Maximierung der Codeperformance durch Optimierung des Speicherzugriffs

VON STUART MCGARRITY

Die meisten MATLAB-Anwender möchten, dass ihre Programme schnell sind – vor allem, wenn sie große Datensätze verarbeiten. Da die Leistung des Hauptspeichers aber nicht in gleichem Maße gewachsen ist wie die CPU-Leistung, ist der begrenzende Faktor für die Ausführungsgeschwindigkeit heute oft die Zugriffszeit auf den Speicher.

Wenn man aber ein wenig mit der Art und Weise vertraut ist, wie MATLAB Daten speichert und wieder einliest, kann man ineffiziente Formen der Speichernutzung vermeiden und damit die Ausführungsgeschwindigkeit verbessern.

Dieser Artikel stellt drei Möglichkeiten zur Verbesserung der Leistung von Programmcode vor, dessen Geschwindigkeit durch den Speicher begrenzt wird:

- Reservierung von Speicherbereichen für Datenfelder, bevor auf diese in Schleifen zugegriffen wird.
- Daten in Spaltenform speichern und abrufen.
- Vermeidung unnötiger Variablen.

In allen drei Fällen vergleichen wir die Ausführungsgeschwindigkeit eines Code-Segments vor und nach der Anwendung der jeweiligen Methode. Um die optimale Leistung zu erzielen, werden die Laufzeiten an Funktions-M-Files gemessen, nicht an Skript-M-Files. Da die Speicherperformance rechnerabhängig ist, wurden zudem alle Tests auf zwei verschiedenen Computern durchgeführt[1].

Reservierung von Speicherbereichen vor dem Zugriff auf diese in Schleifen

Wenn Datenfelder innerhalb von Schleifen erzeugt oder wiederholt verändert werden, sollte der für sie benötigte Speicherbereich immer im Voraus reserviert werden. Von den drei vorgestellten Methoden erzielt diese den größten Leistungsgewinn. In den Code-Segmenten in Abbildung 1 wird in einer `for`-Schleife eine Zahlenreihe nach der Formel *x(k) = 1.05* x(k-1)* berechnet, was der Entwicklung eines Kontostandes bei einem Jahreszins von 5% entspricht.

Code-Segment 1

```
N = 10e3;

x(1) = 1000;
for k = 2:N
    x(k) = 1.05*x(k-1);
end
```

Machine A = 0.1409 sec
Machine B = 0.1281 sec

Code-Segment 2

```
N = 10e3;
x = zeros(N,1); %Preal-
locate
x(1) = 1000;
for k = 2:N
    x(k) = 1.05*x(k-1);
end
```

Machine A = 0.00024 sec
Machine B = 0.00027 sec

Abb. 1: Reservierung von Speicherbereichen. Code-Segment 2 benötigt auf Rechner A 99,8% weniger Zeit als Segment 1 (580 Mal schneller) und 99,7% weniger auf Rechner B (475 Mal schneller).

Warum Code-Segment 2 schneller ist

In der MATLAB-Sprache müssen Variablentypen und -größen nicht ausdrücklich deklariert werden. Man kann darum ein Array ganz einfach durch Neuindexierung vergrößern. Das ist zwar bequem, wenn man gerade ein neues Programm entwirft, aber MATLAB muss bei jedem Durchlauf dieser Prozedur ein größeres Array reservieren, um die Daten unterbringen zu können. Wenn das wiederholt geschieht, etwa in einer Schleife, ist der Programmcode langsam und ineffizient. Code-Segment 2 dagegen reserviert das gesamte Array x bis zu seiner maximalen Größe im Voraus. Während der gesamten Ausführung wird damit nur einmal ein Speicherbe-

1 *Rechner A ist ein Lenovo T60 ThinkPad mit 1.83GHz Intel Core Duo T2400-Prozessor, 2MB L2 Cache, 2GB RAM und 32-Bit Windows XP. Rechner B ist mit zwei 2.1GHz AMD Opteron 248-CPUs, 1MB L2 Cache und 1GB RAM ausgestattet und läuft mit einem 64-Bit-Linux.*

Code-Segment 1

```
N = 2e3;
x = randn(N);
y = zeros(N);

for r = 1:N % Row
    for c = 1:N % Column
        if x(r, c)> = 0
            y(r,c) =
x(r, c);
        end
    end
end
```

Machine A = 0.1605 sec
Machine B = 0.2090 sec

Code-Segment 2

```
N = 2e3;
x = randn(N);
y = zeros(N);

for c = 1:N % Column
    for r = 1:N % Row
        if x(r, c)> = 0
            y(r,c) = x(r,
c);
        end
    end
end
```

Machine A = 0.1083 sec
Machine B = 0.0946 sec

Abb. 2: Daten in Spaltenform speichern und abrufen. Code-Segment 2 benötigt auf Rechner A 33% weniger Ausführungszeit als Segment 1, auf Rechner B 55% weniger.

reich reserviert. Der M-Lint Code Checker gibt darum eine Warnmeldung aus, wenn er eine Möglichkeit zur Vorabreservierung von Speicher entdeckt.

Daten in Spaltenform speichern und abrufen

Bei der Verarbeitung zwei- oder n-dimensionaler Arrays sollte man die Daten in Spaltenform speichern und auf sie zugreifen. Die Code-Segmente in Abbildung 2 kopieren die positiven Werte des zweidimensionalen Arrays `x` in ein neues Array `y`.

Warum Code-Segment 2 schneller ist

Moderne CPUs verfügen über einen schnellen Cache, der die mittlere Zugriffszeit auf den Hauptspeicher verkürzt. Die beste Cache-Effizienz erreicht ein Programmcode, wenn er direkt aufeinander folgende Speicheradressen ausliest. Da MATLAB die Spalten von Matrizen in solchen monoton aufsteigenden Speicheradressen ablegt, nutzt die spaltenweise Verarbeitung von Daten den Cache am effizientesten.

Code-Segment 2 ist schneller, weil es die Elemente des zweidimensionalen Arrays zunächst spaltenweise in der inneren Schleife durchläuft. Wenn es der Algorithmus erlaubt, kann man die Cache-Effizienz außerdem steigern, indem man Einzelindizes verwendet, etwa *x(k)* statt *x(r,c)*.

Cache-Effekte hängen allerdings stark vom Rechner und von der Datengröße ab, weshalb die Leistung hier schwer vorhersagbar ist. Wenn man mit Zeilenvektoren arbeitet, beeinflusst beispielsweise die Spaltenlänge die Speicherzugriffszeit, weil sie den Abstand (den

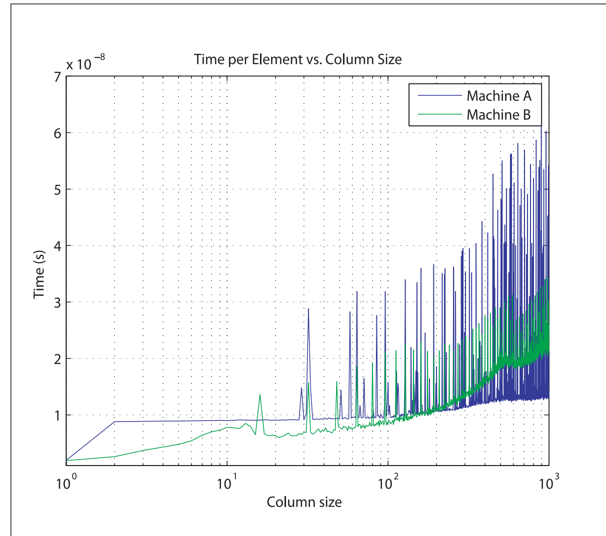

Abb. 3: Laufzeit je Element über die Spaltenanzahl bei der Multiplikation mit einem Zeilenvektor mit 10.000 Elementen.

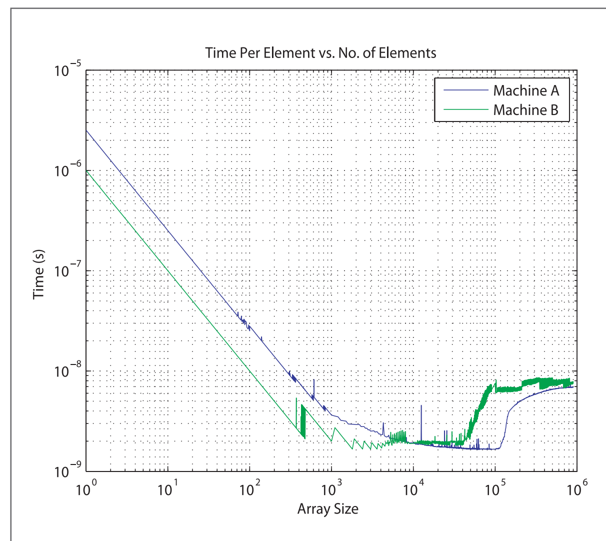

Abb. 4: Laufzeit je Element über der Anzahl der Elemente bei der elementweisen Multiplikation.

„Stride“) zwischen zwei nacheinander verwendeten Speicheradressen bestimmt. Abbildung 3 zeigt, wie sich die je Element benötigte Rechenzeit ändert, wenn man einen Zeilenvektor mit 10.000 Elementen elementweise mit einer Matrix mit veränderlicher Spaltenlänge multipliziert. Am effizientesten ist die Operation für eine Spaltengröße von 1, also einen eindimensionalen Vektor. Bei anderen Spaltengrößen kann der Code um bis zu eine Zehnerpotenz langsamer laufen.

Sogar die Größe des Arrays beeinflusst die Effizienz. Abbildung 4 zeigt die auf beiden Rechnern je Element benötigte Zeit für die Multiplikation eines eindimensionalen Arrays in Abhängigkeit

von dessen Größe. Je nach den Eigenheiten von Cache und Hauptspeicher eines Rechners ist die Verarbeitung großer Arrays um Größenordnungen effizienter als die kleiner Arrays.

In einigen Fällen kann die Verarbeitungszeit von Arrays nur wenig unterschiedlicher Größe sogar erheblich schwanken und es ist durchaus möglich, dass eine auf einem Rechner funktionierende Optimierung für eine bestimmte Problemgröße auf einem anderen Rechner völlig andere Ergebnisse liefert.

Vermeidung unnötiger Variablen

Vor der Erzeugung neuer Variablen oder von Variablen, die Funktionen bereits vorhandener Datensätze sind, sollte man sicherstellen, dass sie für den Algorithmus erforderlich sind. Bei großen Datensätzen ist das besonders wichtig.

Die Code-Segmente in Abbildung 5a und 5b multiplizieren die Matrix x mit einem Skalar. In 5a wird dazu jeweils ein Operator verwendet (Segmente 1 und 2), in 5b dagegen eine M-Funktion (Segmente 3 und 4).

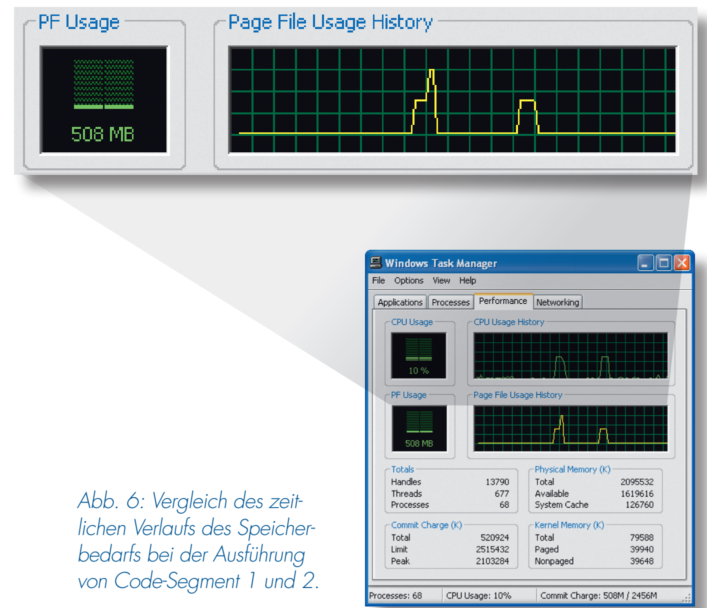

Abb. 6: Vergleich des zeitlichen Verlaufs des Speicherbedarfs bei der Ausführung von Code-Segment 1 und 2.

Code-Segment1

```
N = 3e3;
x = randn(N);
y = x*1.2;
```

Machine A = 0.0847 sec
Machine B = 0.1115 sec

Code-Segment 2

```
N = 3e3;
x = randn(N);
x = x*1.2; % In place
```

Machine A = 0.0506 sec
Machine B = 0.00461 sec

Abb. 5a: Vermeidung unnötiger Variablen durch In-Place-Aufruf von Operatoren. Code-Segment 2 wird auf Rechner A in 40% kürzerer Zeit ausgeführt als Code-Segment 1 und auf Rechner B in 57% kürzerer Zeit.

Code-Segment 3

```
y = myfun(x);

function y = myfun(x)
y = 1.2*x;
```

Machine A = 0.0858 sec
Machine B = 0.1019 sec

Code-Segment4

```
x = myfun_ip(x);

function x = myfun_ip(x)
x = 1.2*x; % In place
```

Machine A = 0.0508 sec
Machine B = 0.0421sec

Abb. 5b: Vermeidung unnötiger Variablen durch Erzeugung von In-Place-Aufrufen in M-Funktionen. Code-Segment 4 wird auf Rechner A in 40% kürzerer Zeit ausgeführt als Segment 3, auf Rechner B benötigt es 59% weniger Zeit.

Warum die Code-Segmente 2 und 4 schneller sind

In MATLAB kann man leicht unbeabsichtigte Kopien seiner Daten anlegen. Bei großen Datensätzen wird dadurch zum einen viel Speicher verbraucht, zum anderen kann der Reservierungs- und Kopiervorgang selbst relativ viel Zeit in Anspruch nehmen.

Die Code-Segmente 2 und 4 sind schneller, weil sie In-Place-Operatoren nutzen und so die Erzeugung neuer Variablen verhindern, die lediglich veränderte Versionen bereits vorhandener Variablen wären. Diese Fähigkeit ist verfügbar für elementweise Operatoren (etwa `.*`,`+`), einige MATLAB-Funktionen wie `sin` und `sqrt` sowie für selbstgeschriebene M-Funktionen. Um ein Funktions-M-File zu erzeugen, das In-Place aufgerufen werden kann, muss das Ausgabeargument mit einem der Eingabeargumente identisch sein. Die mit MATLAB 7.3 eingeführte Fähigkeit, Funktionen In-Place aufzurufen, ist nur verfügbar, wenn die Funktion selbst aus einem Funktions-M-File aufgerufen wird, nicht jedoch durch ein Skript.

Wie bereits erwähnt benötigen In-Place-Operationen weniger Speicher und weniger Rechenzeit. In Abbildung 6 ist der im Windows Task Manager angezeigte zeitliche Verlauf der Speicherbelegung bei der Verarbeitung eines Arrays aus 100 Millionen Elementen mit Code-Segment 1 und 2 dargestellt (um den unterschiedlichen Speicherbedarf besser darstellen zu können, wurde die Berechnung Schritt für Schritt per Hand im Debugger durchgeführt).

Eine geringere Speicherauslastung ist vor allem bei der Arbeit mit Datensätzen wichtig, die größer sind als der vorhandene Hauptspeicher. Betriebssysteme ergänzen nämlich den physischen Speicher bei Bedarf durch eine Auslagerungsdatei, die auf der Festplatte gespeichert wird. Der Zugriff darauf ist um Größenordnungen langsamer als der auf den RAM. Wenn Sie also Wert auf Geschwindigkeit legen, sollten Sie beim Arbeiten mit MATLAB immer darauf achten, dass der Speicherbedarf einer Anwendung den verfügbaren physischen Hauptspeicher nicht übersteigt. ◀◀

Glossar zur Speicherperformance

Cache. Schneller Speicher im Prozessor oder auf dem Mainboard. Verbessert die Leistung des Hauptspeichers durch Zwischenspeicherung von Daten.

RAM. Physikalischer Hauptspeicher, bei 32-Bit-Betriebssystemen meist zwischen 1 GB und 4 GB.

Speicherlimitierter Code. Programmcode, dessen Leistung nicht durch die CPU-Geschwindigkeit, sondern durch die Speichergeschwindigkeit begrenzt wird.

Stride. Abstand zwischen den Speicheradressen zweier nacheinander abgerufener Zeilenelemente einer Matrix.

Vektorisierter Programmcode. Programmcode, der mit Arrays arbeitet und darum keine `for`-Schleifen benötigt.

Weitere Information

- Download der Code-Segmente
  www.mathworks.de/res/code_segments
- Leitfaden zur Speicherverwaltung
  www.mathworks.de/res/memory

Quellen

WEBSITE
www.mathworks.de

TECHNISCHER SUPPORT
www.mathworks.de/support

ONLINE USER COMMUNITY
www.mathworks.de/matlabcentral

DEMOS
www.mathworks.de/products/demos

SCHULUNGEN
www.mathworks.de/training

PRODUKTE VON DRITTANBIETERN
www.mathworks.de/connections

WELTWEITE KONTAKTINFORMATION
www.mathworks.de/contact

E-MAIL
info@mathworks.de

©2007 The MathWorks, Inc. MATLAB, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, SimBiology, SimHydraulics sowie xPC TargetBox sind eingetragene Warenzeichen und SimEvents ist eine Handelsmarke von The MathWorks, Inc. Andere Produkt- oder Markennamen sind Handelsbezeichnungen oder eingetragene Warenzeichen der jeweiligen Eigentümer.

91469v00 06/07