technische universität
dortmund

# GPU-Nutzung in MATLAB®

*Andreas Blume*

26. September 2018

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl für praktische Informatik (LS4)
http://ls4-www.cs.tu-dortmund.de

fakultät für
informatik
Lehrstuhl für praktische Informatik

# Inhaltsverzeichnis

**1 Ein erstes Beispiel** **1**
1.1 Matrixmultiplikation . . . . . 1
1.1.1 CPU-Variante . . . . . 1
1.1.2 GPU-Variante . . . . . 2

**2 GPU-Nutzung in MATLAB®** **5**
2.1 Identifizieren und Auswählen einer GPU . . . . . 5
2.2 Ausführungszeit einer GPU-Berechnung messen . . . . . 5
2.2.1 Variante 1 . . . . . 6
2.2.2 Variante 2 . . . . . 6
2.3 GPU-Performance bestimmen . . . . . 7
2.3.1 GPU-Bandbreite testen . . . . . 7
2.3.2 Testen von speicherintensiven Operationen . . . . . 9
2.3.3 Rechenleistung bestimmen (FLOPS) . . . . . 11

**3 GPUs und CPU in MATLAB® parallel verwenden** **15**
3.1 Vorgehen . . . . . 15
3.2 Zeitmessung . . . . . 17

**A ls4gpu2 am Lehrstuhl 4** **19**
A.1 NVIDIA® Quadro® P6000 . . . . . 19
A.2 Intel® Xeon® E5-2699 v4 . . . . . 20

# Kapitel 1

# Ein erstes Beispiel

In diesem Kapitel wird anhand eines Beispiels die Nutzung von Grafikkarten (GPUs) in MATLAB® gezeigt.

## 1.1 Matrixmultiplikation

Die Matrizenmultiplikation oder Matrixmultiplikation ist in der Mathematik eine multiplikative Verknüpfung von Matrizen. Um zwei Matrizen miteinander multiplizieren zu können, muss die Spaltenzahl der ersten Matrix mit der Zeilenzahl der zweiten Matrix übereinstimmen.

### 1.1.1 CPU-Variante

In MATLAB® steht für die Matrizenmultiplikation der Befehl $C = mtimes(A, B)$ oder die Operatorform $C = A * B$ zur Verfügung und kann wie folgt verwendet werden:

```
1  function MatrixMulCPU
2      % erstelle Matrix A
3      A = [1 3 5; 2 4 7];
4  
5      % erstelle Matrix B
6      B = [-5 8 11; 3 9 21; 4 0 8];
7  
8      % berechne Matrixmultiplikation
9      C = A*B;      % oder C = mtimes(A,B);
10     disp(C);      % zeige Werte
11 
12     clearvars     % alle Variablen loeschen
13 end
```

**Code 1.1:** CPU-Matrixmultiplikation in MATLAB®.

### 1.1.2 GPU-Variante

Mittlerweile können in MATLAB® auch eingebaute Funktionen auf einer GPU[1] ausgeführt werden. Eine Übersicht über alle Funktionen mit GPU-Unterstützung kann unter der folgenden Internetadresse eingesehen werden:

https://de.mathworks.com/help/distcomp/run-matlab-functions-on-a-gpu.html

Darunter befinden sich bekannte Funktionen wie: mtimes(...), sin(...), die diskrete Fourier-Transformation (fft) und viele mehr. Immer wenn eine dieser Funktionen mit mindestens einem **gpuArray**[2] als Eingabeargument aufgerufen wird, wird die Funktion auf der GPU ausgeführt und erzeugt ein gpuArray als Ergebnis. Ein gpuArray konvertiert also ein Array im MATLAB®-Arbeitsbereich in ein Array dessen Elemente auf der GPU gespeichert werden. Es ist also nichts anderes als der Transfer zwischen Host und GPU.[3]

```
1  function MatrixMulGPU
2      g = gpuDevice(1);      % waehle erste GPU des Rechners
3
4      % erstelle ein GPU-Array / Matrix im GPU-Speicher
5      A_gpu = gpuArray([1 3 5; 2 4 7]);
6      disp(class(A_gpu));    % pruefe, ob Matrix A_gpu auf der GPU ist
7      disp(A_gpu);           % zeige Werte
8
9      % erstelle ein GPU-Array / Matrix im GPU-Speicher
10     B_gpu = gpuArray([-5 8 11; 3 9 21; 4 0 8]);
11     disp(class(B_gpu));    % pruefe, ob Matrix B_gpu auf der GPU ist
12     disp(B_gpu);           % zeige Werte
13
14     % berechne Matrixmultiplikation
15     C_gpu = A_gpu*B_gpu;   % oder mtimes(A_gpu,B_gpu);
16     disp(class(C_gpu));    % pruefe, ob Matrix C_gpu auf der GPU ist
17     disp(C_gpu);           % zeige Ergebnis
18
19     clearvars              % alle Variablen loeschen
20     gpuDevice([]);         % GPU-Speicher leeren
21 end
```

**Code 1.2:** GPU-Matrixmultiplikation in MATLAB®.

[1] https://de.mathworks.com/help/distcomp/gpu-support-by-release.html
[2] https://de.mathworks.com/help/distcomp/gpuarray_object.html
[3] https://de.mathworks.com/help/distcomp/establish-arrays-on-a-gpu.html

Um eine Matrixmultiplikation mit den eingebauten Funktionen von MATLAB® auf einer GPU ausführen zu können, muss zunächst eine GPU des Computers ausgewählt werden. Das kann mit der Funktion **gpuDevice()** gemacht werden.[4] Danach werden die beiden Matrizen $A_gpu$ und $B_gpu$ mit Hilfe von gpuArray auf der GPU angelegt und die Berechnung kann wie gewohnt mit:

$$C_gpu = mtimes(A_gpu, B_gpu)$$

oder kurz:

$$C_gpu = A_gpu * B_gpu$$

erfolgen (siehe Code 1.2).

[4] https://de.mathworks.com/help/distcomp/identify-and-select-a-gpu-device.html

# Kapitel 2

# GPU-Nutzung in MATLAB®

## 2.1 Identifizieren und Auswählen einer GPU

Die GPU-Anzahl in einem Computer bestimmen:

```
numGPUs = gpuDeviceCount();
```

Eine GPU auswählen:

```
gpu = gpuDevice(i); % die i-te GPU des Rechners wählen
```

Alle GPU-Informationen anzeigen:

```
gpu = gpuDevice(i);
disp(gpu);
```

Den Namen der GPU anzeigen:

```
gpu = gpuDevice(i);
disp(gpu.Name);
```

GPU resetten (u.a. GPU-Speicher leeren)

```
gpu = gpuDevice(i); % GPU auswählen
gpuDevice([]); % oder reset(gpu);
```

Weitere Informationen zu diesem Thema können unter folgendem Link gefunden werden:
https://de.mathworks.com/help/distcomp/identify-and-select-a-gpu-device.html

## 2.2 Ausführungszeit einer GPU-Berechnung messen

Um die Ausführungszeit einer GPU-Berechnung zu messen stehen zwei Varianten[1] zur Verfügung, die als MATLAB®-Skript unter `code/GPU_performace.m` zu finden sind.

[1] Quelle: https://de.mathworks.com/help/distcomp/measure-and-improve-gpu-performance.html#bt2g5cb-1

### 2.2.1 Variante 1

Der beste Weg, um die Ausführungszeit einer GPU-Berechnung zu messen, ist die Verwendung der GPU-Variante der timeit-Funktion: **gputimeit(F,N)**.[2] Diese Funktion bekommt als Eingabe einen Funktionshandle F (ohne Eingabeargumente) und deren Anzahl an Rückgabeparametern (N). Der Rückgabewert von gputimeit ist die gemessene Ausführungszeit der durch den Funktionshandle spezifizierte Funktion in Sekunden. Dabei stellt gputimeit sicher, dass alle GPU-Operationen vor der engültigen Zeitnahme abgeschlossen sind.

Der Code 2.1 zeigt die Verwendung:

```
function GPU_performace
    g = gpuDevice(1);    % waehle die erste GPU des Rechners

    N = 10000;  % Groesse von Array/Matrix A
    A = rand(N, 'gpuArray');
    fh = @() lu(A);  % LU-Faktorisierung der Matrix A
    disp(['Zeit von Variante 1: ', num2str(gputimeit(fh, 2)),'s']);

    clearvars        % alle Variablen loeschen
    gpuDevice([]);  % GPU-Speicher leeren
end
```

**Code 2.1:** GPU-Performance mit MATLAB® messen. (Variante 1)

### 2.2.2 Variante 2

Eine zweite Variante basiert auf der Verwendung von **tic**[3] und **toc**[4]. Um jedoch eine genaue Zeitmessung für GPUs zu erhalten, muss gewartet werden bis alle Operationen auf der GPU abgeschlossen sind, bevor toc aufgerufen werden darf. Dafür gibt es zwei Möglichkeiten.

- Es kann vor dem Aufruf von toc die letzte GPU-Ausgabe aufgerufen werden. Das hat zur Folge, dass alle Berechnungen abgeschlossen werden müssen, bevor die Zeitmessung durchgeführt wird.
- Alternativ kann die Wartefunktion mit einem GPUDevice-Objekt als Eingabe verwendet werden (siehe Code 2.2).

[2] https://de.mathworks.com/help/distcomp/gputimeit.html
[3] https://de.mathworks.com/help/matlab/ref/tic.html
[4] https://de.mathworks.com/help/matlab/ref/toc.html

```
function GPU_performace
    gd = gpuDevice(1);    % waehle die erste GPU des Rechners

    tic;
    [l,u] = lu(A);
    wait(gd);    % Wichtig: auf das Ende der Berechnung warten!
    tLU = toc;
    disp(['Zeit von Variante 2: ', num2str(tLU), 's']);

    clearvars         % alle Variablen loeschen
    gpuDevice([]);   % GPU-Speicher leeren
end
```

**Code 2.2:** GPU-Performance mit MATLAB® messen. (Variante 2)

## 2.3 GPU-Performance bestimmen

GPUs können verwendet werden, um bestimmte Arten von Berechnungen zu beschleunigen. Da jedoch die GPU-Leistung stark zwischen den verschiedenen GPU-Geräten variiert, werden im folgenden drei Tests[5] vorgestellt. Sie können genutzt werden, um eine GPU in puncto Leistung zu quantifizieren:

1. Wie schnell können Daten an die GPU gesendet oder von ihr zurückgeholt werden?

2. Wie schnell kann die GPU Daten lesen und schreiben?

3. Wie schnell kann der Grafikprozessor Berechnungen durchführen?

### 2.3.1 GPU-Bandbreite testen

Mit Hilfe eines Bandbreiten-Test kann gemessen werden, wie schnell Daten an die GPU gesendet und von ihr gelesen werden können. Da die GPU in den PCI-Bus eingesteckt ist, hängt dies stark davon ab, wie schnell der PCI-Bus ist und wie viele andere Geräte ihn nutzen. Es gibt aber auch einige Overheads, die in den Messungen enthalten sind, insbesondere der Funktionsaufruf-Overhead und die Array-Zuordnungszeit. Da diese in jeder "realen" Nutzung des Grafikprozessors vorhanden sind, ist es sinnvoll, diese mit einzubeziehen.

Der Code 2.3 (ebenfalls unter `code/GPU-performance/HostGPUBandwidth.m` zu finden) zeigt den Bandbreiten-Test.

[5]Quelle: https://de.mathworks.com/help/distcomp/examples/measuring-gpu-performance.html

```
% Setup
gpu = gpuDevice();
fprintf('Using a %s GPU.\n', gpu.Name)
sizeOfDouble = 8; % Each double-precision number needs 8 bytes of
    storage
sizes = power(2, 14:28);

% Testing host/GPU bandwidth
sendTimes = inf(size(sizes));
gatherTimes = inf(size(sizes));
for ii=1:numel(sizes)
    numElements = sizes(ii)/sizeOfDouble;
    hostData = randi([0 9], numElements, 1);
    gpuData = randi([0 9], numElements, 1, 'gpuArray');
    % Time sending to GPU
    sendFcn = @() gpuArray(hostData);
    sendTimes(ii) = gputimeit(sendFcn);
    % Time gathering back from GPU
    gatherFcn = @() gather(gpuData);
    gatherTimes(ii) = gputimeit(gatherFcn);
end

% Determine result
sendBandwidth = (sizes./sendTimes)/1e9;
[maxSendBandwidth,maxSendIdx] = max(sendBandwidth);
fprintf('Achieved peak send speed of %g GB/s\n',maxSendBandwidth)
gatherBandwidth = (sizes./gatherTimes)/1e9;
[maxGatherBandwidth,maxGatherIdx] = max(gatherBandwidth);
fprintf('Achieved peak gather speed of %g GB/s\n',
    max(gatherBandwidth))
```

**Code 2.3:** Test zur Bestimmung der GPU-Bandbreite in MATLAB®.

Das Resultat für die ls4gpu2-Workstation sieht beispielsweise wie folgt aus:

```
Using a Quadro P6000 GPU.
Achieved peak send speed of 10.674 GB/s
Achieved peak gather speed of 3.86652 GB/s
```

Und grafisch:

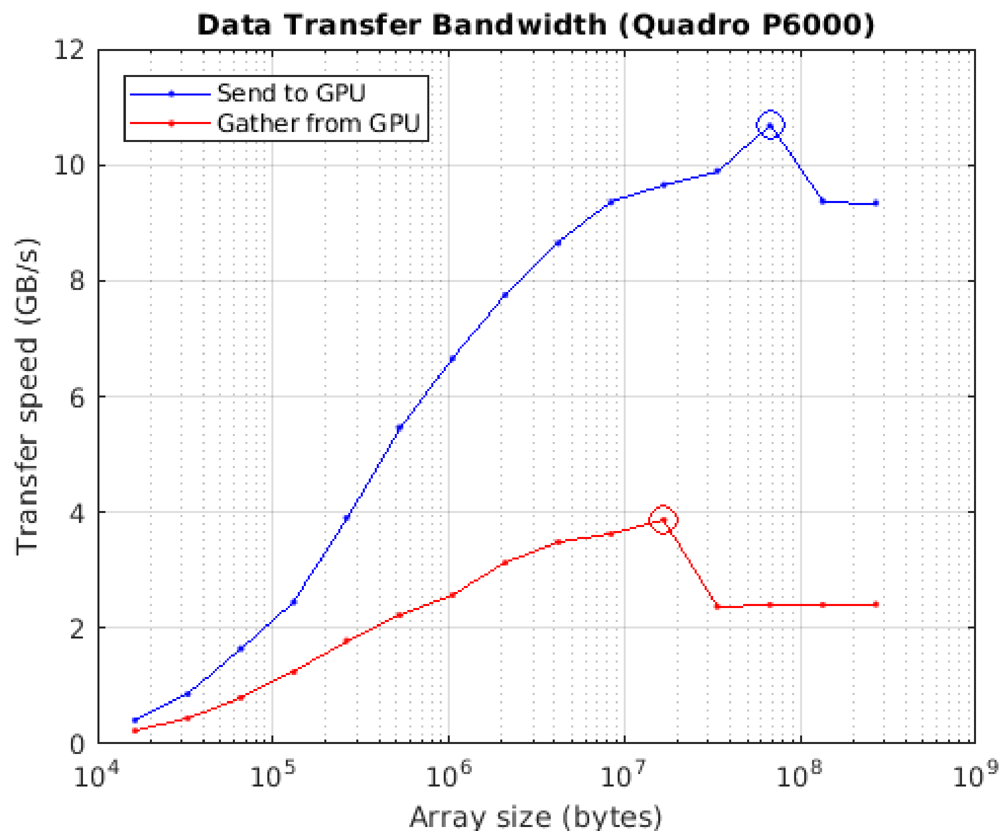


**Abbildung 2.1:** Send- und Gather-Performance der GPU von der ls4gpu2-Workstation.

### 2.3.2 Testen von speicherintensiven Operationen

Viele Operationen rechnen mit jedem Element eines Arrays sehr wenig und werden daher von der Zeit dominiert, die benötigt wird um Daten aus dem Speicher zu holen oder zurückzuschreiben. Selbst einfache MATLAB®-Operatoren wie *plus*, *minus* und *mtimes* berechnen pro Element so wenig, dass ihre Performance an die Speicherzugriffsgeschwindigkeit geknüpft ist.

Die MATLAB®-Operation *plus*[6] führt für jede Gleitkommaoperation einen Speicherzugriff für das Lesen und einen Speicherzugriff zum Schreiben durch. Sie sollte daher durch die Speicherzugriffsgeschwindigkeit begrenzt werden und ist ein guter Indikator für die Geschwindigkeit eines Schreib-/Lesevorgangs.

Der Code 2.4 (ebenfalls unter `code/GPU-performance/MemoryIntensiveOperations.m` zu finden) zeigt den Test zur Bestimmung der Schreib-/Lesegeschwindigkeit auf Basis der MATLAB®-Operation *plus*.

[6] https://de.mathworks.com/help/matlab/ref/plus.html

```
% Setup
gpu = gpuDevice();
fprintf('Using a %s GPU.\n', gpu.Name)
sizeOfDouble = 8; % Each double-precision number needs 8 bytes of
    storage
sizes = power(2, 14:28);

% Testing memory intensive operations (GPU)
memoryTimesGPU = inf(size(sizes));
for ii=1:numel(sizes)
    numElements = sizes(ii)/sizeOfDouble;
    gpuData = randi([0 9], numElements, 1, 'gpuArray');
    plusFcn = @() plus(gpuData, 1.0);
    memoryTimesGPU(ii) = gputimeit(plusFcn);
end
memoryBandwidthGPU = 2*(sizes./memoryTimesGPU)/1e9;
[maxBWGPU, maxBWIdxGPU] = max(memoryBandwidthGPU);
fprintf('Achieved peak read+write speed on the GPU: %g
    GB/s\n',maxBWGPU)

% Testing memory intensive operations (CPU)
memoryTimesHost = inf(size(sizes));
for ii=1:numel(sizes)
    numElements = sizes(ii)/sizeOfDouble;
    hostData = randi([0 9], numElements, 1);
    plusFcn = @() plus(hostData, 1.0);
    memoryTimesHost(ii) = timeit(plusFcn);
end
memoryBandwidthHost = 2*(sizes./memoryTimesHost)/1e9;
[maxBWHost, maxBWIdxHost] = max(memoryBandwidthHost);
fprintf('Achieved peak read+write speed on the host: %g
    GB/s\n',maxBWHost)
```

**Code 2.4:** Test zur Bestimmung der Schreib-/Lesegeschwindigkeit auf Basis der MATLAB®-Operation *plus*.

Das Resultat für die ls4gpu2-Workstation sieht beispielsweise wie folgt aus:

```
Using a Quadro P6000 GPU.
Achieved peak read+write speed on the GPU:  371.035 GB/s
Achieved peak read+write speed on the host: 196.625 GB/s
```

Und grafisch:

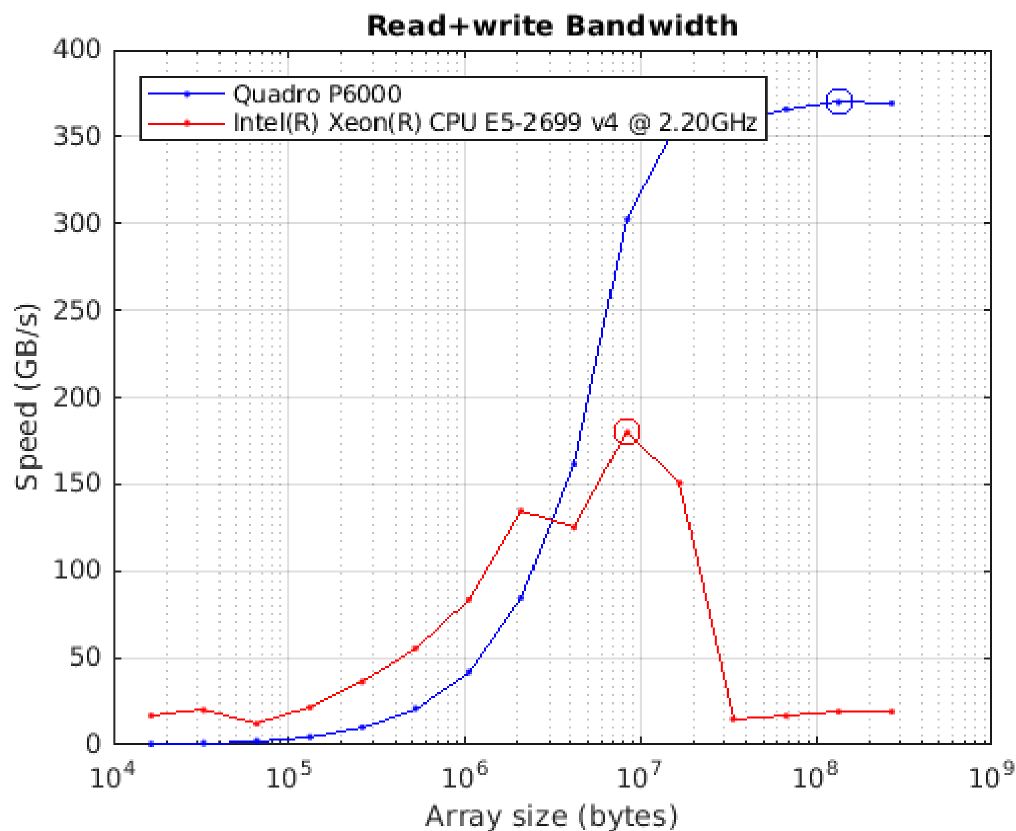


**Abbildung 2.2:** Schreib-/Lesegeschwindigkeit der ls4gpu2-Workstation.

Vergleicht man dieses Diagramm mit dem obigen Datenübertragungsdiagramm (siehe Abbildung 2.1), so wird deutlich, dass GPUs typischerweise viel schneller aus ihrem Speicher lesen und schreiben können, als sie Daten vom Host erhalten können. Es ist daher wichtig, die Anzahl der Host-GPU- oder GPU-Host-Speicherübertragungen zu minimieren. Im Idealfall sollten Programme die Daten auf die GPU übertragen, dann so viel wie möglich damit machen und sie erst dann zum Host zurückschicken, wenn sie fertig sind. Noch besser wäre es, die Daten zunächst auf der GPU zu erstellen.

### 2.3.3 Rechenleistung bestimmen (FLOPS)

Wenn die Anzahl an Gleitkommazahl-Operationen pro Element (sehr) hoch ist, ist die Speichergeschwindigkeit weit weniger wichtig. Ein guter Test um die Rechenleistung eines Systems zu bestimmen, ist die Matrixmultiplikation. Für die Multiplikation zweier Matrizen beträgt die Gesamtzahl der Gleitkommazahl-Operationen:

$$FLOPS(N) = 2N^3 - N^2.$$

Insgesamt werden zwei Eingabematrizen ($A$ und $B$) gelesen und eine daraus resultierende Matrix geschrieben. Dies ergibt eine Rechendichte von $(2N-1)/3$ FLOP/Element (FLOP = *Floating Point Operations*). Im Kontrast dazu steht das im Abschnitt 2.3.2 verwendete *plus*, das eine Rechendichte von nur 1/2 FLOP/Element hat.

Der Code 2.5 (ebenfalls unter `code/GPU-performance/ComputationallyIntensiveOperations.m` zu finden) zeigt einen Test zur Bestimmung der FLOPS (*Floating Point Operations Per Second*) für die GPU sowie Host/CPU.

```
1  % Setup
2  gpu = gpuDevice();
3  fprintf('Using a %s GPU.\n', gpu.Name)
4  
5  % Measure FLOPS (CPU and GPU)
6  sizes = power(2, 12:2:26);
7  N = sqrt(sizes);
8  mmTimesHost = inf(size(sizes));
9  mmTimesGPU = inf(size(sizes));
10 for ii=1:numel(sizes)
11     % First do it on the host
12     A = rand( N(ii), N(ii), 'double' ); % for single precision
13     B = rand( N(ii), N(ii), 'double' ); % use 'single'
14     mmTimesHost(ii) = timeit(@() A*B);
15     % Now on the GPU
16     A = gpuArray(A);
17     B = gpuArray(B);
18     mmTimesGPU(ii) = gputimeit(@() A*B);
19 end
20 mmGFlopsHost = (2*N.^3 - N.^2)./mmTimesHost/1e9;
21 [maxGFlopsHost,maxGFlopsHostIdx] = max(mmGFlopsHost);
22 mmGFlopsGPU = (2*N.^3 - N.^2)./mmTimesGPU/1e9;
23 [maxGFlopsGPU,maxGFlopsGPUIdx] = max(mmGFlopsGPU);
24 
25 % Print results
26 [status,cpuName] = system('grep -m 1 "model name" /proc/cpuinfo |
       cut -d: -f2');
27 fprintf(['Achieved peak calculation for double precision of:\n',
       ...
28     '\t%1.1f GFLOPS (%s)\n\t%1.1f GFLOPS (%s)\n'], ...
29     maxGFlopsHost, strtrim(cpuName), maxGFlopsGPU, gpu.Name)
```

**Code 2.5:** Test zur Bestimmung der FLOPS von GPU sowie Host/CPU in MATLAB®.

Das Resultat für die ls4gpu2-Workstation sieht grafisch wie folgt aus:

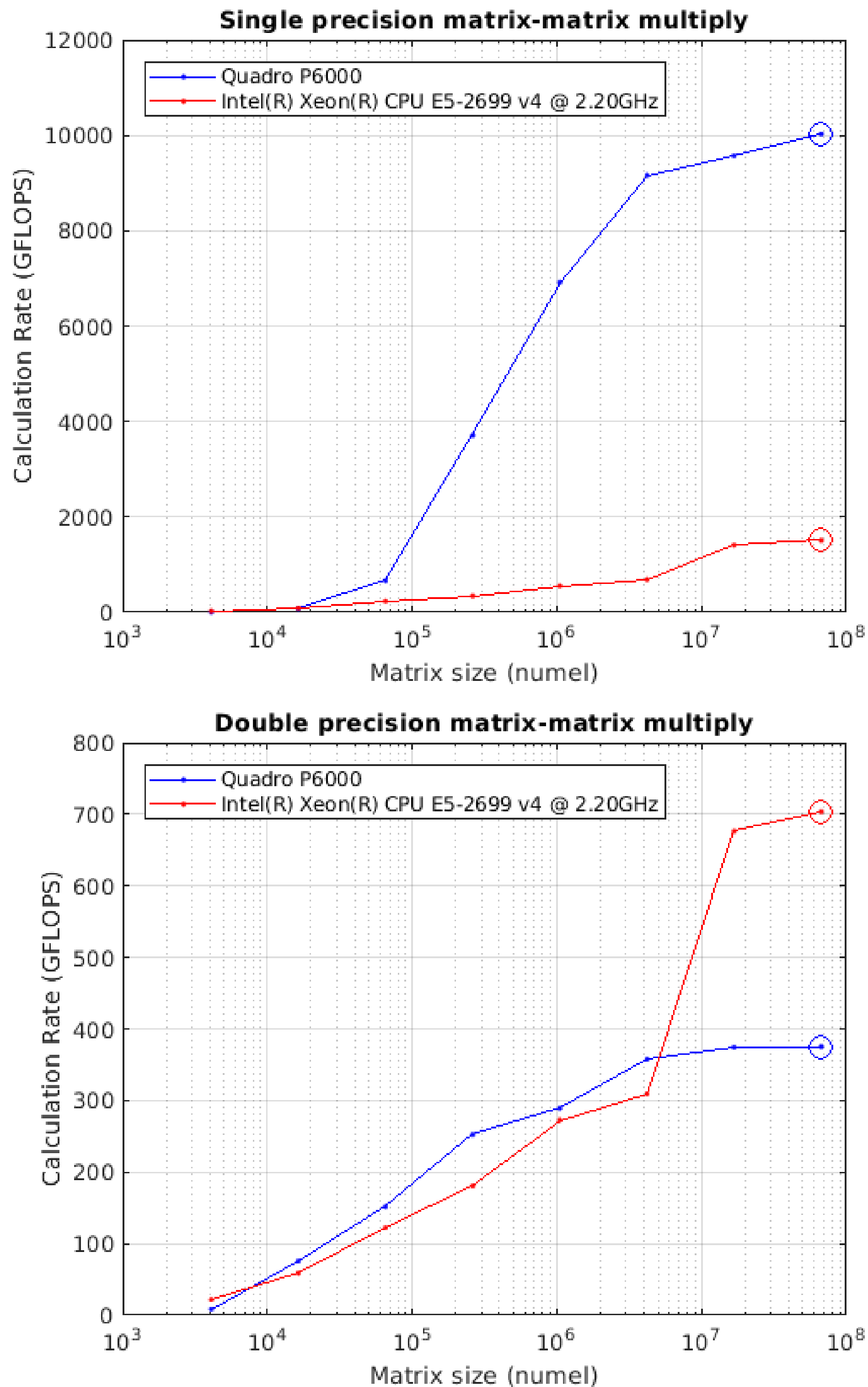


**Abbildung 2.3:** GFLOPS (*Giga*FLOPS = $10^9$ FLOPS) der ls4gpu2-Workstation für einfache und doppelte Genauigkeit.

Und die textuelle Ausgabe für einfache und doppelte Genauigkeit:

```
Using a Quadro P6000 GPU.
Achieved peak calculation rates for single precision of:
      1509.9 GFLOPS (Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz)
     10029.3 GFLOPS (Quadro P6000)


Using a Quadro P6000 GPU.
Achieved peak calculation rates for double precision of:
     703.7 GFLOPS (Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz)
     375.4 GFLOPS (Quadro P6000)
```

# Kapitel 3

# GPUs und CPU in MATLAB® parallel verwenden

In diesem Kapitel wird ein Beispielprogramm für die parallele Nutzung von allen GPUs und der CPU in einem Rechner gezeigt. Dabei berechnen sowohl die GPUs als auch die CPU jeweils eine Matrixmultiplikation, wobei die Implementierung (Code 3.1) auch als MATLAB®-Skript unter `code/useGPUsAndCPUinParallel.m` zu finden ist.

## 3.1 Vorgehen

Um parallel Berechnungen auf den vorhanden GPUs und der CPU ausführen zu können, bietet MATLAB® die Möglichkeit einen **parpool**[1] mit $N$ Workern zu erzeugen. In unserem Fall wird ein parpool mit $gpuDeviceCount + 1$ erzeugt, wobei **gpuDeviceCount**[2] der Anzahl an GPUs im Rechner entspricht (siehe Codezeile 5 im Code 3.1). Anschließend startet der parallele Bereich, der mit dem Schlüsselwort **spmd**[3] (*Single Program, Multiple Data*) beginnt und mit *end* endet. In diesem Bereich können wir mit **labindex**[4] (wichtig: $0 <$ Worker-ID $<=$ Anzahl Worker) die einzelnen Worker (z.B. mit einem einfachem if-Statement) direkt ansprechen und ihnen eine Arbeit zuweisen. Hier unterteilen wir in *labindex == 1* und die Anderen (else). Der Worker mit der ID $= 1$ steuert die Berechnungen der CPU (siehe Codezeile 8 bis 18), während die restlichen Arbeiter die GPUs im Rechner mit Arbeit versorgen (siehe Codezeile 20 bis 33).

Abschließend sollte erwähnt werden, dass die Erzeugung eines parpools mit N Workern etwas Zeit in Anspruch nimmt. Eine Anzeige in der unteren linken Ecke des MATLAB®-Fenster zeigt jeweils den aktuellen Status des parpools (siehe Abbildung 3.1).

---

[1] https://de.mathworks.com/help/distcomp/parpool.html
[2] https://de.mathworks.com/help/distcomp/gpudevicecount.html
[3] https://de.mathworks.com/help/distcomp/spmd.html
[4] https://de.mathworks.com/help/distcomp/labindex.html

```
function useGPUsAndCPUinParallel
    matrixSize = 2^14;  % Maximum fuer CPU-Speicher, ca. 1/3 des GPU-Speichers (ls4gpu2-Workstation)
    disp(sprintf('Multiplikation von zwei %dx%d-Matrizen', matrixSize, matrixSize));

    parpool(gpuDeviceCount + 1);  % Parpool mit n Workern erzeugen, fuer (n-1)x GPU- und 1x CPU-Berechnung

    spmd
        if labindex == 1    % CPU
            disp([num2str(labindex), ': using CPU']);

            % weitere Berechnungen durchfuehren (hier: Matrixmul.)
            tic;
            A_cpu = rand(matrixSize);
            B_cpu = rand(matrixSize);
            C_cpu = A_cpu * B_cpu;
            elapsedTime = toc;

            disp(['Calculation completed. Size: ', num2str(size(C_cpu)), ', Time: ', num2str(elapsedTime), 's']);

        else              % GPUs
            % GPU auswaehlen
            gd = gpuDevice(labindex - 1);
            disp([num2str(labindex), ': using GPU ', num2str(gd.Index), ' (', gd.Name, ')']);

            % weitere Berechnungen durchfuehren (hier: Matrixmul.)
            tic;
            A_gpu = gpuArray(rand(matrixSize));
            B_gpu = gpuArray(rand(matrixSize));
            C_gpu = A_gpu * B_gpu;
            wait(gd);   % auf das Ende der Berechnung warten!
            elapsedTime = toc;

            disp(['Calculation completed. Size: ', num2str(size(C_gpu)), ', Time: ', num2str(elapsedTime), 's']);
        end
    end
    clearvars   % alle Variablen loeschen
    delete(gcp) % Parpool ausschalten und loeschen
end
```

**Code 3.1:** Parallele Nutzung von CPU und GPUs in einem Rechner auf Basis von MATLAB®.

**Abbildung 3.1:** Die parpool-Statusanzeige in der unteren linken Ecke des MATLAB®-Fensters zeigt die Verbindung der aktuellen Clientsitzung zum parpool und den parpool-Status an. Mit einem Klick auf das Symbol erscheint ein Menü mit den unterstützten parpool-Aktionen.

## 3.2 Zeitmessung

Der Code 3.1 wurde auf der Workstation ls4gpu2 mit MATLAB® R2017b ausgeführt. Die Ausführungszeiten, bei der Multiplikation von zwei Matrizen der Größe $2^{14}$, sind wie folgt:

- CPU: $\sim 185,2s$
- beide GPUs: $\sim 32,5s$

Weitere Beispiele für parallele Berechnungen mit MATLAB® sind unter dem folgenden Link zu finden: `https://de.mathworks.com/help/distcomp/examples.html`

# Anhang A

# ls4gpu2 am Lehrstuhl 4

Der Lehrstuhl 4 der Fakultät für Informatik TU Dortmund verfügt über Computer, mit rechenstarken Grafikkarten.

Die Workstation `ls4gpu2` ist mit einer Intel® Xeon® E5-2699 v4 CPU und zwei NVIDIA® Quadro® P6000 Grafikkarten ausgestattet. Außerdem sind 64GB RAM enthalten. Erreichbar ist der Server mit dem Befehl `ssh ls4gpu2.cs.tu-dortmund.de`.

Auf dem ls4gpu2 ist MATLAB® im Verzeichnis: */app/unido-i04linux/* zu finden. Aktuell sind die folgenden Versionen von MATLAB® installiert: matlab2009b, matlab2012b, matlab2015b und matlab2017b. Alle Matlab-Skripte wurden mit der Version matlab2017b getestet.

## A.1 NVIDIA® Quadro® P6000

In der Tabelle A.1 sind die wichtigsten Informationen zur Grafikkarte NVIDIA® Quadro® P6000 zu finden.

| **General** | |
|---|---|
| NVIDIA CUDA® Cores | 3840 |
| Compute Capability | 6.1 |
| **Maximums** | |
| Threads per Block | 1024 |
| Threads per Multiprocessor | 2048 |
| Shared Memory per Block | 48 KiB |
| Shared Memory per Multiprocessor | 96 KiB |
| Registers per Block | 65536 |
| Registers per Multiprocessor | 65536 |

| | |
|---|---|
| Grid Dimensions | [2147483647, 65536, 65536] |
| Block Dimensions | [1024, 1024, 64] |
| Warps per Multiprocessor | 64 |
| Blocks per Multiprocessor | 32 |
| Half Precision FLOP/s | 98,7 GigaFLOP/s |
| Single Precision FLOP/s | 12,634 TeraFLOP/s |
| Double Precision FLOP/s | 394,8 GigaFLOP/s |
| **Multiprocessor** | |
| Multiprocessor | 30 |
| Clock Rate | 1,645 GHz |
| Concurrent Kernel | true |
| Max IPC | 6 |
| Threads per Warp | 32 |
| **Memory** | |
| Global Memory Bandwidth | 433,248 GB/s |
| Global Memory Size | 23,871 GiB |
| Constant Memory Size | 64 KiB |
| L2 Cache Size | 3 MiB |
| Memcpy Engines | 2 |
| **PCIe** | |
| Generation | 3 |
| Link Rate | 8 Gbit/s |
| Link Width | 16 |

**Tabelle A.1:** NVIDIA® Quadro® P6000 Spezifikation. (Quelle: NVIDIA® Visual Profiler)

## A.2 Intel® Xeon® E5-2699 v4

In der Tabelle A.2 sind die wichtigsten Informationen zum Prozessor Intel® Xeon® E5-2699 v4 zu finden.

| **Performance** | |
|---|---|
| Number of Cores | 22 |
| Number of Threads | 44 |
| Processor Base Frequency | 2.20 GHz |
| Max Turbo Frequency | 3,60 Ghz |

| Cache | 55 MB SmartCache |
|---|---|
| Bus Speed | 9,6 GT/s QPI |
| Number of QPI Links | 2 |
| TDP | 145 W |
| **Memory Specifications** | |
| Max Memory Size | 1,54 TB |
| Memory Types | DDR4 1600/1866/2133/2400 |
| Max Number of Memory Channels | 4 |
| Max Memory Bandwidth | 76,8 GB/s |
| Physical Address Extensions | 46-bit |
| ECC Memory Supported | Yes |
| **Expansion Options** | |
| Scalability | 2S |
| PCI Express Revision | 3.0 |
| PCI Express Configurations | x4, x8, x16 |
| Max Number of PCI Express Lanes | 40 |

**Tabelle A.2:** Intel® Xeon® E5-2699 v4 Spezifikation. (Quelle: https://ark.intel.com/en/products/91317/Intel-Xeon-Processor-E5-2699-v4-55M-Cache-2_20-GHz)