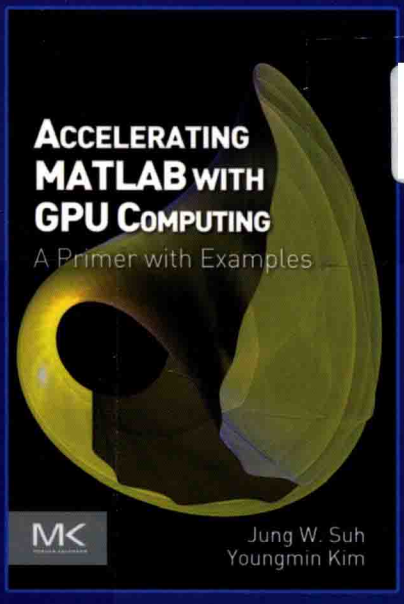
国际信息工程先进技术译丛

GPU与MATLAB混合编程

Accelerating MATLAB with GPU Computing: A Primer with Examples

[韩] 郑郁旭（Jung W. Suh）
金英民（Youngmin Kim） 著
熊磊 李丞 译



- 本书由数据密集计算领域的资深专家撰写
- 以应用实例为主线，提供大量的源代码
- 教你轻松掌握GPU与MATLAB混合编程

机械工业出版社
CHINA MACHINE PRESS

国际信息工程先进技术译丛

GPU与MATLAB混合编程

[韩] 郑郁旭（Jung W·Suh）
金英民（Young min kim） 编著
熊 磊 李 丞 译

机械工业出版社

本书介绍 CPU 和 MATLAB 的联合编程方法，包括不使用 GPU 实现 MATLAB 加速的方法；MATLAB 和计算统一设备架构（CUDA）配置通过分析进行最优规划，以及利用 c-mex 进行 CUDA 编程的方法；MATLAB 与并行计算工具箱和运用 CUDA 加速函数库的方法；计算机图形实例和 CUDA 转换实例。本书通过大量的实例、图示和代码，深入浅出地引导读者进入 GPU 的殿堂。通过阅读本书，读者可以轻松学习使用 GPU 进行并行处理，实现 MATLAB 代码的加速，提高工作效率，从而将更多的时间和精力用于创造性工作和其他事情。

本书可作为相关专业高年级本科生和研究生的教材，也可作为工程技术人员的参考书。

<Accelerating MATLAB with GPU Computing: A Primer with Examples >,
< Jung W.Suh , Youngmin Kim >
ISBN: 978-0-12-408080-5(ISBN of original edition)

北京市版权局著作权登记 图字：01-2015-3498

图书在版编目（CIP）数据

GPU 与 MATLAB 混合编程/（韩）郑郁旭，（韩）金英民编著；熊磊，李丞译．—北京：机械工业出版社，2016.1
（国际信息工程先进技术译丛）
书名原文：Accelerating MATLAB with GPU Computing: APrimer with Examples
ISBN 978-7-111-52904-0

Ⅰ.①G… Ⅱ.①郑… ②金… ③熊… ④李… Ⅲ.①图像处理-程序设计 Ⅳ.①TP391.41

中国版本图书馆 CIP 数据核字（2016）第 024726 号

机械工业出版社（北京市百万庄大街 22 号 邮政编码 100037）
责任编辑：李馨馨 责任校对：张艳霞
责任印制：常天培
北京机工印刷厂印刷（三河市南杨庄国丰装订厂装订）
2016 年 4 月第 1 版·第 1 次印刷
169mm×239mm·13.5 印张·262 千字
0 001—3 000 册
标准书号：ISBN 978-7-111-52904-0
定价：59.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
电话服务 网络服务
服务咨询热线：（010）88361066 机工官网：www.cmpbook.com
读者购书热线：（010）68326294 机工官博：weibo.com/cmp1952
（010）88379203 教育服务网：www.cmpedu.com
封面无防伪标均为盗版 金书网：www.golden-book.com

译 者 序

当前，在视频处理、电磁场分析、移动通信、生物信息、人工智能、医疗诊断、流体力学等诸多领域，密集计算的需求越来越旺盛。台式计算机 CPU 发展远远无法满足密集计算的要求。而采用大型工作站进行密集计算，不仅成本高昂，应用也十分不便。利用图形处理器（GPU）中众多的流处理器，实现并行计算，可以极大加速程序运行，是当前密集计算的重要方法。

但要熟练使用 GPU 实现程序加速，需要用户具有较好的编程能力，往往令人望而却步。而 MATLAB 功能强大，简单易用，应用非常广泛。本书聚焦于 GPU 和 MATLAB 的混合编程，采用实例教学的方式，并附上大量源代码，全书浅显易懂，以最小的学习成本，让读者掌握 GPU 加速 MATLAB 的方法，非常适合作为 GPU 入门读物。

本书第 1 章介绍了不使用 GPU 而直接实现 MATLAB 程序加速，读者得以初窥程序加速的基本方法。第 2 章介绍了使用 GPU 前需要的 MATLAB 和 CUDA 配置方法，并以二维卷积为例，详细介绍了 GPU 实现程序加速的流程。第 3 章介绍了多种时间分析工具，引领读者通过时间分析，发现程序运行的瓶颈。第 4 章介绍了利用 c-mex 进行 CUDA 编程的方法。第 5 章和第 6 章分别介绍了 MATLAB 并行计算工具箱和 CUDA 加速函数库的使用方法。第 7 章以计算机图形学中的 Marching Cubes 算法为例，详细介绍了 GPU 的开发方法。第 8 章介绍了如何将 MATLAB 程序转换为 CUDA 程序。最后这两章是作者多年来实际开发的经验之谈。

在本书即将出版之际，要感谢杨雪莲、刘玉龙、冯如、白璐等几位同学在本书翻译和校对中所做工作。本书还得到了中央高校基本科研业务费项目（2014JBZ021）、轨道交通控制与安全国家重点实验室（北京交通大学）自主课题（RCS2014ZZ03）和中科院无线传感网与通信重点实验室开放课题（2013005）的资助。

GPU 作为研究热点，很多术语没有统一的中文译名，虽经过仔细推敲，但译者水平有限，书中难免存在不妥之处，敬请读者不吝赐教。

本书提供大量的源代码，有需要的读者可登录机械工业出版社的网站（www.cmpbook.com），免费注册并登录后进入“图书展示”页面，搜索到本书页面，点击“相关下载”即可下载本书源代码。

熊 磊

轨道交通控制与安全国家重点实验室（北京交通大学）

前　言

MATLAB 是广泛应用于快速原型设计和算法开发的仿真工具，功能强大，简单易用。许多实验室和研究机构都迫切地希望 MATLAB 代码能够更快地运行，以满足大运算量项目的需要。由于 MATLAB 采用向量/矩阵的数据形式，适合于并行处理，因此采用图形处理单元（Graphics Processing Unit，GPU）对提升 MATLAB 运行速度大有裨益。

本书主要面向工程、科学、技术等专业领域，需要利用 MATLAB 进行海量数据处理的师生和科研人员。MATLAB 用户可能来自各个领域，不一定都具有丰富的程序开发经验。对于那些没有程序开发基础的读者，利用 GPU 加速 MATLAB 需要对他们的算法进行移植，会引入一些不必要的麻烦，甚至还需要设定环境。本书面向具有一定或较多 MATLAB 编程经验，但对 C 语言和计算机并行架构不是很了解的读者，以帮助读者将精力集中在他们的研究工作上，从而避免因使用 GPU 和 CUDA 而对 MATLAB 程序而非算法本身进行大量调整。

作为入门读物，本书从基础知识开始，首先介绍如何设置 MATLAB 运行 CUDA（在 Windows 和 Mac OSX），创建 c-mex 和 m 文件；接着引导读者进入专业级别的主题，如第三方 CUDA 库。本书还提供了许多修改用户 MATLAB 代码的实用方法，以更好地利用 GPU 强大的计算能力。

本书将指导读者使用 NVIDIA 的 GPU 显著提升 MATLAB 的运行速度。NVIDIA 的 CUDA 作为一种并行计算架构，最早用于计算机游戏设计，但由于其高效的大规模计算能力，在基础科学和工程领域也声誉日隆。通过本书，读者无需付出很多的精力和时间，就可以利用 GPU 的并行处理和丰富的 CUDA 科学库，实现 MATLAB 代码的加速，从而提升读者的科研工作水平。

本书第 5 章将使用 Mathworks 并行计算工具箱。虽然 Mathworks 并行计算工具箱是提升 MATLAB 速度的有效工具，但当前的版本在成为通用速度提升解决方案方面还是存在一定的局限，此外该工具箱还需要额外付费购买。特别是，由于并行计算工具箱的目标在于多核、多计算机和/或集群分布式计算，以及 GPU 处理，GPU 优化以提升用户代码运算速度，相对而言既受限于速度提升，又受限于所支持的 MATLAB 函数。此外，如果仅局限于 Mathworks 并行计算工具箱，就很难最大化利用丰富的 CDUA 库。本书第 5 章将介绍当前并行处理工具箱的功能与局限。实践证明，采用 c-mex 的 GPU 是普适性地提升速度的更好方法，而且在当前的环境中能够更为灵活地使用。

通过阅读本书，读者很快就能体会到 MATLAB 代码运行速度惊人的提升，而且通过使用开源 CUDA 资源，可以更好地进行科学研究。支持 Windows 和 Mac 操作系统也是本书的特点之一。

目　录

第 1 章　不使用 GPU 实现 MATLAB 加速

1.1　本章学习目标

本章主要内容为 MATLAB 加速的基本方法，即不使用 GPU 和 c-mex 的固有方法。在本章中，你可以了解到以下内容：

- 采用向量化实现并行处理。
- 采用预分配实现内存有效管理。
- 其他加速 MATLAB 代码的有效方法。
- 循序渐进地提升代码性能的实例。

1.2　向量化

MATLAB 将数据表示为向量/矩阵的形式，所以“向量化”有助于加速 MATLAB 代码的运行。向量化的关键在于尽量减少 for 循环的使用。

考虑以下两个具有相同功能的 .m 文件：

```
% nonVec1.m

clear all;
tic

A = 0:0.000001:10;
B = 0:0.000001:10;

Z = zeros(size(A));
y = 0;

for i = 1:10000001

    Z(i) = sin(0.5*A(i)) * exp(B(i)^2);
    y = y + Z(i);

end
toc
y
```

```
% Vec1.m

clear all;
tic

A = 0:0.000001:10;
B = 0:0.000001:10;

Z = zeros(size(A));
y = 0;

y = sin(0.5*A) * exp(B.^2)';

toc
y
```

左侧的 nonVec1.m 文件使用 for 循环求和，而右侧的 Vec1.m 文件则没有。

```
>> nonVec1
Elapsed time is 0.944395 seconds.

y =
  -1.3042e+48

>> Vec1
Elapsed time is 0.330786 seconds.

y =
  -1.3042e+48
```

两个程序计算结果相同，但程序 Vec1.m 的计算时间大约为程序 nonVec1.m 的三分之一。所以为了更好地实现向量化，在代码中应尽量使用元素运算或者向量/矩阵运算。

1.2.1 元素运算

当用于两个矩阵时，符号*表示矩阵乘法，而符号 .* 则表示矩阵中对应元素相乘。例如，令 x=[1 2 3]，v=[4 5 6]：

```
>> k = x .* v
k =
   4  10  18
```

还有许多其他的运算也可以按元素进行：

```
>> k = x .^2
k =
   1 4 9

>> k = x ./ v
k =
   0.2500  0.4000  0.5000
```

很多函数也支持按元素运算：

```
>> k = sqrt(x)
k =
   1.0000  1.4142  1.7321

>> k = sin(x)
k =
   0.8415  0.9093  0.1411

>> k = log(x)
k =
   0  0.6931  1.0986

>> k = abs(x)
k =
   1  2  3
```

甚至关系运算符也可以按元素运算：

```
>> R = rand(2,3)
R =
   0.8147   0.1270   0.6324
   0.9058   0.9134   0.0975

>> (R > 0.2) & (R < 0.8)

ans =
     0   0   1
     0   0   0

>> x = 5

x =
   5

>> x >= [1 2 3; 4 5 6; 7 8 9]

ans =
     1   1   1
     1   1   0
     0   0   0
```

甚至更复杂的组合运算也可以按元素进行：

```
>> A = 1:10;
>> B = 2:11;
>> C = 0.1:0.1:1;
>> D = 5:14;

>> M = B ./ (A .* D .* sin(C));
```

1.2.2　向量/矩阵运算

MATLAB 基于线性代数软件包，而在线性代数中使用向量/矩阵运算能够有效地替代 for 循环，提高运算速度。矩阵乘法是最常见的向量/矩阵运算，该运算是对每个元素进行乘法和加法的组合运算。

先考虑两个列向量 $\boldsymbol{a}$ 和 $\boldsymbol{b}$，那么它们的点积为 1×1 的矩阵，如下所示：

$$\boldsymbol{a}=\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix},\quad \boldsymbol{b}=\begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix}$$

$$\boldsymbol{a}\cdot\boldsymbol{b}=\boldsymbol{a}^{\mathrm{T}}\boldsymbol{b}=\begin{bmatrix} a_x & a_y & a_z \end{bmatrix}\begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix}=\begin{bmatrix} a_xb_x+a_yb_y+a_zb_z \end{bmatrix}$$

如果两个向量 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 均是行向量，那么 $\boldsymbol{a}\cdot\boldsymbol{b}$ 计算定义为 $\boldsymbol{a}\boldsymbol{b}^{\mathrm{T}}$，由乘法和加法的组合运算，得到 1×1 的矩阵，如下：

```
A = 1:10  % 1×10 matrix
B = 0.1:0.1:1.0  % 1×10 matrix
C = 0;
for i = 1:10
    C = C + A(i) * B(i);
end
```

```
A = 1:10  % 1×10 matrix
B = 0.1:0.1:1.0  % 1×10 matrix
C = 0;
C = A*B';  % A·B^T
```

在许多情况下，以向量运算的形式考虑矩阵乘法是很有效的。例如，可以将矩阵—向量乘法 $\boldsymbol{y}=\boldsymbol{A}\boldsymbol{x}$ 分解为 $\boldsymbol{x}$ 和矩阵 $\boldsymbol{A}$ 各行的点乘：

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_i \end{bmatrix} = \begin{bmatrix} \boldsymbol{a}_1 \\ \boldsymbol{a}_2 \\ \vdots \\ \boldsymbol{a}_i \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_i \end{bmatrix}$$

$$y_i = \boldsymbol{a}_i \cdot \boldsymbol{x}$$

1.2.3 实用技巧

在许多应用中，需要对每个元素设定上边界和下边界。为了实现这一目的，通常使用 if 和 elseif 语句，但这容易破坏向量化。因此，可以使用内置函数 min 和 max 替代 if 和 elseif 语句设置元素边界：

```
% ifExample.m
clear all;
tic
A = 0:0.000001:10;
B = 0:0.000001:10;
Z = zeros(size(A));
y = 0;
for i = 1:10000001

  if(A(i) < 0.1) A(i) = 0.1;
  elseif(A(i) > 0.9) A(i) = 0.9;
  end

  Z(i) = sin(0.5*A(i)) * exp(B(i)^2);
  y = y + Z(i);

end
toc
y
```

```
% nonifExample.m
clear all;
tic
A = 0:0.000001:10;
B = 0:0.000001:10;
Z = zeros(size(A));
y = 0;
A = max(A, 0.1);
% max(A, LowerBound)
% A >= LowerBound

A = min(A, 0.9);
% min(A, UpperBound)
% A <= UpperBound
y = sin(0.5*A) * exp(B.^2)';
toc
y
```

```
>> ifExample
Elapsed time is 0.878781 seconds.

y =
   5.8759e+47

>> nonifExample
Elapsed time is 0.309516 seconds.

y =
   5.8759e+47
```

同样地，如果需要查找和替换某些元素的值，可以用 find 函数替代 if 和 elseif 来保持向量化：

```
% ifExample2.m

clear all;
tic

A = 0:0.000001:10;
B = 0:0.000001:10;

Z = zeros(size(A));
y = 0;

for i = 1:10000001

if(A(i) == 0.5) A(i) = 0;
end

  Z(i) = sin(0.5*A(i)) * exp(B(i)^2);
  y = y + Z(i);

end

toc
y
```

```
% nonifExample2.m

clear all;
tic

A = 0:0.000001:10;
B = 0:0.000001:10;

Z = zeros(size(A));
y = 0;

% Vector A is compared with scalar
% 0.5

A(find(A == 0.5)) = 0;


y = sin(0.5*A) * exp(B.^2)';

toc
y
```

将向量 ***A*** 中的元素与标量 0.5 进行比较，返回一个与向量 ***A*** 相同大小的向量，***A*** 中元素为 0.5 的位置设为 0。find 函数能够给出匹配位置的索引，并替换初始值。

1.3　预分配

由于每次调整数组的大小都涉及内存的释放或分配，以及数值的复制，极为耗费时间。所以通过为所需数组预分配内存，能够获得相当显著的加速。

```
% preAlloc.m

% Resizing Array
tic

x = 8;
```

```
x(2)=10;
x(3)=11;
x(4)=20;

toc

% Pre-allocation
tic

y=zeros(4,1);
y(1)=8;
y(2)=10;
y(3)=11;
y(4)=20;

toc
>> preAlloc
Elapsed time is 0.000032 seconds.
Elapsed time is 0.000014 seconds.
```

在上面的例子中，多次调整了数组 x 的大小，需要对内存重新分配和设置数值，而数组 y 则通过 zeros(4, 1) 进行了初始化。如果在运算前不知道矩阵的大小，可以预先设置一个足够大的数组来存储数据，这样使用该数组进行运算时就不需要进行内存的重新分配了。

1.4 for-loop

许多情况下，不可避免地需要使用 for-loop。作为 Fortran 的演进，MATLAB 按列顺序存储矩阵，即第一列的元素按顺序存储在一起，然后是第二列的元素，以此类推。由于内存为线性结构，系统能够缓存附近元素的值，所以对矩阵按列嵌套循环更有利于程序运行。

```
% row_first.m

clear all;

A=rand(300,300,40,40);
B=zeros(300,300,40,40);

tic
for i=1:300
  for j=1:300

    B(i,j,:,:)=2.5*A(i,j,:,:);

  end
end

toc
```

```
% col_first.m

clear all;

A=rand(300,300,40,40);
B=zeros(300,300,40,40);

tic
for j=1:300
  for i=1:300

    B(i,j,:,:)=2.5*A(i,j,:,:);

  end
end

toc
```

```
>> row_first
Elapsed time is 9.972601 seconds.
>> col_first
Elapsed time is 7.140390 seconds.
```

上述例子中，通过嵌套 for-loop 中 i 和 j 的简单对换，col_first.m 的运行速度比 row_first.m 快大约 30%。所以当使用大规模高维矩阵时，进行以列为主的运算能够获得更高的速度增益。

1.5　考虑稀疏矩阵形式

对于稀疏矩阵（大部分元素是零的大规模矩阵）的处理，使用 MATLAB 中的“稀疏矩阵形式”是一种很好的想法。对于非常大的矩阵，因为只需存储非零元素，所以稀疏矩阵所需内存更少，而且运算过程中不考虑零元素，运算速度也更快。

创建稀疏形式矩阵的简易方法如下：

```
>> i = [5 3 6] % used for row index
>> j = [1 6 3] % used for column index
>> value = [0.1  2.3  3.1] % used for values to fill in
>> s = sparse(i,j,value) % generate sparse matrix
s =
   (5,1)   0.1000
   (6,3)   3.1000
   (3,6)   2.3000
>> full = full(s) % convert the sparse matrix to full matrix
full =
         0         0         0         0         0         0
         0         0         0         0         0         0
         0         0         0         0         0    2.3000
         0         0         0         0         0         0
    0.1000         0         0         0         0         0
         0         0    3.1000         0         0         0
```

或者简单地使用内置指令 sparse 将全矩阵转化为稀疏矩阵：

```
>> SP = sparse(full)
SP =
   (5,1)   0.1000
   (6,3)   3.1000
   (3,6)   2.3000
```

所有 MATLAB 内置运算都适用于稀疏矩阵形式，无需进行任何修改，而且稀疏矩阵的运算结果也是稀疏矩阵形式。

稀疏矩阵形式的运行效率由矩阵的稀疏度决定，即矩阵中零元素越多，矩阵运算速度越快。如果一个矩阵几乎不稀疏，那么使用稀疏矩阵形式，将不会带来速度的提升和内存的节省。

```
% DenseSparse.m
% Dense matrix
A = rand (5000,2000);

b = rand (5000,1);
tic
   x = A\b ;
toc
% Convert to sparse form in MATLAB
sp_denseA = sparse(A);
tic
   x = sp_denseA\b ;
toc
```

```
% RealSparse.m
% Sparse matrix
sp_A2 = sprand(5000, 2000, 0.2);
b2 = rand (5000,1);

% Convert sparse matrix to full
% matrix
full_A2 = full(sp_A2);
tic
   y = full_A2\b ;
toc
tic
   y = sp_A2\b2 ;
toc
```

```
>> denseSparse
Elapsed time is 5.846979 seconds.
Elapsed time is 41.050271 seconds.

>> RealSparse
Elapsed time is 5.879175 seconds.
Elapsed time is 3.798073 seconds.
```

在例子 DenseSparse.m 中，尝试将密集矩阵（A:5000×5000）转换为稀疏矩阵（sp_denseA），稀疏矩阵形式的运算比密集矩阵花费了更多时间。然而在例子 RealSparse.m 中，稀疏矩阵形式的速度提高了很多。函数 sprand(5000, 2000, 0.2) 用于生成 5000×2000 的稀疏矩阵。第三个参数 0.2 意味着，在生成的矩阵中大约 20%的元素为非零元素，其他 80%的元素为零元素。在矩阵运算中，即使有 20%的非零元素，稀疏矩阵也能够获得非常大的速度提升。

观察每个矩阵所需的内存大小（如下所示），稀疏形式的稀疏矩阵（sp_A2）比全矩阵（80MB）需要更少的内存（大约 29MB）。然而稀疏形式的密集矩阵（sp sp_denseA）比全矩阵（80MB）需要更多的内存（大约 160MB），这是因为稀疏矩阵形式需要更多的内存来存储矩阵的索引信息。