes to hold $\mathbf{d}_n$ where p=4 for single precision and p=8 for double precision.
  - Since we perform FFT the output in complex form requires $2 \cdot 4 \cdot M$ bytes for single precision.
  - Buffer of b bytes to make the computations.
- Say we submit K $\mathbf{d}$-vectors the total memory consumption is around:

$$K \; \{p\,M \quad + \quad 2\,p\,M \quad + \quad b(M)\}$$

- Say we have a total of C bytes available on the GPU and we need a buffer of B bytes for various minor and unaccounted for things.
- We can then estimate the number of $\mathbf{d}$-vectors to submit to the GPU in one go from:

$$C \; - \; B \quad = \quad K \; \{p\,M \quad + \quad 2\,p\,M \quad + \quad b(M)\}$$

- Leading to the following where the floor function is applied for a conservative approach:

$$K_{\mathrm{sp}} \; = \; \left\lfloor \frac{C \; - \; B}{12\,M \; + \; b(M)} \right\rfloor \qquad\qquad K_{\mathrm{dp}} \; = \; \left\lfloor \frac{C \; - \; B}{24\,M \; + \; b(M)} \right\rfloor$$

- **Function code:**

```
function [ R ] = foo( D )
  % foo Computes a functional response to a matrix
  %     (set of vectors) input.

  R = zeros(size(D,2),1,class(D));
  if isa(D,'garray')
    gfor ii=1:size(D,2)
      R(ii) = mean(abs(fft(d) ...
              /Ld).^2+1E-1*d.^2+1E-3*d.^3+1E-5*d.^4);
    gend
    fprintf('GPU in action ...\n');
  else
    for ii=1:size(D,2)
      R(ii) = mean(abs(fft(d) ...
              /Ld).^2+1E-1*d.^2+1E-3*d.^3+1E-5*d.^4);
    end
    fprintf('CPU in action ...\n');
  end
end
```

GFOR applied when the input is of Jacket type.

FOR applied when the input is of MATLAB type.

- **Master code to control the test:**

```
function [ tc, tg, Rc, Rg ] = master_foo( M, N )
  % master_foo Master file for the foo function.

  % Global memory buffer + GFOR iterator related memory buffer
  MemBuffer = 100E6;

  % Precision; p=4 for SP and p=8 for DP
  p = 8;

  % Set up matrix
  reset(RandStream.getDefaultStream);
  A = 5*repmat(rand(1,N,'double'),M,1);
  PHI = 2*pi*100*repmat(randn(1,N,'double'),M,1) ...
        .* repmat((1:M)',1,N)./1321E3;
  fullD = A .* cos(PHI) + randn(M,N,'double');

  % Get info on GPU memory
  clear gpu_hook;
  gpuInfo = gpu_entry(13);
  estNperRun = floor((gpuInfo.gpu_free - MemBuffer) ...
               /(3*p*M+2*p*M));
  estRuns = ceil(N/estNperRun);
  estNperRun = round(N/estRuns);
```

Estimate buffer memory.

Set up the full D-matrix.

Estimate number of vectors per call to the foo function.

- Master_foo function code to control the test (cont'd):

```
  %% CPU
  low = (0:estRuns-1)*estNperRun+1;
  hgh = [(1:estRuns-1)*estNperRun , N]
  Rc = zeros(N,1,'double');
  t1 = tic;
  for ii=1:estRuns
    D = double(fullD(:,low(ii):hgh(ii)));
    Rc(low(ii):hgh(ii)) = foo( D );
  end
  tc=toc(t1);

  %% GPU
  lowG = gsingle(low);   hghG = gsingle(hgh);
  Rg = gzeros(N,1,'double');
  gsync; t1 = tic;
  for ii=1:estRuns
    D = gdouble(fullD(:,lowG(ii):hghG(ii)));
    Rg(lowG(ii):hghG(ii)) = foo( D );
  end
  geval(Rg);   % GEVAL placed here loop fills part of the matrix
  gsync; tg=toc(t1);
end
```

MATLAB benchmarking. The matrix is reduced in size to fit what is expected for the GPU. The same is done here for comparison.

Jacket benchmarking. The matrix size is reduced to fit what is expected for the GPU.

- **Run.m script, which sets up the size of the problem and post-process the data:**

```
% run.m : Sets up the test of master_foo and foo

% Number of columns and rows in fullD
M = 2^12;
N = 80E3;

% Execute test
[tc,tg,rc,rg,npr] = master_foo(M,N);

% Max. relative error of the absolute difference
x=abs((rc-rg)./rc)*100;
MaxRelErrPct = max(x(:))

% Speedup
Speedup = tc / tg
```

Handling Large Amounts of Data
Problem and Methodology

- **Results based on:**
  - Colfax CXT2000i: Intel Core i7-970 with 24 GB memory; NVIDIA Quadro 4000; MATLAB R2011a, Jacket 1.7.1.
- **Results with M=2^12 and N=80,000 and FOR in the Jacket part – just for reference:**
  - Single precision, two groups: 1-40000, 40001-80000.
  - Double precision, three groups: 1-26667, 26668-53334, 53335-80000.

```
>> run
=========================================
SINGLE PRECISION:
# of iterations per call:      40000.000000
Max. relative error pct.:          0.000533
Mean relative error pct.:          0.000081
CPU >> GPU speedup:                1.462709
-----------------------------------------
DOUBLE PRECISION:
# of iterations per call:      26667.000000
Max. relative error pct.:          0.000541
Mean relative error pct.:          0.000081
CPU >> GPU speedup:                1.856559
=========================================
>>
```

This test compares MATLAB+FOR and JACKET+FOR.

Single Precision: the speedup is modest – slightly above 1. When using a for-loop the loop content is executed sequentially – one after another.

Double Precision: the speedup is close to 2 .

As seen from the error metrics the agreement is very good indeed.

- **Results with M=2^12 and N=80,000 and GFOR in the Jacket part:**
  - Single precision, two groups: 1-40000, 40001-80000.
  - Double precision, three groups: 1-26667, 26668-53334, 53335-80000.

```
>> run
================================================
SINGLE PRECISION:
# of iterations per call:     40000.000000
Max. relative error pct.:         0.247169
Mean relative error pct.:         0.238038
CPU >> GPU speedup:              71.231101
------------------------------------------------
DOUBLE PRECISION:
# of iterations per call:     26667.000000
Max. relative error pct.:         0.000544
Mean relative error pct.:         0.000081
CPU >> GPU speedup:              40.199516
================================================
>>
```

This test compares MATLAB+FOR and JACKET+GFOR.

Single Precision: the speedup is above 70 when using GFOR compared to a speedup slightly above 1 when using a plain FOR-loop. Three calls to the GFOR loops with approximately 26667 parallel executions (threads) helps a lot on performance.

Double Precision: a speedup above 40 for a GFOR loop compared to 1.8 for a plain FOR-loop.

As seen from the error metrics the agreement is likely sufficient.

- **Results with M=2^22 and N=80 and FOR in the Jacket part – just for reference:**
  - Single precision, two groups: 1-40, 41-80.
  - Double precision, three groups: 1-27, 28-54, 55-80.

```
>> run
============================================
SINGLE PRECISION:
# of iterations per call:     40.000000
Max. relative error pct.:      0.007242
Mean relative error pct.:      0.005126
CPU >> GPU speedup:           13.639340
--------------------------------------------
DOUBLE PRECISION:
# of iterations per call:     40.000000
Max. relative error pct.:      0.007217
Mean relative error pct.:      0.005123
CPU >> GPU speedup:           17.374977
============================================
>>
```

This test compares MATLAB+FOR and JACKET+FOR.

Single Precision: the speedup of Jacket is above 13. The foo-function is called twice with 40 columns.

Double Precision: here the speedup is above 17. The foo-function is called three times with 26-27 columns each.

As seen from the error metrics the agreement is very good indeed.

- **Results with M=2^22 and N=80 and GFOR in the Jacket part:**
  - Single precision, two groups: 1-40, 41-80.
  - Double precision, three groups: 1-27, 28-54, 55-80.

```
>> run
================================================
SINGLE PRECISION:
# of iterations per call:     40.000000
Max. relative error pct.:      0.006987
Mean relative error pct.:      0.004885
CPU >> GPU speedup:           18.183597
------------------------------------------------
DOUBLE PRECISION:
# of iterations per call:     27.000000
Max. relative error pct.:      0.007219
Mean relative error pct.:      0.005123
CPU >> GPU speedup:            7.766916
================================================
>>
```

This test compares MATLAB+FOR and JACKET+GFOR.

Single Precision: the speedup is above 18 in when using GFOR compared to a speedup above 13 when using a plain FOR-loop. Two calls to the GFOR loop with 40 parallel executions.

Double Precision: the speedup is above 7 when using GFOR compared to a speedup of 17 when using a plain FOR-loop. Three calls to the GFOR loops with 26-27 parallel executions (threads).

As seen from the error metrics the agreement is very good.

Handling Large Amounts of Data
Conclusions

- **What happens if we push a computational task to the GPU where there is insufficient memory?**

```
gpuInfo = gpu_entry(13);
estNperRun = floor((gpuInfo.gpu_free - MemBuffer)/(3*p*M+2*p*M));
```

```
gpuInfo = gpu_entry(13);
estNperRun = floor((gpuInfo.gpu_free - MemBuffer)/(3*p*M+p*M));
```

- **This causes a memory allocation fault! Insufficient memory.**

- **It is for sure worth estimating the memory requirements and use Jackets**

```
% Get info on GPU memory
clear gpu_hook;                     % Empty garbage
gpuInfo = gpu_entry(13);            % Extract various GPU info
gpuInfo.gpu_free                    % Amount of free memory
```

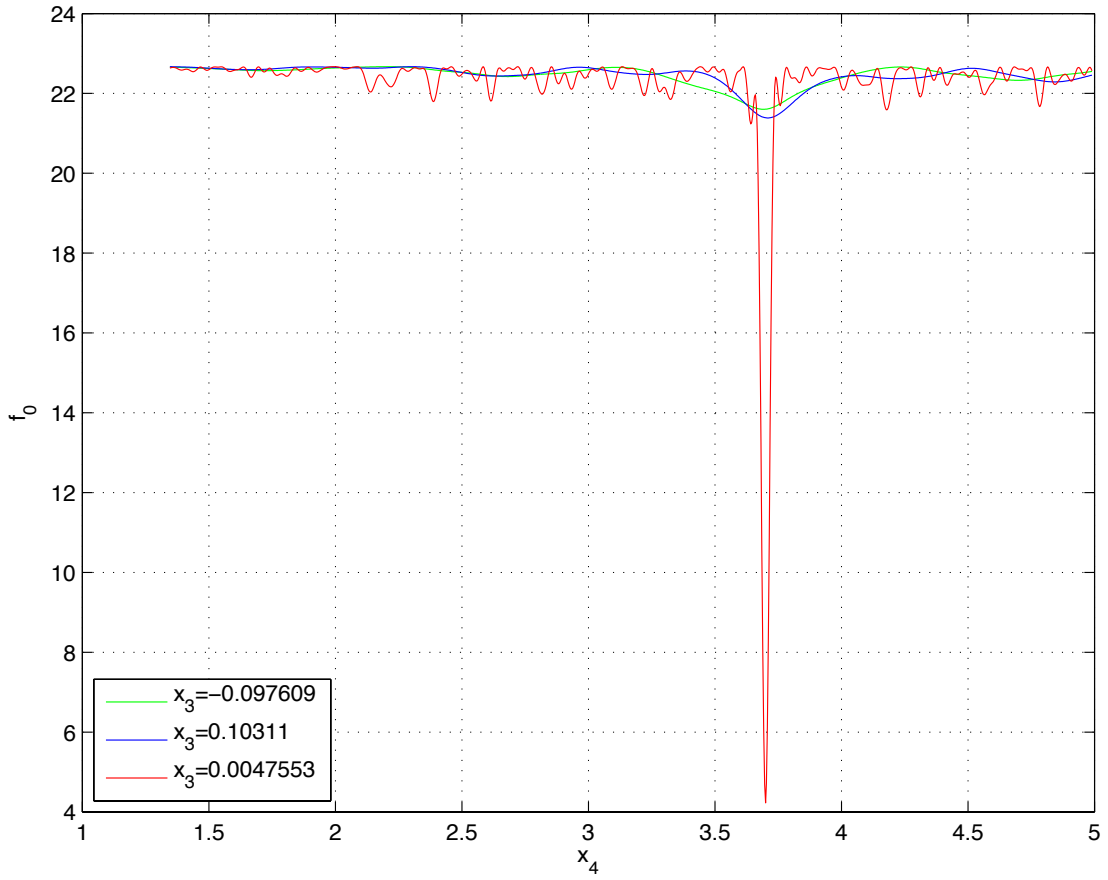
- **Conclusions**:
  - For GFOR to be efficient there should be many iterations (gfor ii=1:N – N should be large). A large N means many threads (simultaneous computations run in parallel). This allows the GPU to take advantage of its many physical computational cores.
  - Be careful to estimate the memory requirement when using GFOR – otherwise you will see memory allocation errors or in particularly nasty cases it simply omits performing the computations it does not have room for in the GPU.

- **Good programming style:**
  - For HPC problems (requiring many computation resources) **it is a MUST to save data to disk frequently** – and make the program such that it can be restarted in the identical state where it for some reason stopped.
  - **Avoid moving data across the PCI bus when at all possible** (it is a severe bottleneck at only 8 GB/s theoretical and more like 1-3 GB/s in reality). Generate data locally at the GPU when at all possible (for example rand/randn data) – even when it costs more computations (in most cases this is much faster than moving data across the PCI bus).
  - **When data MUST be moved across the PCI bus, move as large chunks at a time as possible.** It takes much longer time to move 100 x 1 MB than to move 1 x 100 MB.
  - **When possible take advantage of the fast single precision computations.** Test accuracy and be aware that for example high order filters and matrix inversion often need double precision.

Case Study: Jacobi and Hessian Computation

Case Study: Jacobi and Hessian Computation
The Problem

- **We have a nasty non-convex optimization problem with 6 variables where the objective function versus one of the optimization variables x_4 is:**



Case Study: Jacobi and Hessian Computation
The Problem

- As part of an optimization problem we need to perform the following:

$$\text{minimize} \quad f_0(\mathbf{x})$$

- where the objective function is:

$$\begin{aligned} f_0(\mathbf{x}) &= \sum_{n=1}^{N} |e(\mathbf{x};n)|^2 = \|\mathbf{e}(\mathbf{x})\|_2^2 \\ e(\mathbf{x};n) &= \{x_1 + j\,x_2\}\,t(n)\,\exp[-(x_3 + j\,\kappa\,x_4)\,(n-1)] \\ &\quad -(x_5 + j\,x_6) - r(n) \end{aligned}$$

- The variables are:
  - $x_1,\ldots,x_6$ are optimization variables.
  - t is a test signal.
  - r is a reference signal.
  - $\kappa = 2\pi T_{\text{symb}}$ is a constant where T_{symb} is the symbol time.
  - N is the number of symbols (can typically be in the range 300-10E6 depending on the application and methodology applied in the given situation).

Case Study: Jacobi and Hessian Computation

The Problem

- To perform optimization we often need the Jacobi vector and the Hessian matrix.
- The Jacobi vector can be determined from:

$$\mathbf{j}(\mathbf{x}) = \left[\frac{\partial f_0(\mathbf{x})}{\partial x_1}, \ldots, \frac{\partial f_0(\mathbf{x})}{\partial x_6}\right]^{\mathrm{T}} \in \mathcal{R}^{6\times 1}$$

- leading to:

$$\mathbf{j}(\mathbf{x}) = 2\sum_{n=1}^{N} \Re\left[e^*(\mathbf{x};n)\,\mathbf{e}_\partial(\mathbf{x};n)\right]$$

- with:

$$\mathbf{e}_\partial(\mathbf{x};n) = \left[\frac{\partial e(\mathbf{x};n)}{\partial x_1}, \ldots, \frac{\partial e(\mathbf{x};n)}{\partial x_6}\right]^{\mathrm{T}}$$

- It can be derived that:

$$
\begin{aligned}
\frac{\partial e(\mathbf{x};n)}{\partial x_1} &= g(\mathbf{x};n) \\
\frac{\partial e(\mathbf{x};n)}{\partial x_2} &= j\, g(\mathbf{x};n) \\
\frac{\partial e(\mathbf{x};n)}{\partial x_3} &= -(n-1)\left\{x_1 + j\, x_2\right\} g(\mathbf{x};n) \\
\frac{\partial e(\mathbf{x};n)}{\partial x_4} &= -j\,\kappa\,(n-1)\left\{x_1 + j\, x_2\right\} g(\mathbf{x};n) \\
\frac{\partial e(\mathbf{x};n)}{\partial x_5} &= -1 \\
\frac{\partial e(\mathbf{x};n)}{\partial x_6} &= -j \\
\\
g(\mathbf{x};n) &= t(n)\, \exp[-(x_3 + j\,\kappa\, x_4)\,(n-1)]
\end{aligned}
$$

- For the Hessian matrix we have:

$$\mathbf{H}(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f_0(\mathbf{x})}{\partial x_1 \, \partial x_1} & \cdots & \frac{\partial^2 f_0(\mathbf{x})}{\partial x_1 \, \partial x_6} \\ \vdots & & \vdots \\ \frac{\partial^2 f_0(\mathbf{x})}{\partial x_6 \, \partial x_1} & \cdots & \frac{\partial^2 f_0(\mathbf{x})}{\partial x_6 \, \partial x_6} \end{bmatrix} \in \mathcal{R}^{6 \times 6}$$

- where it can be derived that:

$$\frac{\partial^2 f_0(\mathbf{x})}{\partial x_p \, \partial x_q} = 2 \sum_{n=1}^{N} \Re \left[e^*(\mathbf{x}; n) \, \frac{\partial^2 e(\mathbf{x}; n)}{\partial x_p \, \partial x_q} \right] + 2 \sum_{n=1}^{N} \Re \left[\frac{\partial e^*(\mathbf{x}; n)}{\partial x_p} \, \frac{\partial e(\mathbf{x}; n)}{\partial x_q} \right], \quad p, q = 1, \ldots, 6$$

- The Hessian can be computed as:

$$\mathbf{H}(\mathbf{x}) \;=\; 2\sum_{n=1}^{N} \Re\left\{e^{*}(\mathbf{x};n)\,\mathbf{E}_{\partial^2}(\mathbf{x};n) \;+\; \mathbf{E}_{\partial\partial}(\mathbf{x};n)\right\}$$

- where the two key matrices are:

$$\mathbf{E}_{\partial^2}(\mathbf{x};n) \;=\; \begin{bmatrix} 0 & 0 & -g_1(\mathbf{x};n) & -j\,\kappa\,g_1(\mathbf{x};n) & 0 & 0 \\ 0 & 0 & -j\,g_1(\mathbf{x};n) & \kappa\,g_1(\mathbf{x};n) & 0 & 0 \\ -g_1(\mathbf{x};n) & -j\,g_1(\mathbf{x};n) & g_2(\mathbf{x};n) & j\,\kappa\,g_2(\mathbf{x};n) & 0 & 0 \\ -j\kappa\,g_1(\mathbf{x};n) & \kappa\,g_1(\mathbf{x};n) & j\,\kappa\,g_2(\mathbf{x};n) & -g_3(\mathbf{x};n) & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

- with:

$$\begin{aligned} g_1(\mathbf{x};n) &= (n-1)\,g(\mathbf{x};n) \\ g_2(\mathbf{x};n) &= \{x_1 + j\,x_2\}\,(n-1)^2\,g(\mathbf{x};n) \\ g_3(\mathbf{x};n) &= \{x_1 + j\,x_2\}\,\kappa^2\,(n-1)^2\,g(\mathbf{x};n) \end{aligned}$$

- The second matrix is given by:

$$\begin{aligned}\mathbf{E}_{\partial\partial}(\mathbf{x};n) &= \begin{bmatrix} \frac{\partial e^*(\mathbf{x};n)}{\partial x_1} & \frac{\partial e(\mathbf{x};n)}{\partial x_1} & \cdots & \frac{\partial e^*(\mathbf{x};n)}{\partial x_1} & \frac{\partial e(\mathbf{x};n)}{\partial x_6} \\ \vdots & & & \vdots & \\ \frac{\partial e^*(\mathbf{x};n)}{\partial x_6} & \frac{\partial e(\mathbf{x};n)}{\partial x_1} & \cdots & \frac{\partial e^*(\mathbf{x};n)}{\partial x_6} & \frac{\partial e(\mathbf{x};n)}{\partial x_6} \end{bmatrix} \\ &= \mathbf{e}^*_{\partial}(\mathbf{x};n)\,\mathbf{e}^{\mathrm{T}}_{\partial}(\mathbf{x};n)\end{aligned}$$

Case Study: Jacobi and Hessian Computation
Implementation #1

- **An initial implementation of the Jacobi vector:**
  - **Step 1:** Compute g.
  - **Step 2:** The partial derivative of the error is formed as a matrix where n is varied in a loop.
  - **Step 3:** The Jacobi vector is formed by summing the elements in a loop.
- **And of the Hessian matrix:**
  - **Step 1:** A 3D-array is formed for the product of derivatives $\mathbf{E}_{\partial\partial}$ by looping over the symbol number $n=1,\ldots,N$.
  - **Step 2:** A 3D-array is formed for the second order derivatives by looping over the symbol number $n=1,\ldots,N$.
  - **Step 3:** Compute the sum of contributions caused by second order derivatives and product of first order derivatives by looping over the symbol number $n=1,\ldots,N$.
- **Comments**:
  - This approach is very close to the equations and the risk of errors is low compared to more fancy implementations.
  - This implementation should always be made – if nothing else then to have something to compare with.
  - In this particular case we should obviously also make a numerical approximation to get the Jacobi vector and Hessian matrix just by having the objective function.

Case Study: Jacobi and Hessian Computation
Implementation #1

- **Version 1 of the code (MATLAB only):**

```
function [ jx, Hx ] = JacHes1( r, t, fsymb, x )
  % Constants
  N = length(r);   kappa = 2*pi/fsymb;

  % Define g-vector function
  g  = exp(1j*x(2))*t.*exp(-(x(3)+1j*kappa*x(4))*(0:N-1).');

  %% JACOBI
  ed = zeros(6,N) + 1j*zeros(6,N);
  for n=1:N
    ed(1,n) = g(n);
    ed(2,n) = 1j*x(1)*g(n);
    ed(3,n) = -(n-1)*x(1)*g(n);
    ed(4,n) = -1j*kappa*x(1)*(n-1)*g(n);
    ed(5,n) = -1;
    ed(6,n) = -1j;
  end

  jx = zeros(6,1);
  [ e, ~ ] = eVector( r, t, fsymb, x );
  for n=1:N,   jx = jx + real(conj(e(n))*ed(:,n));   end
  jx = 2*jx;
```

Case Study: Jacobi and Hessian Computation

Implementation #1

```
  %% HESSIAN
  % Edd matrix part
  Edd = zeros(6,6,N) + 1j*zeros(6,6,N);
  for n=1:N,   Edd(:,:,n) = conj(ed(:,n))*transpose(ed(:,n));   end

  % Ed2 matrix part
  Ed2 = zeros(6,6,N) + 1j*zeros(6,6,N);
  for n=1:N
    Ed2(2,1,n) = 1j*g(n);                       Ed2(3,1,n) = -(n-1)*g(n);
    Ed2(4,1,n) = -1j*kappa*(n-1)*g(n);          Ed2(1,2,n) = 1j*g(n);
    Ed2(2,2,n) = -x(1)*g(n);                    Ed2(3,2,n) = -1j*x(1)*(n-1)*g(n);
    Ed2(4,2,n) = kappa*x(1)*(n-1)*g(n);         Ed2(1,3,n) = -(n-1)*g(n);
    Ed2(2,3,n) = -1j*x(1)*(n-1)*g(n);           Ed2(3,3,n) = x(1)*(n-1)^2*g(n);
    Ed2(4,3,n) = 1j*kappa*x(1)*(n-1)^2*g(n);    Ed2(1,4,n) = -1j*kappa*(n-1)*g(n);
    Ed2(2,4,n) = kappa*x(1)*(n-1)*g(n);         Ed2(3,4,n) = 1j*kappa*x(1)*(n-1)^2*g(n);
    Ed2(4,4,n) = -x(1)*kappa^2*(n-1)^2*g(n);
  end

  % Hessian
  Hx = zeros(6,6);
  for n=1:N,   Hx = Hx + real(conj(e(n))*Ed2(:,:,n) + Edd(:,:,n));   end
  Hx = 2*Hx;
end
```

Case Study: Jacobi and Hessian Computation
Implementation #1

- **The function eVector is given by:**

```
function [ e, f0 ] = eVector( r, t, fsymb, x )
  % Initialization
  N = length(r);
  kappa = 2*pi/fsymb;

  % Compute error vector, e, and objective function, f0
  e = (x(1)+1j*x(2))*t.*exp(-(x(3)+1j*kappa*x(4))*(0:N-1).') ...
      - (x(5)+1j*x(6)) - r;
  f0 = sum(abs(e).^2);
end
```

Case Study: Jacobi and Hessian Computation
Results #1

- **Results for the first version of the code with N=200:**
  - Dual Intel Xeon X5570 with 48 GB memory and NVIDIA Tesla C2070; Ubuntu Linux, Jacket 1.7.1.

```
=== N = 200 ======================
MATLAB1 - Single [s]:   0.01729
Jacket1 - Single [s]:   1.29047
Speedup1 - Single [-]:  0.01340
----------------------------------
MATLAB1 - Double [s]:   0.00224
Jacket1 - Double [s]:   1.19611
Speedup1 - Double [-]:  0.00187
==================================
```

First of all notice the warm up problem of Jacket.

The result is not really impressive – in particular for Jacket. We see some variation in the MATLAB timing since I haven't used a repetition loop here. I also tried with a larger problem (increase N) but this just means waiting a long, long time for Jacket.

We might be tempted to give up!

Case Study: Jacobi and Hessian Computation
Possible Improvements

- So not all that impressive … the time as such for JacHes1 may not be all that bad but we may have to call it many, many times depending on application. So we need to do something. Without ending up with version 2, 3, …, 20, 21 let's condense what we can do:

- **Objectives**:
  - We want the same code to adapt automatically to MATLAB as well as Jacket inputs (reference and test signals).
  - If at all possible we want to avoid loops – loops kill performance. And even gfor is very, very expensive here. Just for the Hessian we have dimension $6 \times 6 \times N$, which for N=5000 and double precision would mean 1.44 MB for holding one array. If we use gfor we need to multiply this with the number of parallel threads meaning that we must multiply with N. This would cost 7.2 GB of memory just for the Hessian 3D array. Then imagine what happens for N=1E6.

Case Study: Jacobi and Hessian Computation
Implementation #2

- **Version 2 of the code (MATLAB and Jacket enabled):**

```
function [ jx, Hx ] = JacHes2( r, t, fsymb, x )
  % Determine common class, and define
  % scalar '0' used in colon expansion
  cls = superiorfloat(r,t);
  zero = zeros(cls);

  %% INITIALIZATION
  % Constants
  N = length(r);
  kappa = 2*pi/fsymb;

  oneV = ones(N,1,cls);
  nV = (zero:N-1)';
  g  = t.*exp(-(x(3)+1j*kappa*x(4))*nV);
  nVg = nV .* g;
  nVSg = nV .* nVg;

  ed = zeros(6,N,cls) + 1j*zeros(6,N,cls);
  Hx = zeros(6,6,cls);
```

Class is inherited from the reference and test input vectors. This means we can use the same code for both MATLAB and Jacket.

We use an advanced colon indexing, which is inherited via the class of the input. This slows down MATLAB slightly but significantly fires up on Jacket

Arrays are pre-allocated and class inherited via the input reference and test vectors.

Case Study: Jacobi and Hessian Computation

Implementation #2

```
  %% JACOBI VECTOR
  ed(1,:) = g;
  ed(2,:) = 1j*g;
  ed(3,:) = -(x(1)+1j*x(2))*nVg;
  ed(4,:) = -1j*kappa*(x(1)+1j*x(2))*nVg;
  ed(5,:) = -oneV;
  ed(6,:) = -1j*oneV;
  e  = (x(1)+1j*x(2))*t.*exp(-(x(3)+1j*kappa*x(4))*nV) ...
       - (x(5)+1j*x(6)) - r;
  jx = 2*sum(real(repmat(ctranspose(e),6,1) .* ed),2);

  %% HESSIAN MATRIX
  nVge  = (nVg .* conj(e)).';
  nVSge = ((x(1)+1j*x(2))*nVSg .* conj(e)).';
  ced = conj(ed);

  Hx(1,1) = 2*sum(real(ced(1,:).*ed(1,:)));
  Hx(2,1) = 2*sum(real(ced(2,:).*ed(1,:)));
  Hx(3,1) = 2*sum(real(-nVge+ced(3,:).*ed(1,:)));
  Hx(4,1) = 2*sum(real(-1j*kappa*nVge+ced(4,:).*ed(1,:)));
  …
end
```

Jacobi: Avoids using loops.

Hessian: Also this avoids loops.

We use an advanced colon indexing, which is inherited via the class of the input. This slows down MATLAB slightly but significantly fires up on Jacket.

Arrays are pre-allocated and class inherited via the input reference and test vectors.

Case Study: Jacobi and Hessian Computation
Results #2

- **Results master_JacHes2b:**
  - Colfax GPU HPC: dual Intel Xeon X5570 with 48 GB memory and NVIDIA Tesla C2070; Ubuntu Linux, Jacket 1.7.1.

```
=== N = 5000 ====================
MATLAB2 - Single [s]:     0.00336
Jacket1 - Single [s]:    32.47931
Jacket2 - Single [s]:     0.01425
Speedup1 - Single [-]:    0.00010
Speedup2 - Single [-]:    0.23566
--------------------------------
MATLAB2 - Double [s]:     0.00327
Jacket1 - Double [s]:    30.18145
Jacket2 - Double [s]:     0.01372
Speedup1 - Double [-]:    0.00011
Speedup2 - Double [-]:    0.23831

=== N = 5000 ====================
MATLAB2 - Single [s]:     0.00337
Jacket1 - Single [s]:    32.47139
Jacket2 - Single [s]:     0.01427
Speedup1 - Single [-]:    0.23596
--------------------------------
MATLAB2 - Double [s]:     0.00327
Jacket1 - Double [s]:    30.18021
Jacket2 - Double [s]:     0.01373
Speedup1 - Double [-]:    0.00011
Speedup2 - Double [-]:    0.23832
```

Results from **master_JacHes2a**:

For this N value the version 2 of Jacket is close to 2,000 times faster than version 1. It just gets higher the larger N is.

For this hardware platform, break even between the dual Xeon X5570 CPUs and one C2070 is around N=25E3. Normally, we can't expect to have 2 CPUs available.

When considering Jacket, version 2 is more than 2000 times faster than version 1 when N=5000. For larger N the difference is even bigger.

The **upper results is obtained with maxNumCompThreads(16)** and the **one to the left is obtained with maxNumCompThreads(4)**. The result is basically the same indicating that MATLAB does not use the dual CPU capability.

Case Study: Jacobi and Hessian Computation
Results #2

- **Results master_JacHes2b:**
  - Colfax GPU HPC: dual Intel Xeon X5570 with 48 GB memory and NVIDIA Tesla C2070; Ubuntu Linux, Jacket 1.7.1.

```
=== N = 200E3 ====================
MATLAB2 - Single [s]:     0.12240
Jacket2 - Single [s]:     0.02140
Speedup2 - Single [-]:    5.72005
---------------------------------
MATLAB2 - Double [s]:     0.19489
Jacket2 - Double [s]:     0.02695
Speedup2 - Double [-]:    7.23084
=================================


=== N = 10E6 =====================
MATLAB2 - Single [s]:     7.43233
Jacket2 - Single [s]:     0.44356
Speedup2 - Single [-]:   16.75617
---------------------------------
MATLAB2 - Double [s]:    11.39839
Jacket2 - Double [s]:     0.73921
Speedup2 - Double [-]:   15.41967
=================================
>>
```

Results from **master_JacHes2b**:

First of all observe that Jacket version 1 has been omitted for the large N values shown here. The reason being that the execution time is extremely high.

As seen from the results, Jacket is substantially faster than MATLAB. And this increases with N.

Break even between MATLAB and Jacket happens around N=25E3 but that obviously depends on the available hardware platform.