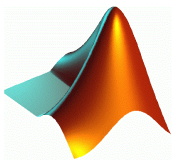
Parallel High Performance Computing

With Emphasis on Jacket Based GPU Computing

Benchmarking

Torben Larsen
Aalborg University



jacket

Outline

- **Benchmarking procedures**
  - Overview
  - Core principles
  - Floating point
  - Memory Transfer
  - Functions
  - Disk access
  - Toolboxes

Overview

Benchmarking Procedures
Overview

- **Objective**:
  - Measure the time it takes to run a certain function. Indicate speedup for Jacket relative to MATLAB

- **Requirements:**
  - **Results must be correct**; measure what we actually want to measure
  - **Reproducible**; we should be able to get the same result several times if the conditions are the same
  - **Easy to use**; if it is easy to use we are more likely to use it

XCORR

PSD

FFT

RANDN

MULTIPLY

SVD

LOAD

Benchmarking Procedures
Overview

- **In the end what we want to know is what speedup we can achieve for our given end-2-end application**
- However, this kind of benchmarking is only relevant to the one given problem and we can't derive any generic information on where the strength of Jacket is

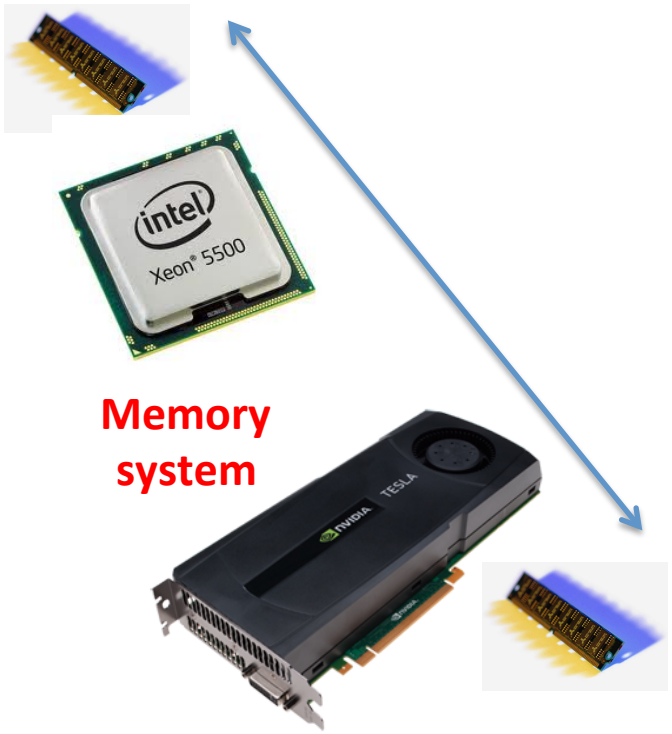
Generic approach >> Functional benchmarking

- Be aware of dependencies:
  - Hardware.
  - Installed software (MATLAB / Jacket / OS / … version).
  - Available hardware (CPU / GPU / motherboard / …).
  - Single vs. double precision / 32 vs. 64 bit integers / …
  - Size of input arrays.
  - …

Be as specific as possible regarding conditions

Benchmarking Procedures
Overview

- **What to benchmark?**



Memory system

Functions

SVD
XCORR
LOAD
FFT
PSD
MULTIPLY
RANDN

1.22248569373
3.24393650673
3.39473085359
9.58363856284
5.68364967395
2.45846496739
7.56739365926

Floating point performance

Benchmarking Procedures
Overview

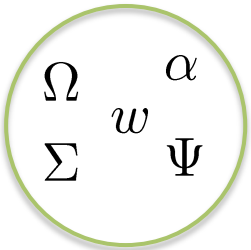
- **Things we must be aware of …**

Synchronize CPU/GPU

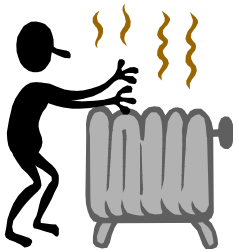
High priority process

Force evaluation of variables

Measurement time > T



Cast variables



Warm-up CPU/GPU

Benchmarking Procedures
Overview

- **Synchronize:** "`gsync`" – ensures synchronization between CPU and GPU
- **Measurement time:** Use "`while-end`" loop to ensure that we measure over sufficient time
- **High priority process:**
  - **Windows:** 1: Use "task manager" to set the priority of the MATLAB process to high. 2: Use maximum performance power plan – ensure full speed on every item.
  - **Mac**: 1: Start "<<MATLAB>>" and perform "sudo renice –n -20 PROCESSID" to give high priority to the MATLAB process. 2: Use maximum performance power plan – ensure full speed on every item.
  - **Linux**: Use "sudo nice –n -20 <<MATLAB>>" to give high priority to the MATLAB process – if possible (perhaps you need to convince the sysadm). 2: Use maximum performance power plan – ensure full speed on every item.
- **Cast variables: Ensure you transfer variables correctly:**
  - To GPU: "`gsingle`", "`gdouble`", "`gint32`", …
  - To CPU: "`single`", "`double`", "`int32`", …
- **Force evaluation of variables:**
  - Use "`geval(VAR1,VAR2,…)`" to ensure variables are actually evaluated. Otherwise nasty surprises
  - In loop ensure you change the content of what you move. Multiple identical for-end content is only evaluated once in many cases in Jacket
- **Warm-up CPU/GPU:**
  - Warm-up to yield reproducible results
  - The optimum is to run precisely the same as you later want to time

- **Some characteristics of selected CPUs and and GPUs:**

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 C2050	448	3072	144.0	238	2.0

Core Principles

Benchmarking Procedures
Core Principles

MATLAB
Input;
FUN: Function handle
TMIN: Min. measurement time

Default measurement time

Initial warm-up and estimate number of repetitions

Re-estimate number of repetitions and do full warm-up

Perform timing; while loop ensures we meet the minimum measurement time

Compensate for plain repetition loop time and compute execution time

```
function [ TC, TELAPSC, RPTC ] = timefun( FUN, TMIN )
  % 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);

  % 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
```

Benchmarking Procedures
Core Principles

Jacket
Input;
FUN: Function handle
TMIN: Min. measurement time

Default measurement time

Initial warm-up and estimate number of repetitions

Re-estimate number of repetitions and do full warm-up

Perform timing; while loop ensures we meet the minimum measurement time

Compensate for plain repetition loop time and compute execution time

```
function [ TG, TELAPSG, RPTG ] = gtimefun( FUN, TMIN )
  % 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);

  % 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
```

Floating Point

232451.22248569373
656463.24393650673
345425.39473085359
995453.58363856284
514286.68364967395
218695.45846496739
753708.56739365926
835734.23874312937
348753.12849327503
434985.38754312493
234812.32435941347

\+
*
/
-

Benchmarking Procedures
Floating Point

- Numerical linear algebra (LINPACK) – also used by Top-500:
  - Jack J. Dongarra, **Performance of Various Computers Using Standard Linear Equations Software**. Technical Report CS–89–85, 2010. University of Tennessee, Oak Ridge National Laboratory, and University of Manchester.
  - www.top500.org.
- GeMM (Generalized Matrix Multiply Technique) which is typically the one used for GPUs.
  - V. Volkov and J. W. Demmel, Benchmarking GPUs to Tune Dense Linear Algebra. **SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing**, pp. 1-11, Austin, Texas, USA, 2008.
  - Torben Larsen, Gallagher Pryor, and James Malcolm: **"Jacket: GPU Powered MATLAB Acceleration".** In Wen-Mei Hwu (editor): "GPU Computing Gems", Jade Edition, Morgan-Kaufmann, July 2011.
- Specialized linear algebra techniques (QR factorization, SVD etc.):
  - Stanimire Tomov, Michael McGuigan, Robert Bennett, Gordon Smith, and John Spiletic: "Benchmarking and implementation of probability-based simulations on programmable graphics cards". **Computers & Graphics**, vol. 29, no. 1, pp. 71–80, 2005.
  - Nico Galoppo, Naga K. Govindaraju, Michael Henson, and Dinesh Manocha: "LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware". **SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing**, 2005, pp. 3–14.
  - Jeff Bolz, Ian Farmer, Eitan Grinspun, Peter Schröder: "Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid". **SIGGRAPH '03: ACM SIGGRAPH 2003 Papers**, pp. 917–924, San Diego, California, USA.

- **The GeMM computation:**

$$\mathbf{R} := \alpha\,\mathbf{A}\,\mathbf{B} \;+\; \beta\,\mathbf{R}$$

- where typically:

$$\alpha, \beta \in \mathcal{R};\; \mathbf{A}, \mathbf{B}, \mathbf{R} \in \mathcal{R}^{N \times N}$$

- **Number of floating point (multiply or add) operations:**

$$\mathbf{R} \quad := \quad \alpha\,\mathbf{A}\,\mathbf{B} \quad + \quad \beta\,\mathbf{R}$$

N^2 mult

$N^2 N$ mult
$N^2(N-1)$ add

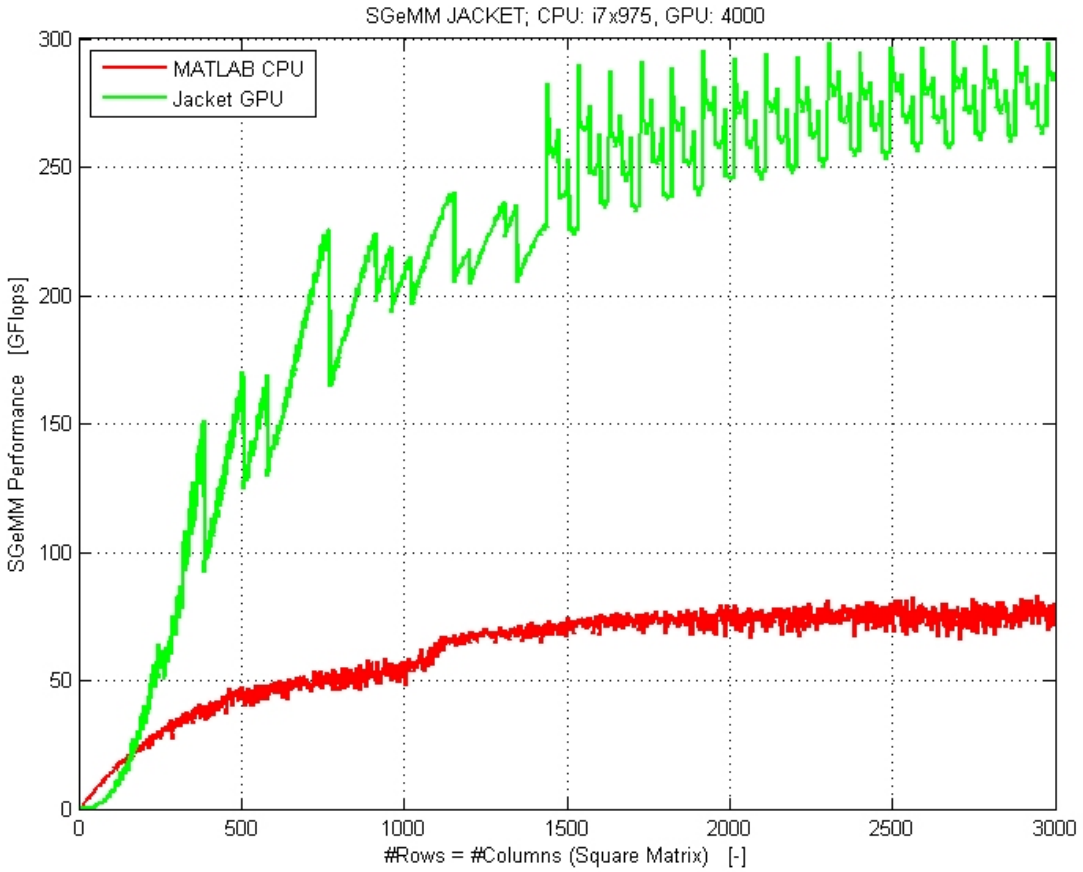
N^2 add

N^2 mult

$$\#_{\text{float ops.}} = 2\,(N^3 + N^2)$$

Benchmarking Procedures
Floating Point

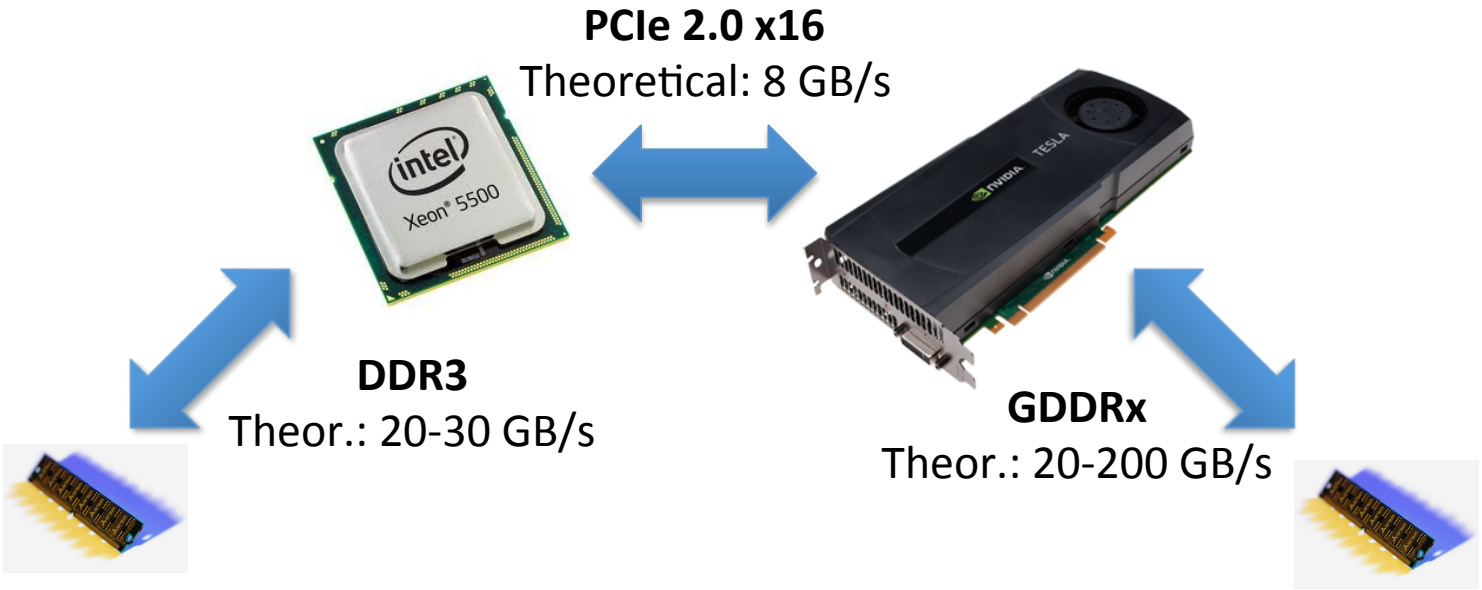
- **Measured single point floating point performance:**
  - **Colfax CXT2000i;** Intel Core i7-970 with 24 GB memory; **NVIDIA Quadro 4000 GPU;** **Asus P6T7 WS SuperComputer**



Memory Transfer

Benchmarking Procedures
Memory Transfer

- **MATLAB resides in CPU memory** >> **Jacket works on GPU data**
  - **Raw transfer rate** >> limited by the PCI bus
  - **Latency** >> caused by read/write access to memory and initialization of buses



- **Transfer time host >> device (CPU >> GPU) model:**

$$t_{\mathrm{cm}\rightarrow\mathrm{gm}}(s) = t_{\mathrm{cg},0} + \frac{1}{r_{\mathrm{cm}\rightarrow\mathrm{gm}}} s$$

$t_{\mathrm{cg},0}$: Initialization time (CPU to GPU case)
$r_{\mathrm{cm}\rightarrow\mathrm{gm}}$: Transfer rate GPU to CPU excl. $t_{\mathrm{cg},0}$ [MB/s]
s: Size of array to be moved [MB]

- **Transfer time device >> host (GPU >> CPU) model:**

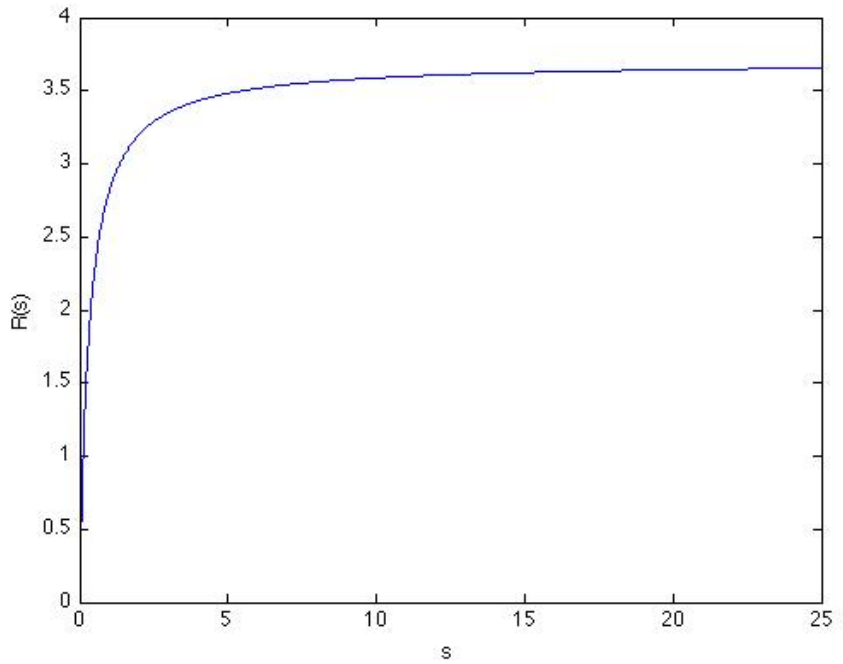
$$t_{\mathrm{gm}\rightarrow\mathrm{cm}}(s) = t_{\mathrm{gc},0} + \frac{1}{r_{\mathrm{gm}\rightarrow\mathrm{cm}}} s$$

$t_{\mathrm{gc},0}$: Initialization time (GPU to CPU case)
$r_{\mathrm{gm}\rightarrow\mathrm{cm}}$: Transfer rate GPU to CPU excl. $t_{\mathrm{gc},0}$ [MB/s]
s: Size of array to be moved [MB]

- **The size dependent transfer rates according to the models are:**

$$R_{\mathrm{cm}\to\mathrm{gm}}(s) = \frac{s}{t_{\mathrm{cm}\to\mathrm{gm}}(s)} = \frac{r_{\mathrm{cm}\to\mathrm{gm}}\, s}{r_{\mathrm{cm}\to\mathrm{gm}} t_{\mathrm{cg},0} + s}$$

$$R_{\mathrm{gm}\to\mathrm{cm}}(s) = \frac{s}{t_{\mathrm{gm}\to\mathrm{cm}}(s)} = \frac{r_{\mathrm{gm}\to\mathrm{cm}}\, s}{r_{\mathrm{gm}\to\mathrm{cm}} t_{\mathrm{gc},0} + s}$$



Array size at which we reach αr:

$$s(\alpha) = \frac{\alpha\, r\, t_0}{1-\alpha}$$

Example: r = 3.7 GB, t = 85 μs:

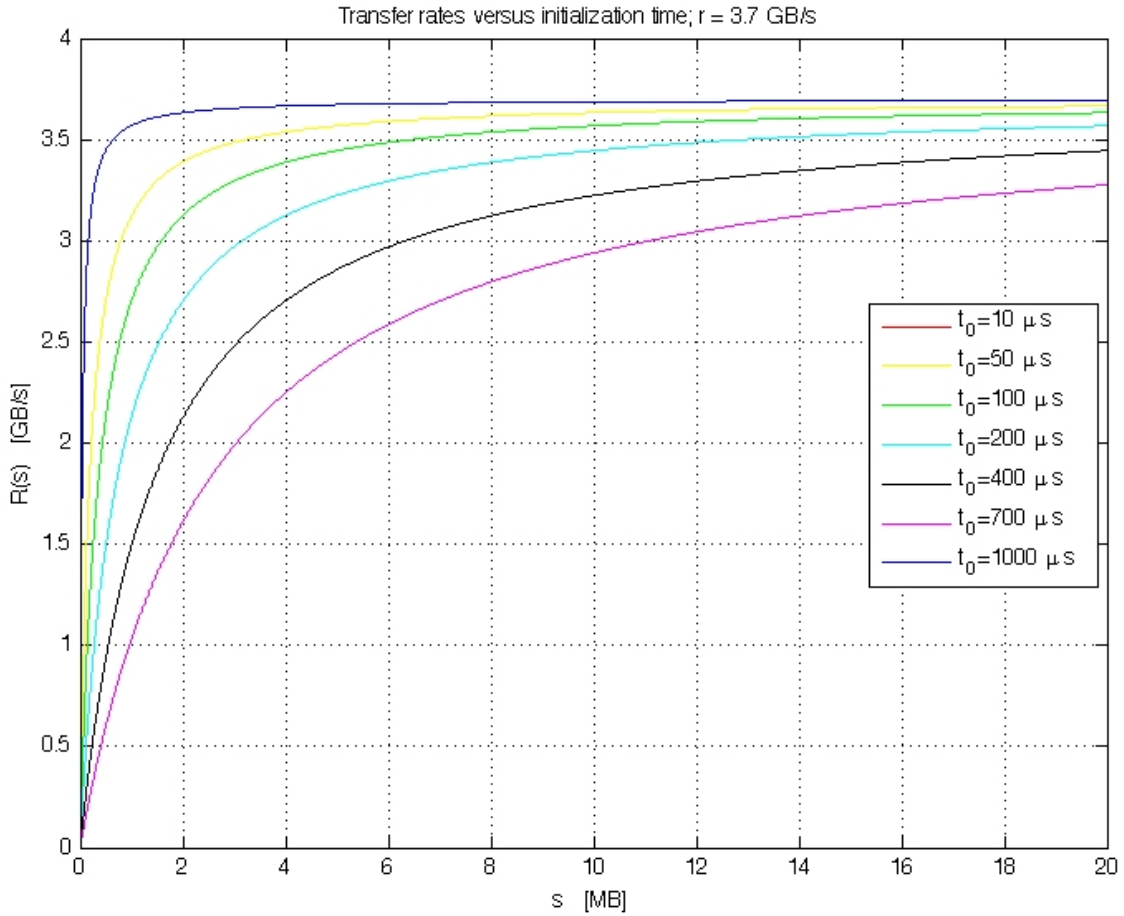
$$s(0.50) = 314\ \mathrm{kB}$$

$$s(0.75) = 944\ \mathrm{kB}$$

$$s(0.90) = 2.83\ \mathrm{MB}$$

$$s(0.95) = 5.98\ \mathrm{MB}$$

- **Transfer rate versus the initialization time:**



Benchmarking Procedures
Memory Transfer

Loop over different array sizes

Determine number of repetitions. Problematic to just transfer e.g. 20 GB in total no matter the array size. This means a huge number of repetitions for small array sizes such as 1 kB. We need to set a max. number of repetitions to perform

Reset generator

Form CPU array reference

Form GPU array reference. Force evaluation to push array to the GPU

```
% Loop over the different array sizes
 no = 0;
for sz=Size
    % Update index number
    no = no + 1;

    % Print progress
    fprintf('VecTransfer: %4.0f of %4.0f: ', no, length(Size));

    % Determine averaging factor such that a minimum
    % amount of data is moved.
    % Here 10 times the largest array size is moved ? however
    % done such that
    % there is a certain max for the number of repetitions.
    E = min(ceil(TotalTransfer/(4*sz)),2E5);

    % Reference vector
    reset(RandStream.getDefaultStream);
    a = rand(sz,1,'single');
    a_ = gsingle(a); geval(a_);
    .
    .
    .
```

Benchmarking Procedures
Memory Transfer

Warm-up and time the CPU activity. The time to change the MATLAB defined array, tvec1, should not be included as CPU-GPU transfer time

Warm-up and time the CPU-GPU transfer time. We need to change an array element in each repetition. Jacket is so clever that it will see if nothing is changed and will then only make the transfer once and not the E times we expect.

Save the timing result and compute the transfer rate. Here we measure single precision arrays. The results are size wise similar when using double precision.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% MOVING DATA FROM CPU TO GPU
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Compensation: warm-up and time the CPU activity
for e=1:E, a(1)=e; a; end
tic; for e=1:E, a(1)=e; end; tvec1=toc;

% Measurement: Warm-up and measure time
for e=1:E, a(1)=e; geval(gsingle(a)); end
gsync; tic;
for e=1:E, a(1)=e; geval(gsingle(a)); end;
gsync; tend1=toc;

% Memory transfer and print result
c2g_time(no) = (tend1-tvec1)/E;
c2g_rate(no) = 4*sz/c2g_time(no);
fprintf('    C>G: %6.3f [GB/s],', c2g_rate(no)/1E9);
```

Benchmarking Procedures
Memory Transfer

Warm-up and time the CPU activity. The time to change the MATLAB defined array, tvec1, should not be included as CPU-GPU transfer time

Warm-up and time the GPU-CPU transfer time. We need to change an array element in each repetition. We want to make sure no run time optimization means saving the transfer.

Save the timing result and compute the transfer rate. Here we measure single precision arrays. The results are size wise similar when using double precision.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% MOVING DATA FROM GPU TO CPU
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Compensation: warm-up and time the GPU activity
for e=1:E, a_(1)=e; geval(a_); end
gsync; tic;
for e=1:E, a_(1)=e; geval(a_); end;
gsync; tvec2=toc;

% Measurement: Warm-up and measure time
for e=1:E, a_(1)=e; single(a_); end
gsync; tic;
for e=1:E, a_(1)=e; single(a_); end;
gsync; tend2=toc;

% Memory transfer and print result
g2c_time(no) = (tend2-tvec2)/E;
g2c_rate(no) = 4*sz/g2c_time(no);
fprintf('   G>C: %6.3f [GB/s]\n', g2c_rate(no)/1E9);
```

Benchmarking Procedures
Memory Transfer

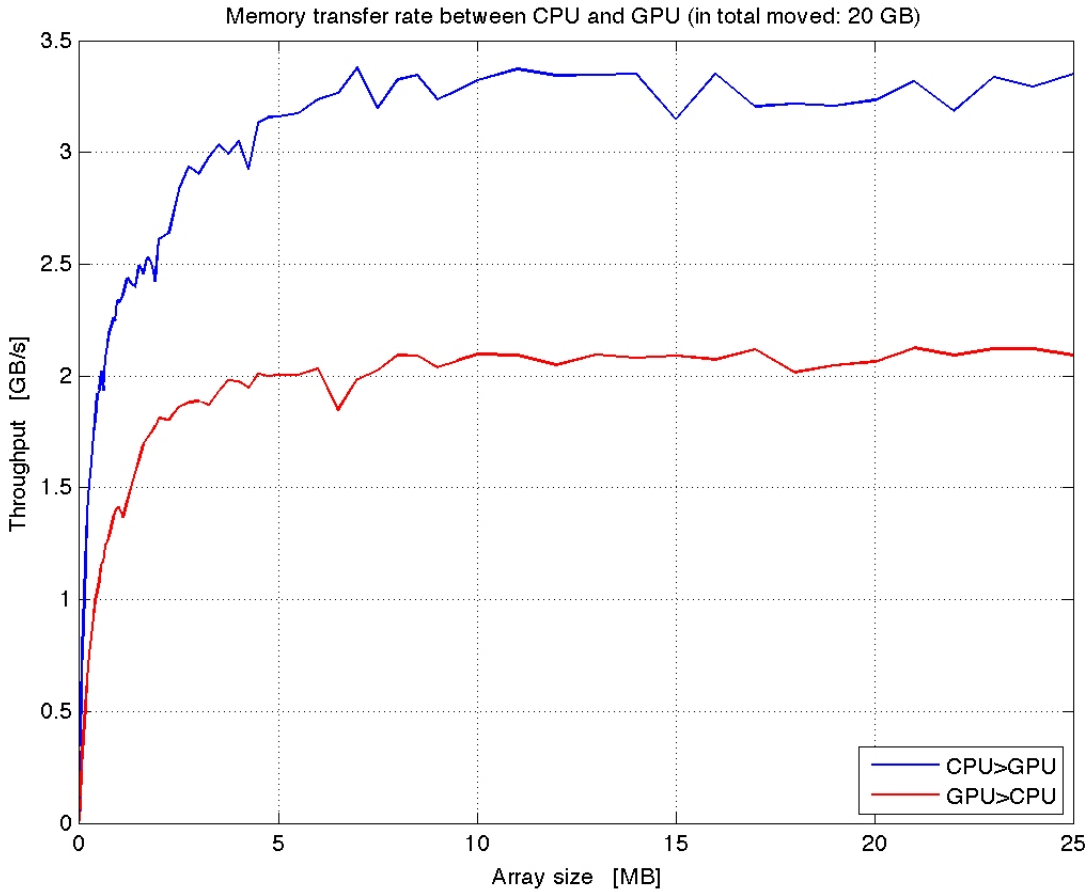
Plot data and save intermediate results for every 10 array sizes measured.
Makes it possible to resume the test in case of a failure or whatever.

```
        %% PLOT DATA EVERY NOW AND THEN
        if rem(no,10)==0
            pltfct(Fname, GPU, Size_bytes(1:no), c2g_rate(1:no), ...
                   g2c_rate(1:no), c2g_time(1:no), g2c_time(1:no), ...
                   TotalTransfer);
        end

        % Release GPU memory
        clear gpu_hook;

    end
```

Release GPU memory not used to be on the safe side

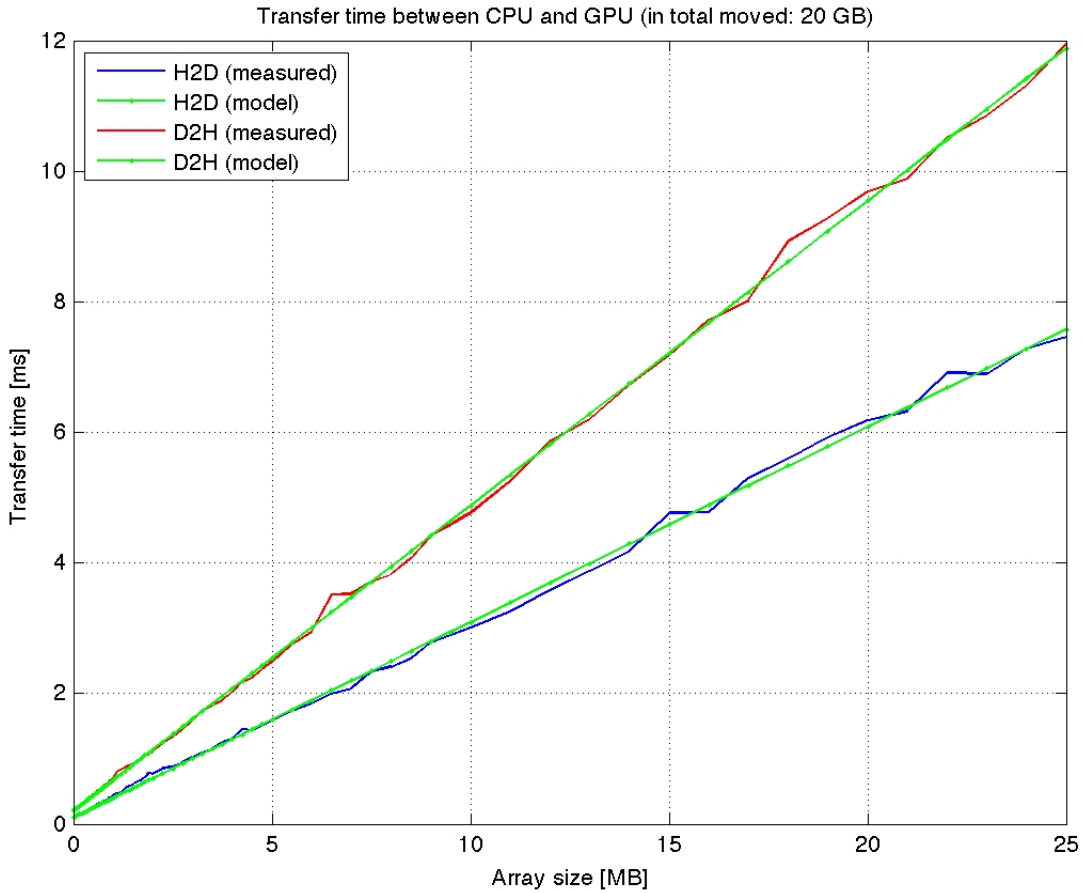
Benchmarking Procedures
Memory Transfer

- **Transfer rate:**
  - **Colfax CXT2000i;** Intel Core i7-975 with 12 GB memory**; NVIDIA Quadro 4000 GPU; Asus P6T7 WS SuperComputer**



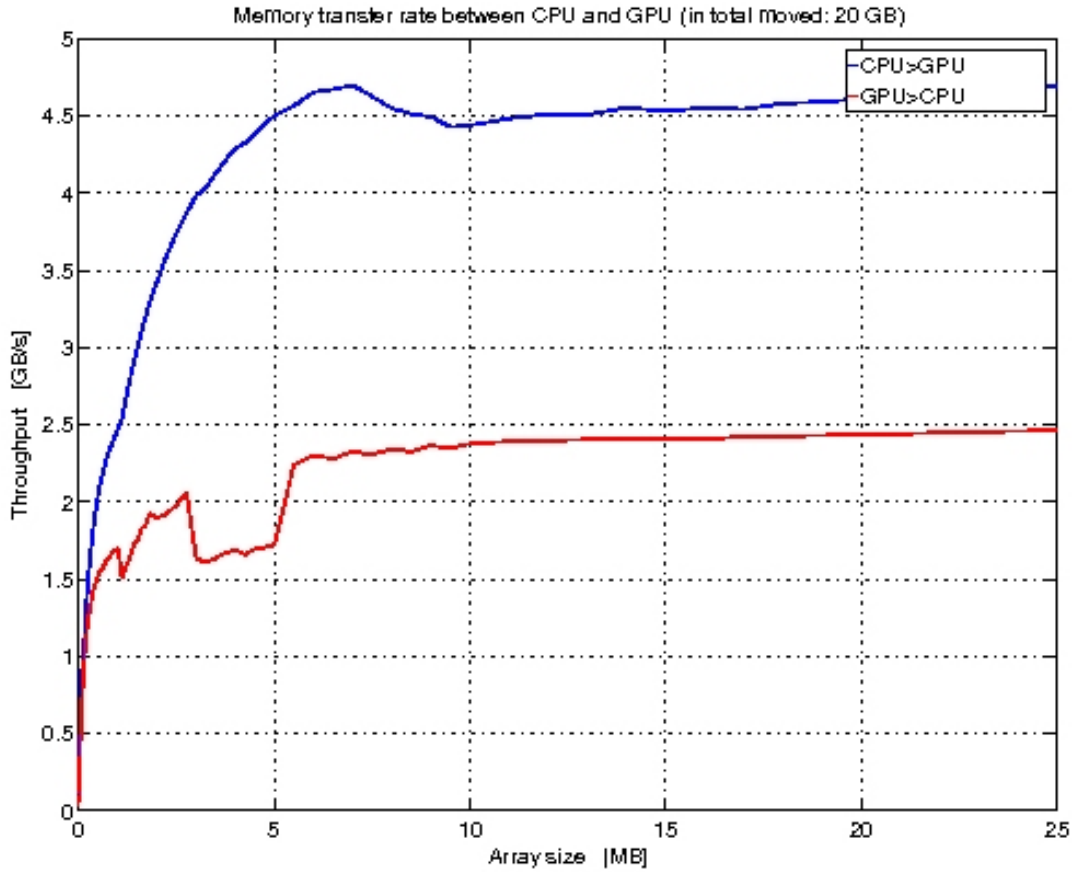
Benchmarking Procedures
Memory Transfer

- **Transfer time:**
  - **Colfax CXT2000i;** Intel Core i7-975 with 12 GB memory**; NVIDIA Quadro 4000 GPU;** Asus P6T7 WS SuperComputer

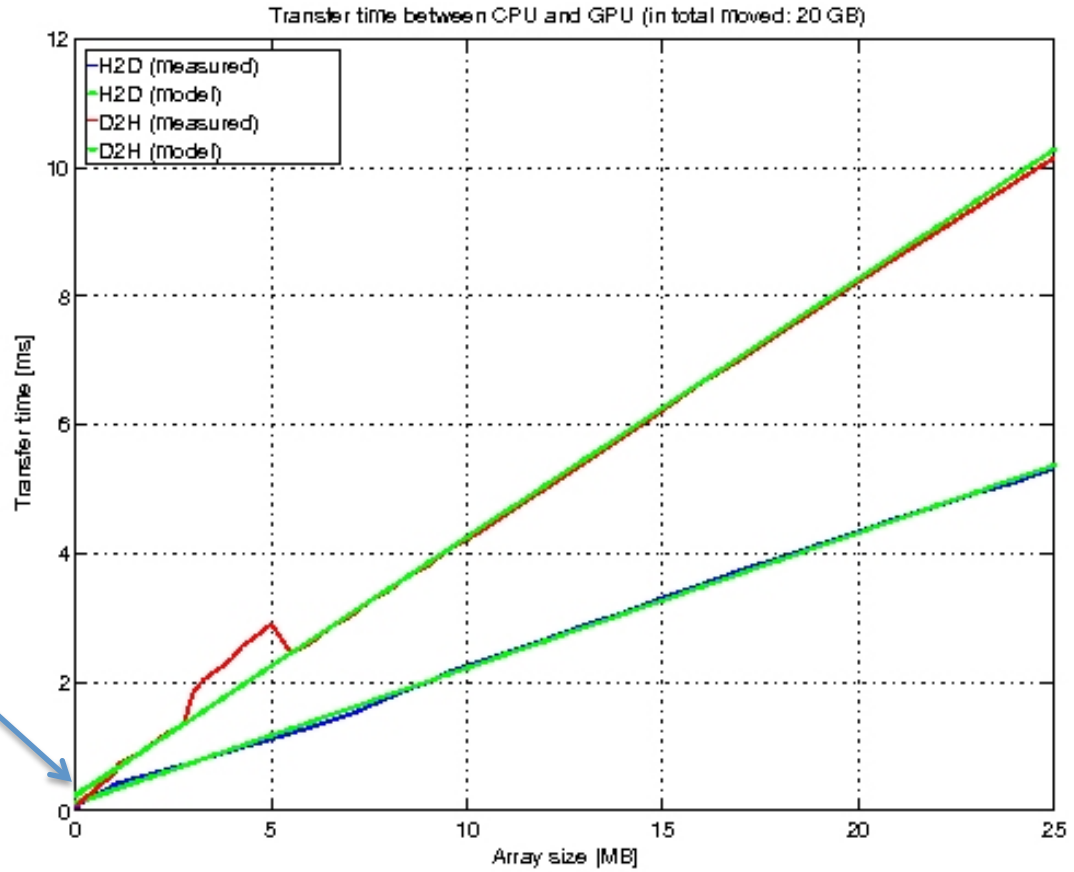


Benchmarking Procedures
Memory Transfer

- **Transfer rate:**
  - **Colfax CXT2000i;** Intel Xeon X5570 with 48 GB memory; **NVIDIA Tesla C2070 GPU;** Colfax CXT4000 computer



- **Transfer time:**
  - **Colfax CXT2000i;** Intel Xeon X5570 with 48 GB memory**; NVIDIA Tesla C2070 GPU; Colfax CXT4000 computer**



Linear regression seems to overestimate t0.

Benchmarking Procedures
Memory Transfer

- **Measured parameters** host >> device (CPU >> GPU):

GPU type	$t_{cg,0}$ [μs]	$r_{cm \rightarrow gm}$ [GB/s]	Peak rate [GB/s]	Array size [MB]
MacBook Air / 320M	00.0	000.0	0.000	0.000
GeForce GTX465	00.0	000.0	0.000	0.000
GeForce GTX580	84.1	3.871	3.885	7.000
Quadro FX3800	84.7	3.705	3.773	25.000
Quadro 2000	00.0	0.000	0.000	0.000
Quadro 4000	83.1	3.870	3.892	7.000
Tesla C1060	000.0	0.000	0.000	0.000
Tesla C2050 (ECC)	78.6	2.571	2.705	6.000
Tesla C2050 (no ECC)	85.0	3.850	3.906	7.000

- Parameters across non ECC enabled GPUs on the Asus P6T7 motherboard:
  - Initialization time: ~ 85 µs.
  - Transfer rate: 3.75 GB/s.
- **The Colfax computers are of very high quality – don't expect the same high performance for discount systems.**

- **Measured parameters** device >> host (GPU >> CPU):

GPU type	$t_{gc,0}$ [μs]	$r_{gm \to cm}$ [GB/s]	Peak rate [GB/s]	Array size [MB]
MacBook Air / 320M	000.0	0.000		
GeForce GTX465	000.0	0.000		
GeForce GTX580	169.1	2.559	2.520	10.000
Quadro FX3800	206.2	2.444	2.433	19.000
Quadro 2000	000.0	0.000		
Quadro 4000	178.8	2.569	2.532	21.000
Tesla C1060	000.0	0.000		
Tesla C2050 (ECC)	285.2	1.884	1.842	21.000
Tesla C2050 (no ECC)	176.8	2.540	2.535	21.000

Functions

FIND

SVD

XCORR

LOAD

PWELCH

FFT

PSD

MULTIPLY

IFFT

RANDN

MLDIVIDE

RANK

XCORR

ANY

MULTIPLY

ALL

Benchmarking Procedures
Functions

- To observe …
  - **Start MATLAB with high priority**
    - Windows: task manager; Mac/Linux: “nice –n –XXX matlab …”
  - **For the CPU the number of threads is very important for some functions**
    - check with “`maxNumCompThreads`”

Benchmarking Procedures
Functions

Perform CPU benchmark until minimum test time is met

First time estimate number of repetitions based on two executions of the function; second or later times increase number of repetitions

Warm-up by computing the same number of repetitions as in the actual test

Perform the actual test using the estimated number of repetitions

Measure the loop time – this time is so short that a loop around it is needed to estimate it

CPU execution time for one call of the function under test

```
whilecount = 0;   Telap_cpu = -1;
while Telap_cpu < Tmin
  whilecount = whilecount + 1;
  if Telap_cpu == -1
    t1 = tic;   Rc = all(Ac);   Rc = all(Ac);   Telap_cpu = toc(t1)/2;
    NoRunsCPU = ceil(1.5*Tmin/Telap_cpu);
  else
    NoRunsCPU = ceil(1.5*whilecount*NoRunsCPU/Telap_cpu*Tmin);
  end

  % Warm-up and benchmark
  for no=1:NoRunsCPU,   Rc = all(Ac);   end
  tstart1 = tic;
  for no=1:NoRunsCPU,   Rc = all(Ac);   end
  Telap_cpu = toc(tstart1);
end

% Determine time for CPU loop alone
RPT = min(5E3,ceil(MaxAvg/NoRunsCPU));   tstart = tic;
for AvgNo=1:RPT,   for no=1:NoRunsCPU, end;   end
T_CPU_Loop = toc(tstart)/RPT;

% Compute CPU times
T_CPU = max((Telap_cpu-T_CPU_Loop)/NoRunsCPU,2.5E-10);
T_CPU_tot = Telap_cpu;
```

Benchmarking Procedures
Functions

Perform GPU benchmark until minimum test time is met

First time estimate number of repetitions based on two executions of the function; second or later times increase number of repetitions

Warm-up by computing the same number of repetitions as in the actual test

Perform the actual test using the estimated number of repetitions

Measure the loop time – this time is so short that a loop around it is needed to estimate it

```
Ag = gdouble(Ac);   whilecount = 0;   Telap_gpu = -1;
while Telap_gpu < Tmin
  whilecount = whilecount + 1;
  if Telap_gpu == -1
    gsync;  t1 = tic;
    Rg = all(Ag);  geval(Rg);  Rg = all(Ag);  geval(Rg);
    gsync;  Telap_gpu = toc(t1)/2;
    NoRunsGPU = ceil(1.5*Tmin/Telap_gpu);
  else
    NoRunsGPU = ceil(1.5*whilecount*NoRunsGPU/Telap_gpu*Tmin);
  end

  % Warm-up and benchmark
  for no=1:NoRunsGPU,   Rg = all(Ag);  geval(Rg);   end;
  gsync;   tstart1 = tic;
  for no=1:NoRunsGPU,   Rg = all(Ag);  geval(Rg);   end
  gsync;   Telap_gpu = toc(tstart1);
end

% Determine time for GPU loop alone
RPT = min(5E3,ceil(MaxAvg/NoRunsGPU));
tstart = tic;
for AvgNo=1:RPT,   for no=1:NoRunsGPU,  end;   end
T_GPU_Loop = toc(tstart)/RPT;
```

Benchmarking Procedures
Functions

GPU execution time for one call of the function under test

Speed-up of GPU relative to CPU

Estimate memory usage and release memory not used

```
% Compute GPU times
T_GPU = max((Telap_gpu-T_GPU_Loop)/NoRunsGPU,2.5E-10);
T_GPU_tot = Telap_gpu;
fprintf('   T_GPU: %6.1f', T_GPU_tot);

% Speed-up
Speedup = T_CPU/T_GPU;
fprintf('   |   Speed-up (all-MD):      %10.4f', Speedup);

% Memory
gpu_info = gpu_entry(13);
Mem_MB = gpu_info.gpu_free/1E6;
clear gpu_hook;
fprintf('   |   Mem free [MB]:  %6.1f\n', Mem_MB);
```

Benchmarking Procedures
Functions

- **Colfax CXT2000i;** CPU: Intel Core i7-970; GPU**: NVIDIA Quadro 4000; MB**: Asus P6T7 WS SuperComputer

Measured Speed–Up for Quadro 4000 vs. Core i7–970				
Function	Matrix		Vector	
	Single	Double	Single	Double
all	2.28	2.50	5.10	5.73
any†	2.25	2.51	5.39	5.76
asinh	16.55	5.54	16.63	5.66
atan2	108.54	48.25	108.57	48.32
atan	13.34	3.49	13.10	3.49
chol	26.93	1.11	—	—
conv2	0.90	—	—	—
cos	9.13	5.39	8.96	5.35
det	0.77	14.78	—	—
exp	10.91	6.82	11.79	7.01
fft	8.90	4.05	47.54	18.63
find	11.16	11.03	10.82	11.04

Benchmarking Procedures

Functions

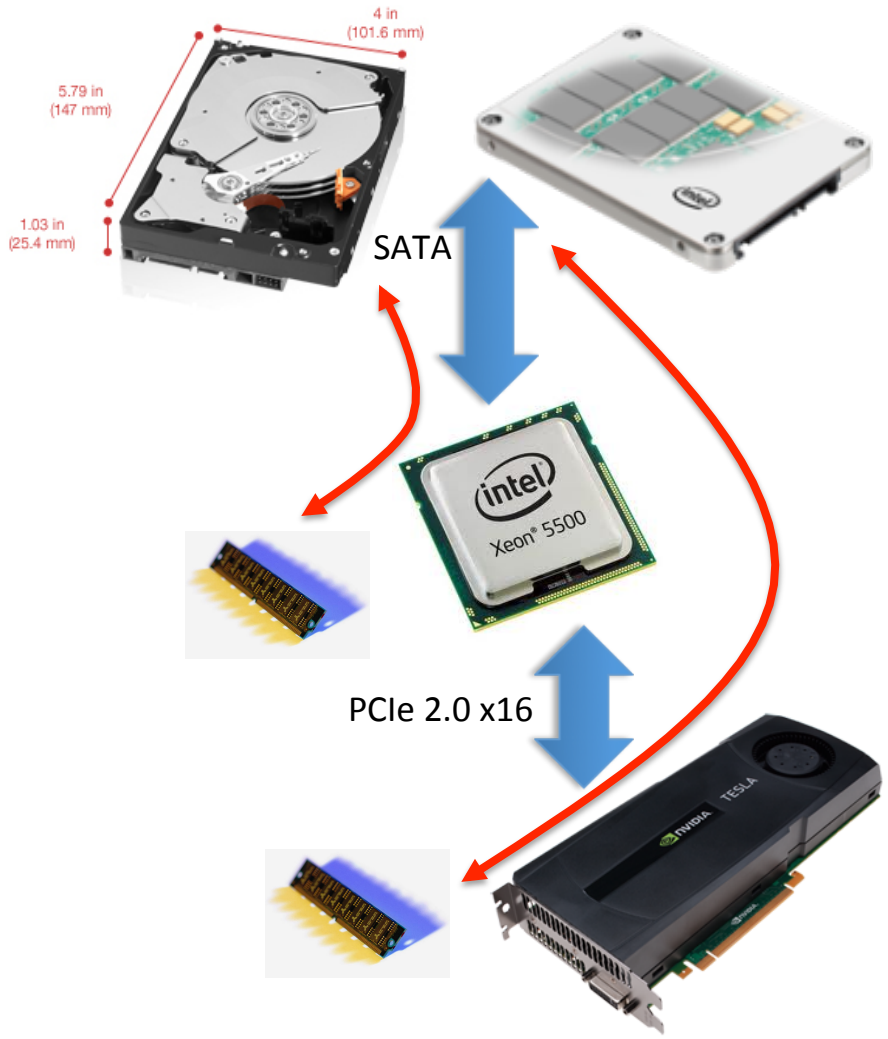
Measured Speed–Up for Quadro 4000 vs. Core i7–970				
Function	Matrix		Vector	
	Single	Double	Single	Double
ifft	3.59	3.62	23.75	15.05
interp1	—	—	111.98	99.58
interp2	193.83	—	—	—
inv	—	5.33	—	—
load	0.96	1.00	0.96	0.97
log	8.29	8.81	8.71	8.82
lu	0.92	16.89	0.45	0.49
max	0.62	1.32	1.76	3.32
min	0.61	1.32	1.75	3.33
minus	8.70	8.39	8.65	8.41
mldivide	1.20	7.05	—	—
mpower	2.54	2.87	—	—

Benchmarking Procedures

Functions

Measured Speed–Up for Quadro 4000 vs. Core i7–970				
Function	Matrix		Vector	
	Single	Double	Single	Double
norm	0.33	0.61	1.84	17.88
plus	8.57	8.39	8.59	8.47
power	16.17	6.87	16.39	6.83
rand	30.86	32.86	30.40	32.98
randn	23.29	12.62	23.21	12.65
rank	0.06	0.07	0.00	0.00
rdivide	4.65	4.00	4.70	4.00
save	1.01	1.02	0.99	0.98
sort	2.84	2.37	4.73	1.86
subsasgn	0.11	0.10	0.02	0.01
sum	0.66	1.31	2.41	3.37
svd	0.04	0.07	0.01	0.02
times	8.70	8.38	8.63	8.38
trapz	1.05	11.56	0.74	0.86

Disk Access

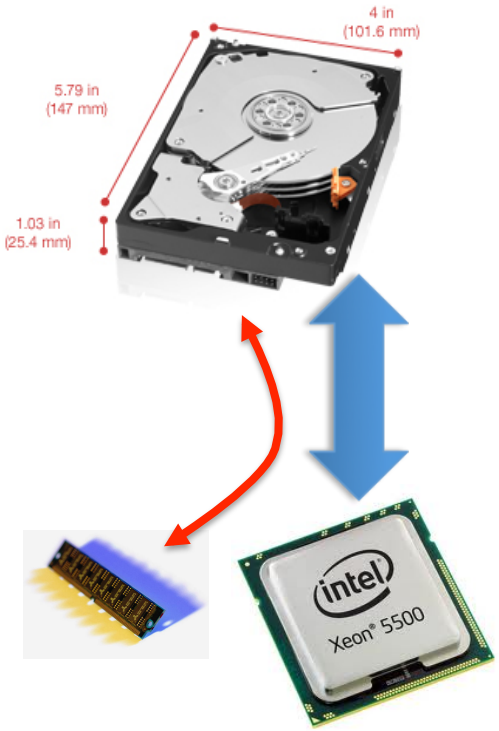

4 in
(101.6 mm)
5.79 in
(147 mm)
1.03 in
(25.4 mm)
SATA
intel
Xeon® 5500
PCIe 2.0 x16
NVIDIA
TESLA

Benchmarking Procedures
Disk Access

- **Measurement method**; two approaches:
  - Use of MATLABs `save/load` using ".mat" file types
  - Use of MATLABs `fwrite/fread` in a binary format

Benchmarking Procedures
Disk Access

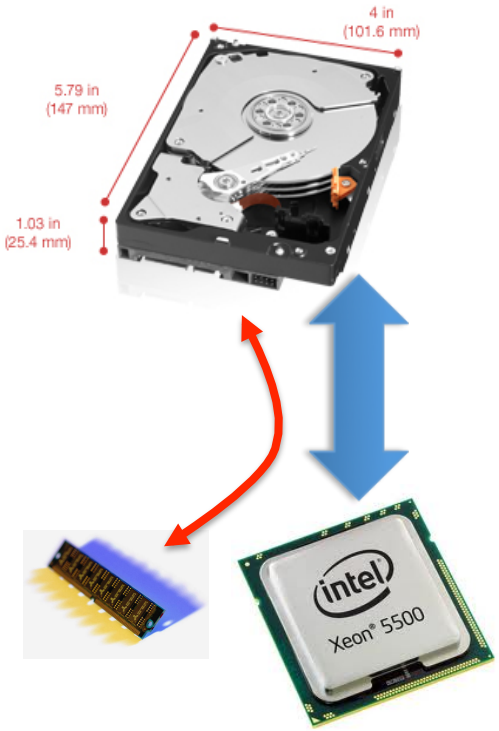
save/load
.MAT




```
function [thr_c2dMAT,thr_d2cMAT] = saveCPU2Disk(sz,Fname,RPT,VERB)
  % SaveCPU2Disk Transfer rate by use of save/load in MAT format.

  % Generate reference matrix
  reset(RandStream.getDefaultStream);
  Acpu = randn(sz,sz,'single');

  % Save data from the CPU to disk
  tstart = tic;
  for rpt=1:RPT
    save(Fname, 'Acpu');
  end
  T_c2dMAT = toc(tstart)/RPT;  clear Acpu;

  % Load data from disk to the CPU
  tstart = tic;
  for rpt=1:RPT
    load(Fname);
  end
  T_d2cMAT = toc(tstart)/RPT;

  % Compute data
  Size = 4*sz^2;
  thr_c2dMAT = Size/T_c2dMAT;
  thr_d2cMAT = Size/T_d2cMAT;
end
```

Benchmarking Procedures
Disk Access

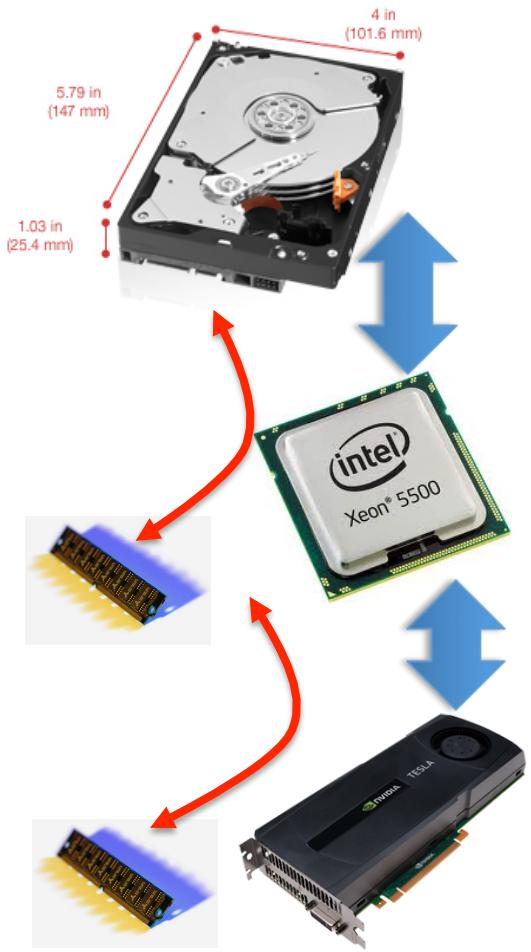
fwrite/fread
.MAT




```
function [thr_c2dBIN,thr_d2cBIN] = fwriteCPU2Disk
(sz,Fname,RPT,VERB)
  % fwriteCPU2Disk Transfer rate using fwrite/fread, BIN format.

  % Generate reference matrix
  reset(RandStream.getDefaultStream);    Acpu = randn
(sz,sz,'single');

  % Save data from the CPU to disk
  tstart = tic;
  for rpt=1:RPT
    fid = fopen(Fname,'w');  fwrite(fid, Acpu, 'single');
    fclose(fid);
  end
  T_c2dBIN = toc(tstart)/RPT;    clear Acpu;

  % Load data from disk to the CPU
  for rpt=1:RPT
    fid = fopen(Fname, 'r');
    Acpu_fread = fread(fid, [sz,sz], '*single');  fclose(fid);
  end
  T_d2cBIN = toc(tstart)/RPT;

  % Compute data
  Size = 4*sz^2;
  thr_c2dBIN = Size/T_c2dBIN;
  thr_d2cBIN = Size/T_d2cBIN;
end
```

Benchmarking Procedures
Disk Access

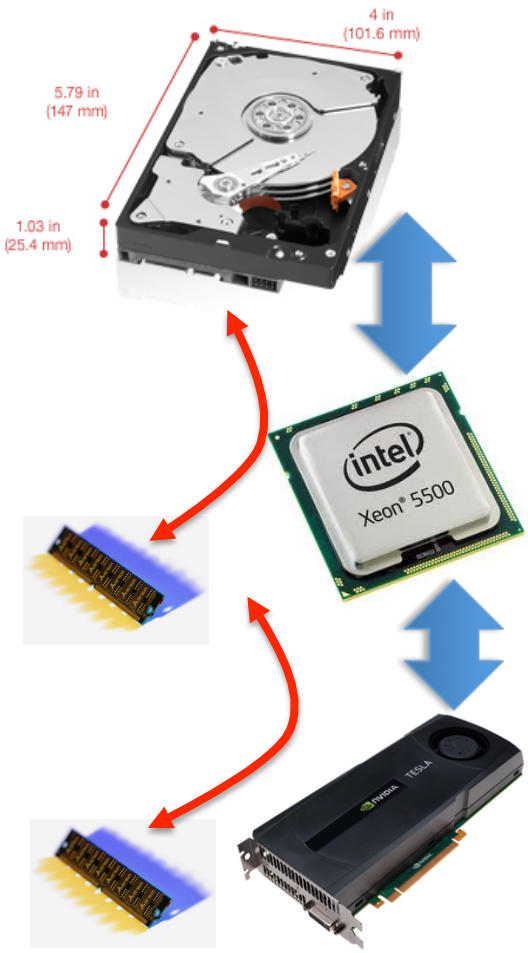
save/load .MAT




```
function [thr_g2dMAT,thr_d2gMAT] = saveGPU2Disk(sz,Fname,RPT,VERB)
  % SaveGPU2Disk Transfer rate by use of save/load in MAT format.

  % Generate reference matrix
  reset(RandStream.getDefaultStream);
  Agpu = gsingle(randn(sz,sz,'single'));  geval(Agpu);

  % Save data from the GPU to disk
  tstart = tic;
  for rpt=1:RPT
    Acpu = single(Agpu);  save(Fname, 'Acpu');
  end
  T_g2dMAT = toc(tstart)/RPT;  clear Acpu;

  % Load data from disk to the GPU
  tstart = tic;
  for rpt=1:RPT
    load(Fname,'Acpu');
    Agpu = gsingle(Acpu);
    geval(Agpu);
  end
  T_d2gMAT = toc(tstart)/RPT;

  % Compute data
  Size = 4*sz^2;
  thr_g2dMAT = Size/T_g2dMAT;
  thr_d2gMAT = Size/T_d2gMAT;
end
```

Benchmarking Procedures
Disk Access

fwrite/fread
.MAT




```
function [thr_g2dBIN,thr_d2gBIN] = fwriteGPU2Disk
(sz,Fname,RPT,VERB)
  % fwriteGPU2Disk Transfer rate using fwrite/fread in BIN format.

  % Generate reference matrix
  reset(RandStream.getDefaultStream);
  Agpu = gsingle(randn(sz,sz,'single'));  geval(Agpu_ref);

  % Save data from the GPU to disk
  tstart = tic;
  for rpt=1:RPT
    Acpu = single(Agpu);          fid = fopen(Fname,'w');
    fwrite(fid, Acpu, 'single');  fclose(fid);
  end
  T_g2dBIN = toc(tstart)/RPT;  clear Acpu;

  % Load data from disk to the GPU
  for rpt=1:RPT
    fid = fopen(Fname, 'r');
    Acpu_fread = fread(fid, [sz,sz], '*single');
    Agpu = gsingle(Acpu_fread);
    geval(Agpu);  fclose(fid);
  end
  T_d2gBIN = toc(tstart)/RPT;

  % Compute data
  Size = 4*sz^2;
  thr_g2dBIN = Size/T_g2dBIN;   thr_d2gBIN = Size/T_d2gBIN;
end
```

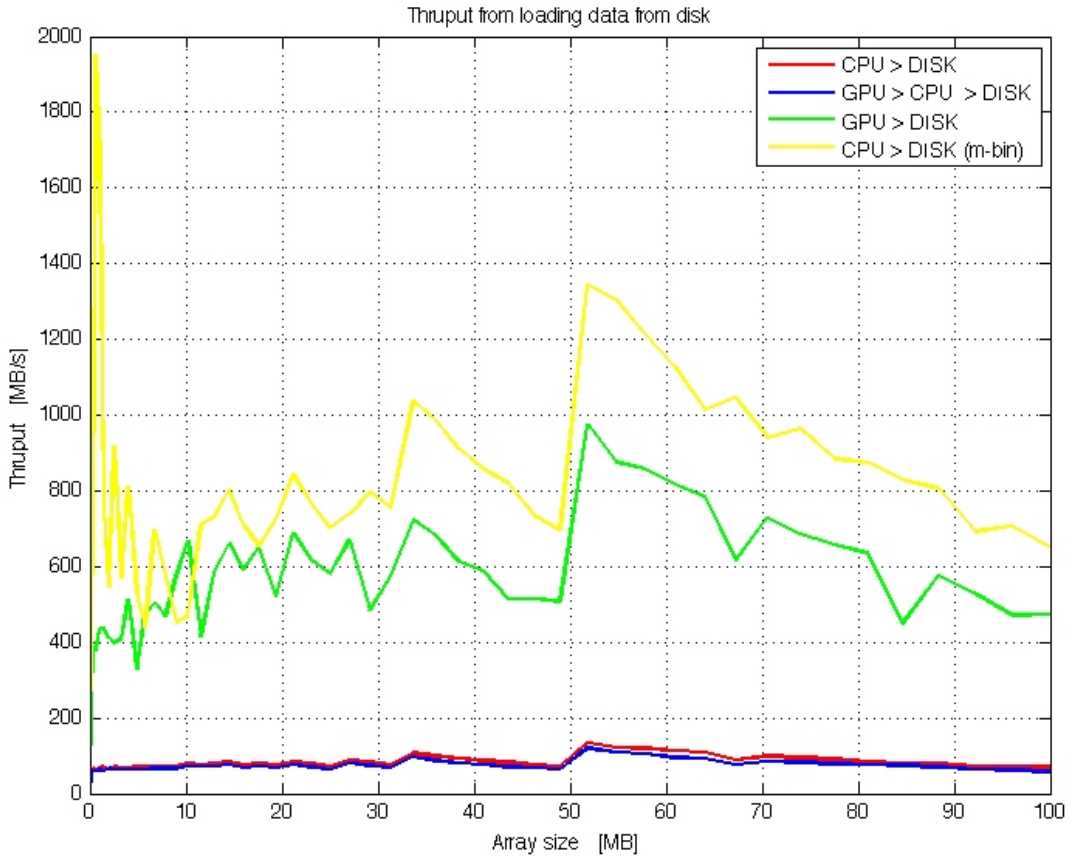
Disk Access

- **Performance of disk system when saving/loading from MATLAB and Jacket**
  - The spindle disk is a Western Digital RE3 (1 TB)
  - Solid state disk is an Intel X25M-G2 (160 GB)

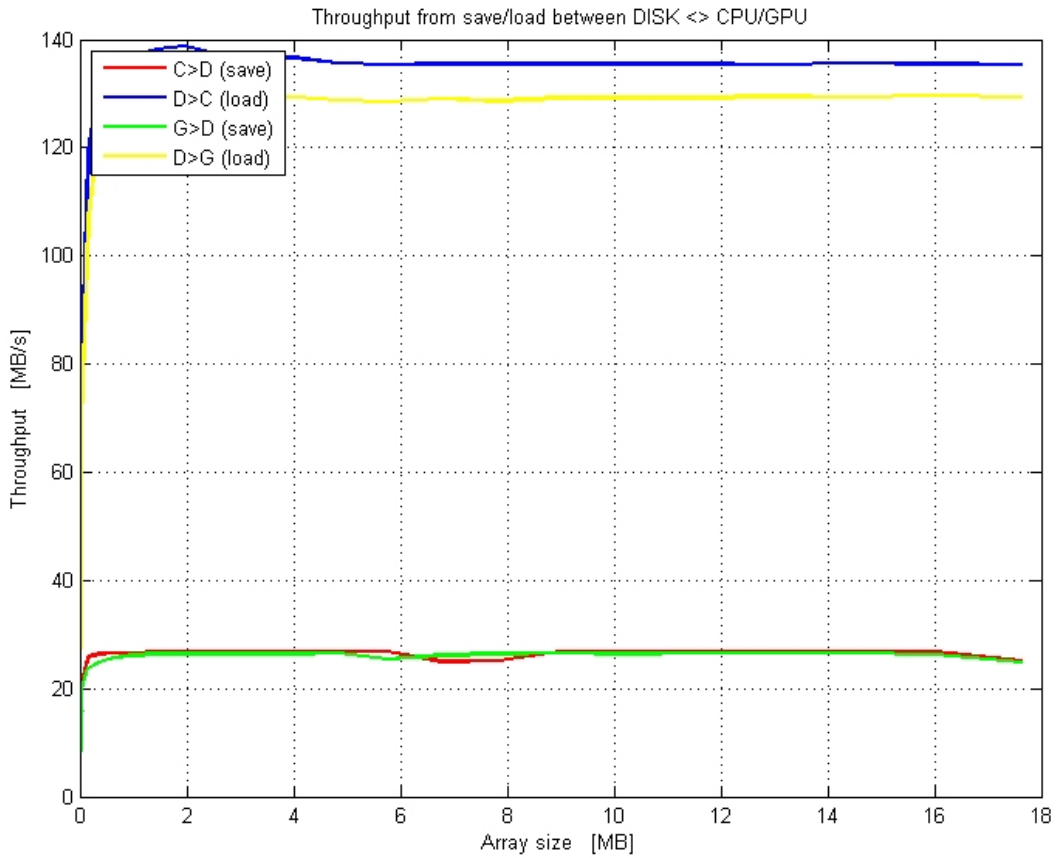
	MATLAB		Jacket	
	Save MB/s	Load MB/s	Save MB/s	Load MB/s
Spindle Disk	32.2	167.0	30.8	157.6
Solid State Disk	32.2	165.3	31.2	156.1

Benchmarking Procedures
Disk Access

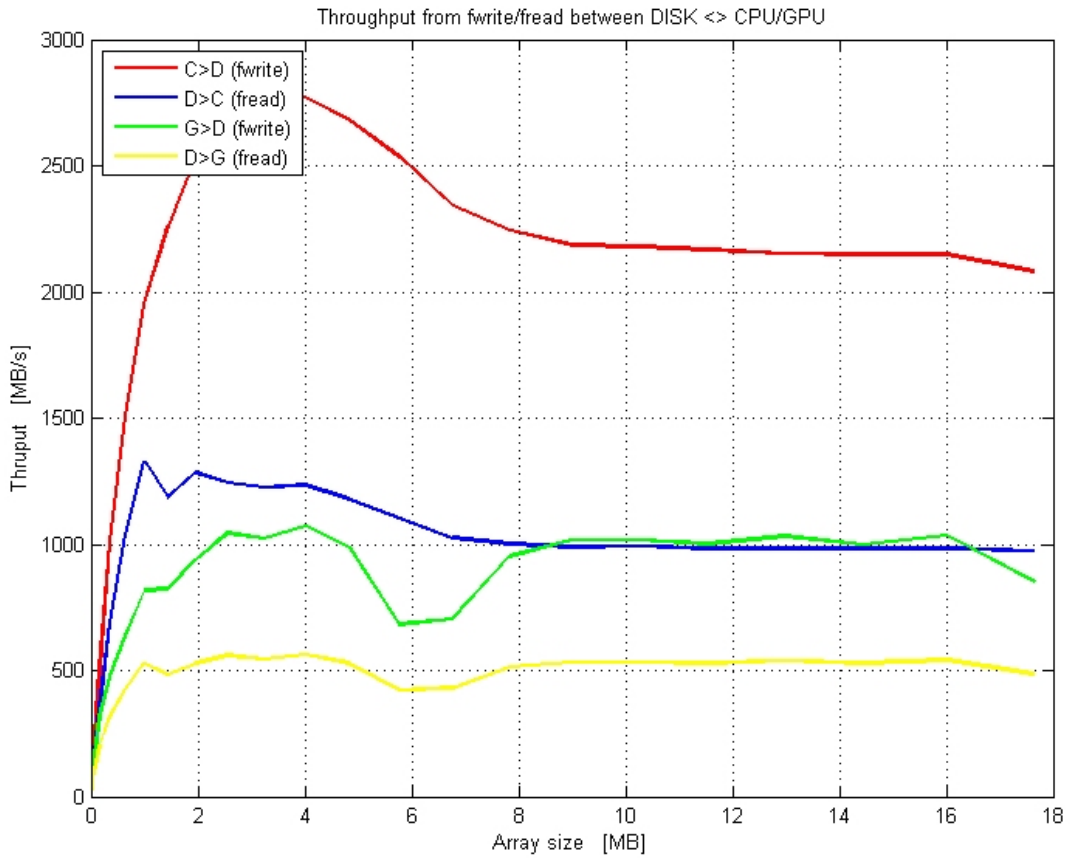
- **Throughput when loading data from disk to CPU and GPU:**



- **Throughput when using SAVE/LOAD:**



- **Throughput when using `FREAD/FWRITE`:**



Toolboxes

Benchmarking Procedures
Toolboxes

- **Four toolboxes are freely available to perform benchmarks:**
  - `GeMM.bench` > Floating point benchmarking
  - `Functional.bench` > Functional benchmarking
  - `MemTransfer.bench` > Memory transfer
  - `Disk.bench` > Disk performance

Further Reading

1) AccelerEyes: Torben's Corner. http://wiki.accelereyes.com/wiki/index.php/Torben's_Corner.
2) Torben Larsen, Gallagher Pryor, and James Malcolm: **"Jacket: GPU Powered MATLAB Acceleration".** In Wen-Mei Hwu (editor): "GPU Computing Gems", Jade Edition, Morgan-Kaufmann, July 2011.
3) NVIDIA: NVIDIA CUDA C Programming Guide, 2010.
4) Jason Sanders and Edward Kandrot: "CUDA By Example – An Introduction to General-Purpose GPU Programming", Addison-Wesley, (Boston, MA, USA), 2011.
5) V. Volkov and J. W. Demmel, Benchmarking GPUs to Tune Dense Linear Algebra. **SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing**, pp. 1-11, Austin, Texas, USA, 2008.
6) Jeff Bolz, Ian Farmer, Eitan Grinspun, Peter Schröder: "Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid". **SIGGRAPH '03: ACM SIGGRAPH 2003 Papers**, pp. 917–924, San Diego, California, USA.
7) Jack J. Dongarra: "**Performance of Various Computers Using Standard Linear Equations Software**". Technical Report CS–89–85, 2010. University of Tennessee, Oak Ridge National Laboratory, and University of Manchester.
8) Stanimire Tomov, Michael McGuigan, Robert Bennett, Gordon Smith, and John Spiletic: "Benchmarking and implementation of probability-based simulations on programmable graphics cards". **Computers & Graphics**, vol. 29, no. 1, pp. 71–80, 2005.
9) Nico Galoppo, Naga K. Govindaraju, Michael Henson, and Dinesh Manocha: "LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware". **SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing**, 2005, pp. 3–14.