Parallel High Performance Computing

With Emphasis on Jacket Based GPU Computing

Programming Multiple GPUs

Torben Larsen
Aalborg University

Outline

SPMD:
Single Program Multiple Data

SPMD: Single Program Multiple Data
Introduction

- The spmd construct in MATLAB is a flexible structure to execute computations in parallel. There are a number of special special things to observe.

- First of all it is necessary to open a matlabpool – matlabpool opens a number of workers, which can do tasks independently (but also transfer data among workers if need).

```
matlabpool config #workers
```

- On small workstations with up to 4 GPUs it generally suffices to use the "local" configurations. On larger cluster the administrative staff normally have defined some configurations. The config gives info on max. number of workers, data location, …
- Example:

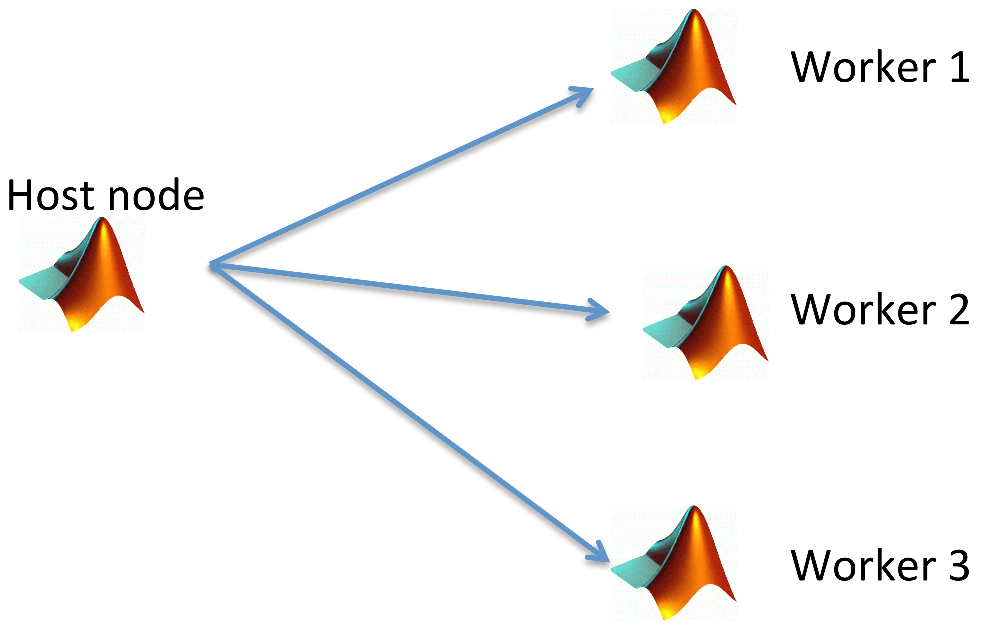
```
>> matlabpool local 2
Starting matlabpool using the 'local' configuration ... connected to 2 labs.
>>
```

- So this opens two "labs" in the local configuration.

SPMD: Single Program Multiple Data
Introduction

- **As an example we might have the following situation with 3 workers:**

```
>> matlabpool('local','3')
      Starting matlabpool using the 'local' configuration ... connected to 3 labs.
>>
```



SPMD: Single Program Multiple Data
Example

- Let’s have a look at a simple example, which clearly illustrates what happens in terms of transferring data between host and workers.
- It is assumed that a matlabpool has been defined by e.g. “**matlabpool(‘local’,’2’);**”.

- The code running on the host node is:

```
X = randn(N,N);
spmd;
  Y = X + labindex;
  LBIDX = labindex;
  Sgpu = sum(sum(gsingle(X)));
end

for jj=1:matlabpool('size')
  LBindxNo(jj) = LBIDX{jj};
  SUMgpu(ii) = Sgpu{jj};
end
```

SPMD: Single Program Multiple Data
Example

- **So what happens …?**
  - First X is defined on the host.
  - Second, when SPMD is met the tasks are distributed to all workers … but no transfer of data or anything
  - But this happens when “Y=X+labindex;” is met. Now X is copied to all workers. It is not guaranteed from MATLAB that it takes it in order worker1,worker2, … It is my experience that this is the way it is done though. Obviously it is very inefficient to transfer the same data across the network #Workers times. Rumors say that CUDA Toolkit 4.0 may change this.
  - “LBIDX=labindex” saves the number of the worker in the LABIDX variable.
  - “Agpu=sum(sum(gsingle(X)));” casts X to the GPU, computes the sum of all matrix elements and stores that number in the “Agpu” variable.
  - When the “end” corresponding to the “SPMD” is met it is very important to note that there is no automatic transfer of data back from the workers to the host node. The transfer first happens when the head node needs the data for some reason (to display the value or whatever).

  - The “for jj=…” loop requests data from the workers – this is done by requesting data from the composite data object as in “LBIDX{jj}” and “Agpu{jj}”. Here, “jj” refers to the worker number.
  - So when “LBindxNo(jj)=LBIDX{jj}” is met in the code, the host node requests transfer of the array LBIDX from worker “jj”. Similar happens for “SUMgpu(jj)”.

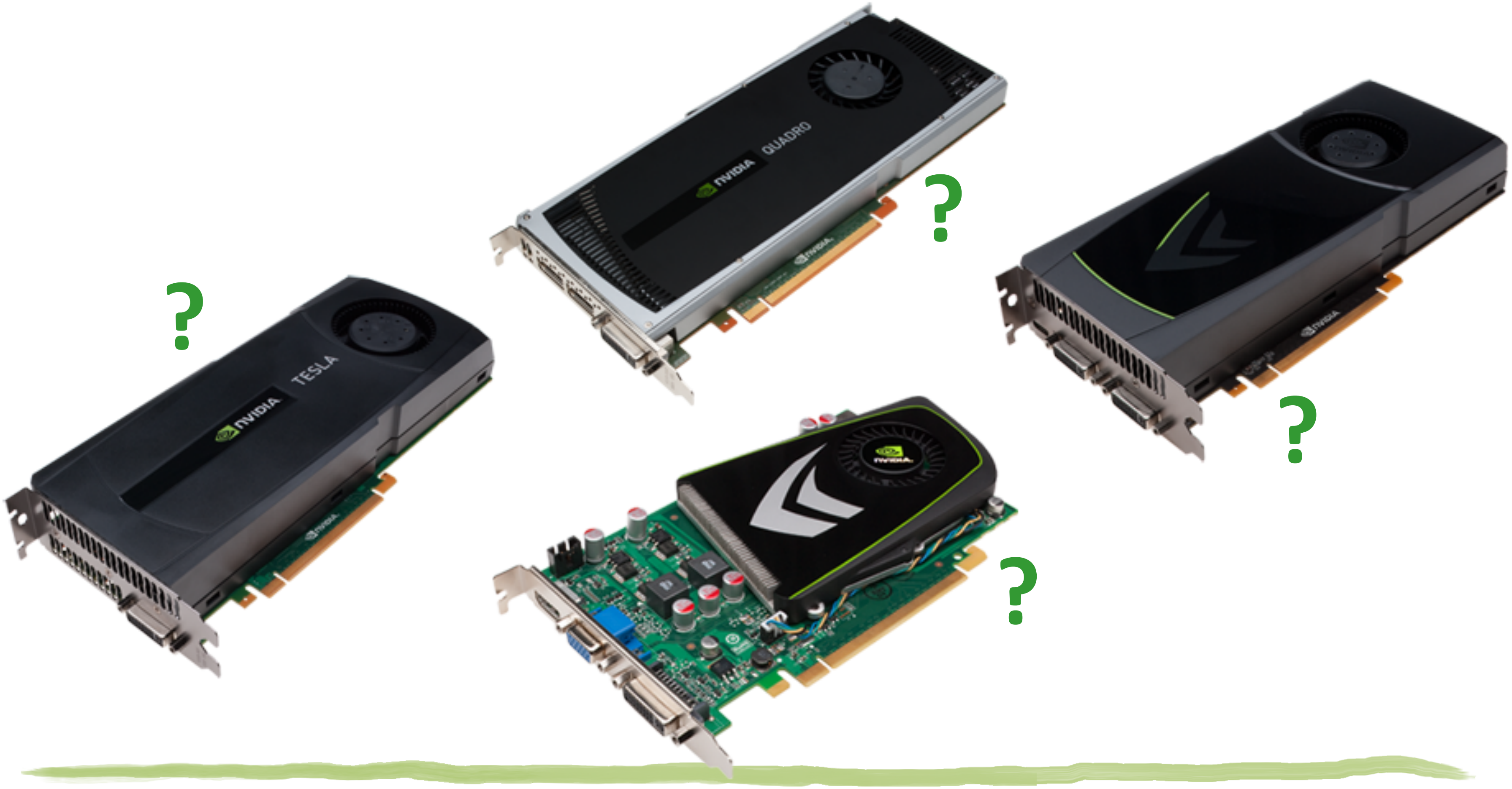
  - Again it is important to note that “jj” is not necessarily identical to the “labindex” number.

SPMD: Single Program Multiple Data
Overview

- **SPMD characteristics:**
  - It is very flexible and can be used to distribute just about anything between workers. PARFOR is restricted to loops alone – and loops behaving is a particularly "nice" way.

- **PARFOR** is easy to use though – it works by simply distributing the next iterator value to the next free resource. This is simple – but robust and generally works fine. SPMD is a more flexible way to distribute tasks and can also be used for hybrid CPU and GPU cases.

Selecting Specific GPUs



Selecting Specific GPUs

- **With the workers defined how do we select GPUs for the different tasks?**

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

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

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

- Important things to notice:
  - There are 4 GPUs available: GPU0,…,GPU3.
  - License wise we have access to 4 simultaneous working GPUs.
  - To use multiple GPUs here we need MATLAB PCT (Parallel Computing Toolbox).

Selecting Specific GPUs

```
>> matlabpool('close','force'); matlabpool('open','local',2)
>> spmd; if labindex==1, gselect(1); elseif labindex==2, gselect(3); end; ginfo; end
Lab 2:
  Jacket v1.7.1 (build 58de35b) by AccelerEyes
  Detected CUDA-capable GPUs:
  CUDA driver 263.06, CUDA toolkit 3.2
  GPU0 Quadro 2000, 1251 MHz, 994 MB VRAM, Compute 2.1 (single,double)
  GPU1 Quadro 4000, 810 MHz, 2015 MB VRAM, Compute 2.0 (single,double)
  GPU2 Quadro 4000, 810 MHz, 2015 MB VRAM, Compute 2.0 (single,double)
  GPU3 Quadro 4000, 810 MHz, 2015 MB VRAM, Compute 2.0 (single,double) (in use)
  Display Device: GPU3 Quadro 4000
  GPU Memory Usage: 1977 MB free (2015 MB total)
Lab 1:
  Jacket v1.7.1 (build 58de35b) by AccelerEyes
  Detected CUDA-capable GPUs:
  CUDA driver 263.06, CUDA toolkit 3.2
  GPU0 Quadro 2000, 1251 MHz, 994 MB VRAM, Compute 2.1 (single,double)
  GPU1 Quadro 4000, 810 MHz, 2015 MB VRAM, Compute 2.0 (single,double) (in use)
  GPU2 Quadro 4000, 810 MHz, 2015 MB VRAM, Compute 2.0 (single,double)
  GPU3 Quadro 4000, 810 MHz, 2015 MB VRAM, Compute 2.0 (single,double)
  Display Device: GPU3 Quadro 4000
  GPU Memory Usage: 1977 MB free (2015 MB total)
```

Use labindex to control what GPUs to use in the computation.

When SPMD is used the GPU selection can be done even if you have performed Jacket computations in the MATLAB workspace. But NOT in the SPMD construct itself. Results from a Colfax CXT2000i.

Example: Simultaneous Multi-GPU GFLOPS Measurement

Example: Simultaneous Multi-GPU GFLOPS Measurement

Problem

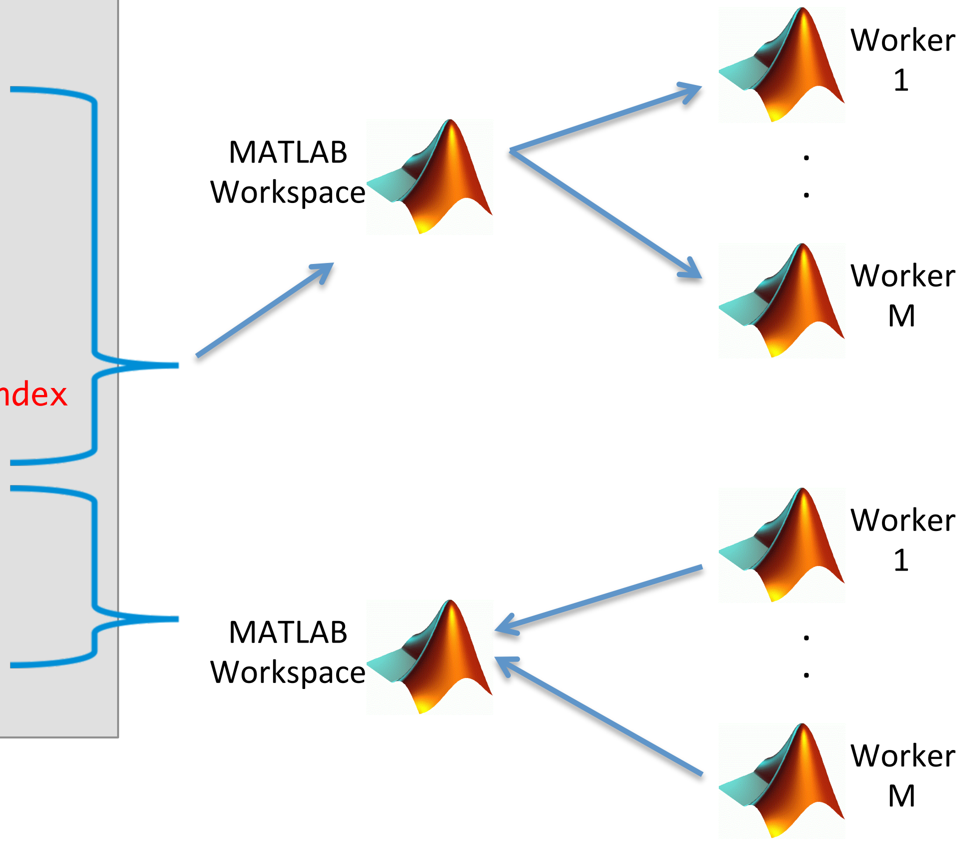
- This example demonstrates how we can use an SPMD construct to measure the floating point performance of multiple GPUs at the same time.
- We create a number of MATLAB workers by use of MATLAB PCT (Parallel Computing Toolbox).
- Each worker can have access to one GPU – it can also just be used for CPU computing if desired.
- We push the measurement onto all workers/GPUs – note that we FORTUNATELY don't need a worker for each core of the GPU. Just one core per GPU. This is a favorable situation for GPUs compared to CPUs.

Example: Simultaneous Multi-GPU GFLOPS Measurement

Implementation

- The purpose of the example is, by use of SPMD, to measure the floating point performance of M GPUs simultaneously.
- The code is roughly:

```
% Define matrix size etc.

% Perform computations
spmd
    % Define matrices
    % Warm-up Jacket

    for jj=1:Repetitions
        % Compute flop count
    end

    % Compute GFLOPS, store labindex
end

% Fetch data from all workers
for jj=1:M
    % Fetch data from workers
end

% Save data
```



Example: Simultaneous Multi-GPU GFLOPS Measurement
Implementation

- Let's take the actual code part-by-part:

```
%% OPEN MATLABPOOL
matlabpool('close','force');
matlabpool(GPUDEF,'open');
```

- The important thing here is that:
  - We open the matlabpool with the correct definition – mainly meaning that we get the expected number of workers.

– Depending on what has been done on the GPUs earlier it may be an advantage to do garbage collection:

```
spmd;
  clear gpu_hook;
end
```

- Jacket by itself links workers to GPUs – we can change it by GSELECT() if we want but it is not necessary.

Example: Simultaneous Multi-GPU GFLOPS Measurement

Implementation

- Next we set up the SPMD construct:

```
%% PERFORM COMPUTATIONS
spmd
  % Set matrix size and repetition
  N = 1000;   RPT = 100;

  % Seed the PRNG
  grandn('state', 3205438543);

  % Generate the matrices
  A = grandn(N,N,'single');
  B = grandn(N,N,'single');

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

  % Perform computation
  gsync; ts = tic;
  for jj=1:RPT
    R = A*B; geval(R);
  end
  gsync; te = toc(ts)/RPT;

  % FLOPS computation and index number
  GFLOPS = N^2*(2*N-1)/(te*1E9);
  LABindx = labindex;
end
```

- **Breaking this down in parts:**
  - The parts needed to be computed simultaneously are put inside the spmd-end construct.
  - First, the PRNG is initialized to ensure identical conditions across all workers – often the runtime depends on the specific data.
  - Jacket is warmed up by running the multiplication twice. “GEVAL” is crucial to force computation of R.
  - The floating point performance is measured RPT times, and afterwards the average GFLOPS count is determined. Observe the use of GEVAL and GSYNC.
  - The labindex is returned to know what worker the performance is located to.
- Once all workers have completed their jobs, the execution exits from the SPMD-END construct and returns to the MATLAB code.

Example: Simultaneous Multi-GPU GFLOPS Measurement
Implementation

- Finally, we pull back the needed results from the workers and save the result to file:

```
%% RETURN RESULTS TO THE MATLAB WORKSPACE
% Preallocate vectors
Meas_gflops = zeros(matlabpool('size'),1);
labindx = zeros(matlabpool('size'),1);

% Pull back data
for jj=1:matlabpool('size')
  Meas_gflops(jj) = GFLOPS{jj};
  labindx(jj) = MABindx{jj};
  fprintf('GPU%s:    %10.4f\n', num2str(labindx(jj)), Meas_gflops(jj));
end

% Save measured results
save('meas_flops.mat','Meas_gflops','labindx');
```

- The result vectors in the MATLAB workspace named “Meas_gflops” and “labindx” are pre-allocated.
- Following that, the results are pulled back from the workers to the MATLAB workspace
  - This is done by commands of the form: Workspace_var(ii)=Worker_var{ii}.
- And finally the results are saved to file.

Example: Simultaneous Multi-GPU GFLOPS Measurement
Results

- **Results:**
  - **Colfax CXT2000i;** Core i7-970 with 24 GB memory, **1 x Quadro 2000** **&** **3 x Quadro 4000**

```
>> ex_gflops
Sending a stop signal to all the labs ... stopped.
Did not find any pre-existing parallel jobs created by ... matlabpool.

Starting matlabpool using the 'local' configuration ...
connected to 4 labs.

GPU1:        133.0960     % Quadro 2000
GPU2:        206.3843     % Quadro 4000
GPU3:        206.1575     % Quadro 4000
GPU4:        206.3544     % Quadro 4000
```

Case Study: Multiple GPU Computations

Case Study: Multiple GPU Computations
Problem

- We have a computational task which is well suited to parallel execution across multiple GPUs
- The challenge is that the GPU resources are heterogeneous – we focus on problems where the difference in execution time is the only issue we need to consider (memory is sufficient no matter what GPU we use)

Case Study: Multiple GPU Computations
Analysis

- **In the following we will analyze a typical class of parallel problem where the matrix A is given by:**

$$\mathbf{A} = [\mathbf{a}_1, \ldots, \mathbf{a}_M] \in \mathcal{R}^{N \times M}, \quad \mathbf{a}_m = \begin{bmatrix} a_{1,m} \\ \vdots \\ a_{N,m} \end{bmatrix} \in \mathcal{R}^{N \times 1}$$

- The function $f(\cdot)$ performs computations on the matrix like:

$$y = f(\mathbf{A}) = g([h(\mathbf{a}_1), \ldots, h(\mathbf{a}_M)])$$

The function $h()$ process the $\mathbf{A}$ matrix column by column. This can be done in parallel.

- The function $g(\cdot)$ then combines the sub-results into the final result y.

Case Study: Multiple GPU Computations
Analysis

- Let's say we have a simple problem where the execution time depends linearly with problem size. In this case they use the following time to process m columns:

$$\begin{aligned} t_0(m) &= \gamma_0 \cdot m \quad \rightarrow \quad \gamma_0 = \frac{t_0(M)}{M} \\ t_1(m) &= \gamma_1 \cdot m \quad \rightarrow \quad \gamma_1 = \frac{t_1(M)}{M} \end{aligned}$$

- This means that the constants can be measured by for example see how much time each uses to compute the complete problem – or if that is too much just some part of it:

- So given we have γ_0 and γ_1 how should we distribute the load between the GPUs?
- If we give the fraction δ_0 to GPU0 then the time used by the two GPUs to solve the computational problem is:

$$t = \max\{\gamma_0 \cdot \delta_0 \cdot M; \gamma_1 \cdot (1-\delta_0) \cdot M\}$$

- So the optimum is when the two GPUs are done with each their tasks after precisely the same time.

Case Study: Multiple GPU Computations
Analysis

- This means we should select δ_0 based on:

$$\gamma_0 \cdot \delta_0 \;=\; \gamma_1 \cdot (1 - \delta_0)$$

- Thereby the loading of GPU0 δ_0 should be:

$$\delta_0 \;=\; \frac{\gamma_1}{\gamma_0 + \gamma_1}$$

- Or in terms of relative strength between the GPUs:

$$\delta_0 \;=\; \frac{\mu}{\mu + 1} \quad \text{where} \quad \mu = \frac{\gamma_1}{\gamma_0}$$

- If for example we have two identical GPUs we then have μ=1 and then δ_0=0.5 as expected.

Case Study: Multiple GPU Computations
Implementation

- An example MATLAB code is given here:

```
function [R] = multiGPU(COLS,GPU0,GPU1,LAB0LoadPct)
  %% SET UP VARIABLES
  eLOW  = round(LAB0LoadPct/100*COLS);

  %% PERFORM COMPUTATIONS
  spmd;
    % Initialize PRNG and create A, k1 and k2
    grandn('state', 17834534);
    A = grandn(2^17,COLS,'double');
    k1 = pi/4;
    k2 = pi/5;

    if labindex==GPU0
      R = zeros(1,COLS,class(A));
      for e=1:eLOW
        R(e) = computefun(A(:,e), k1, k2);
      end
    elseif labindex==GPU1
      R = zeros(1,COLS,class(A));
      for e=eLOW+1:COLS
        R(e) = computefun(A(:,e), k1, k2);
      end
    end
  end
end
```

The function process columns in parallel
– so the function can be executed in parallel as slices of a matrix.

Case Study: Multiple GPU Computations
Implementation

- With the sub-function doing the computations:

```
function [ R ] = computefun( a, k1, k2 )
  Rt = zeros(1,25,class(a));
  for rpt=1:25
    Rt(rpt) = (k1*k2^2*abs(fft(a)).^2/(length(a)^2)).' ...
              *ones(length(a),1,class(a));
  end
  R = Rt * ones(25,1,class(a));
end
```

- In the following we assume the workers are defined by for example:

```
matlabpool('close','force');
matlabpool('open','local?,2);

% Clear GPU memory
spmd; clear gpu_hook; end
```

Case Study: Multiple GPU Computations
Implementation

- The master file to determine the performance constants is:

```
% Constants
COLS = 200; % Number of columns
GPU0 = 2;   % First GPU number (numbering as labindex; >0)
GPU1 = 3;   % Second GPU number (numbering as labindex; >0)

% GPU0 performance constants
[R] =  multiGPU( COLS, GPU0, GPU1, 100 ); geval(R);
gsync; ts=tic;
[R] =  multiGPU( COLS, GPU0, GPU1, 100 ); geval(R);
gsync; gamma0=toc(ts)/COLS

% GPUy performance constants
[R] =  multiGPU( COLS, GPU1, GPU0, 100 ); geval(R);
gsync; ts=tic;
[R] =  multiGPU( COLS, GPU1, GPU0, 100 ); geval(R);
gsync; gamma1=toc(ts)/COLS

% Estimated optimum GPU0 loading
delta0 = gamma1/(gamma0 + gamma1)
```

Case Study: Multiple GPU Computations
Implementation

- The master function, which is sweeping the loading of the first GPU (GPUx), is then:

```
% Constants
COLS = 200; % Number of columns
GPUx = 1;   % First GPU number (numbering as labindex; >0)
GPUy = 2;   % Second GPU number (numbering as labindex; >0)

% Perform sweep of GPUx load percentage
te = zeros(101,1);
for ii=0:100
  [R] =  multiGPU( COLS, GPUx, GPUy, ii ); geval(R);
  gsync; ts=tic;
  [R] =  multiGPU( COLS, GPUx, GPUy, ii ); geval(R);
  gsync; te(ii+1)=toc(ts);
  fprintf('Load percentage of GPUx:   %3.0f\n',ii);
end

% Plot results
figure(1); clf;
plot( (0:100)', te, 'g', 'LineWidth', 1.5 );
xlabel('Load Percentage {Quadro-2000 / Quadro-4000}   [%]');
ylabel('Multi-GPU Execution Time   [s]');
title('Multi-GPU Execution Time vs. Load Percentage');
grid;
print('-djpeg99', 'Q2000-Q4000.jpg');
```

Case Study: Multiple GPU Computations
Results

- **Results:**
  - **Colfax CXT2000i; Intel Core i7-970 with 24 GB memory; GPU0: NVIDIA Quadro 4000 + GPU1: Quadro 4000; MATLAB R2011a; Jacket 1.7.1.**

```
>> master_perfconsts   % Double precision

gamma0 = 0.0494
gamma1 = 0.0495

delta0 = 0.5005
```

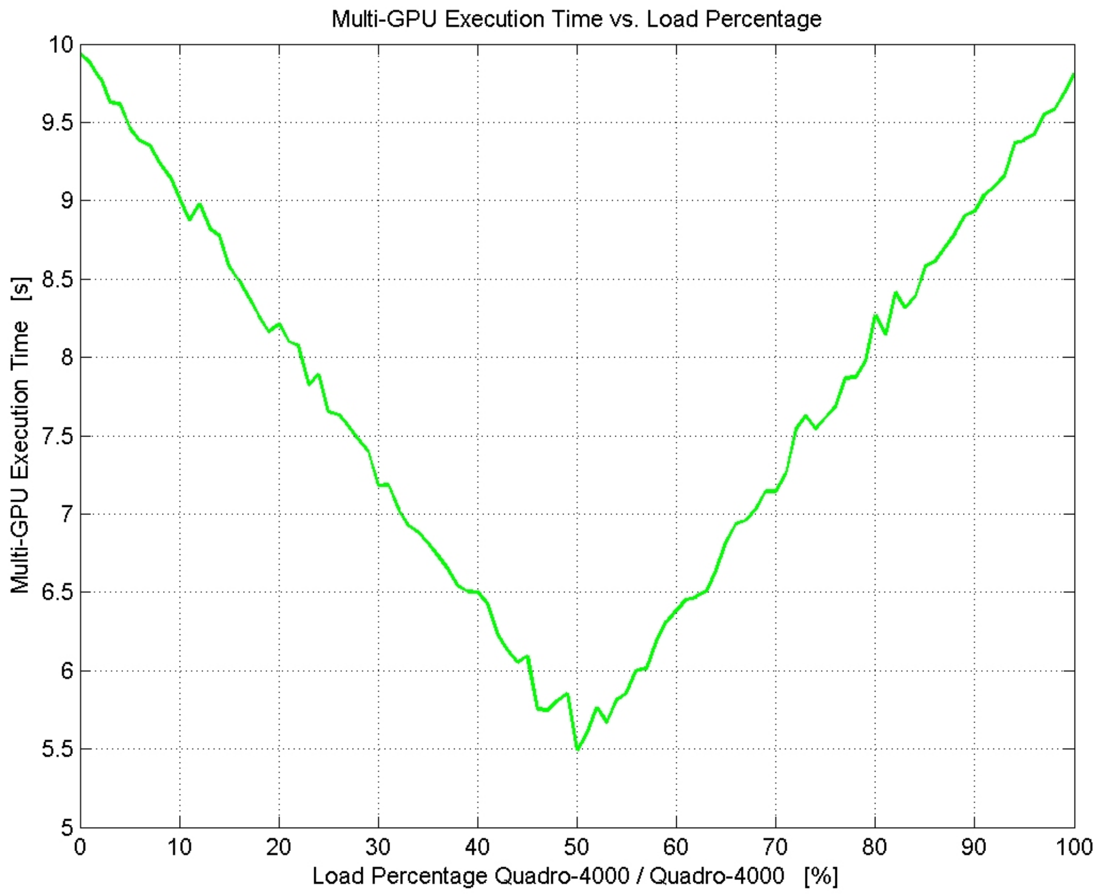
The difference between the two γ values is insignificant as expected. The warm-up before the timing is essential for this. Total time needed for one GPU alone is 9.9 seconds.

As we can see to the right, a direct measurement of the function with different loadings confirm that approximately 50% is indeed the best choice.

```
>> gsync; tic; R=multiGPU( 200, 2, 3, 49 ); geval(R); gsync; toc
Elapsed time is 5.691489 seconds.
>> gsync; tic; R=multiGPU( 200, 2, 3, 50 ); geval(R); gsync; toc
Elapsed time is 5.556264 seconds.
>> gsync; tic; R=multiGPU( 200, 2, 3, 51 ); geval(R); gsync; toc
Elapsed time is 5.558510 seconds.
>> gsync; tic; R=multiGPU( 200, 2, 3, 52 ); geval(R); gsync; toc
Elapsed time is 5.667506 seconds.
```

Case Study: Multiple GPU Computations
Results

- **Results:**
  - **Colfax CXT2000i; Intel Core i7-970 with 24 GB memory; GPU0: NVIDIA Quadro 4000 + GPU1: Quadro 4000; MATLAB R2011a; Jacket 1.7.1.**



The figure shows the execution time when using GPU0 and GPU1 on the same task versus the loading of GPU0. Since the GPUs are identical in type, we would expect a loading percentage of GPU0 to be 50% for optimum performance. This is also very close to being the case as seen from the figure.

There is a bit of "noise" on the figure, which probably can be reduced by introducing averaging. Due to simulation time this has not been implemented here.

Case Study: Multiple GPU Computations
Results

- **Results:**
  - **Colfax CXT2000i; Intel Core i7-970 with 24 GB memory; GPU0: NVIDIA Quadro 2000 + GPU1: Quadro 4000; MATLAB R2011a; Jacket 1.7.1.**

```
>> master_perfconsts   % Double precision

gamma0 = 0.0673
gamma1 = 0.0500

delta0 = 0.4261
```

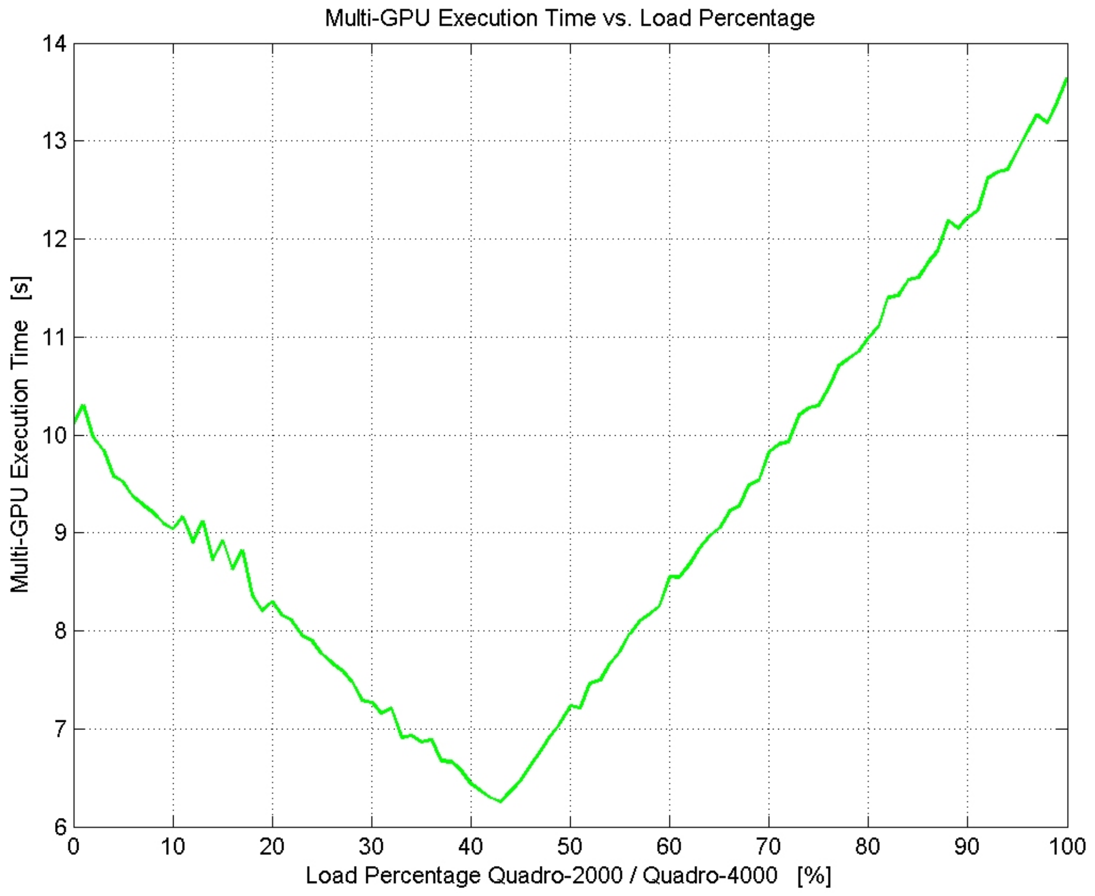
The difference between the two γ values is now quite clear.

The predicted optimum loading of GPU0 (the Quadro 2000) is very close to what is also measured. A loading of 42-43 % is where we have the minimum runtime.

```
>> gsync; tic; R=multiGPU( 200, 1, 2, 40 ); geval(R); gsync; toc
Elapsed time is 6.397907 seconds.
>> gsync; tic; R=multiGPU( 200, 1, 2, 41 ); geval(R); gsync; toc
Elapsed time is 6.323015 seconds.
>> gsync; tic; R=multiGPU( 200, 1, 2, 42 ); geval(R); gsync; toc
Elapsed time is 6.208712 seconds.
>> gsync; tic; R=multiGPU( 200, 1, 2, 43 ); geval(R); gsync; toc
Elapsed time is 6.238109 seconds.
>> gsync; tic; R=multiGPU( 200, 1, 2, 44 ); geval(R); gsync; toc
Elapsed time is 6.373369 seconds.
```

Case Study: Multiple GPU Computations
Results

- **Results:**
  - **Colfax CXT2000i; Intel Core i7-970 with 24 GB memory; GPU0: NVIDIA Quadro 2000 + GPU1: Quadro 4000; MATLAB R2011a; Jacket 1.7.1.**



The figure shows the execution time when using GPU0 and GPU1 on the same task versus the loading of GPU0. The GPUs are different here and the loading seems to be optimum at around 42-43 % loading of GPU0 – as expected.

There is a bit of "noise" on the figure, which probably can be reduced by introducing averaging. Due to simulation time this has not been implemented here.