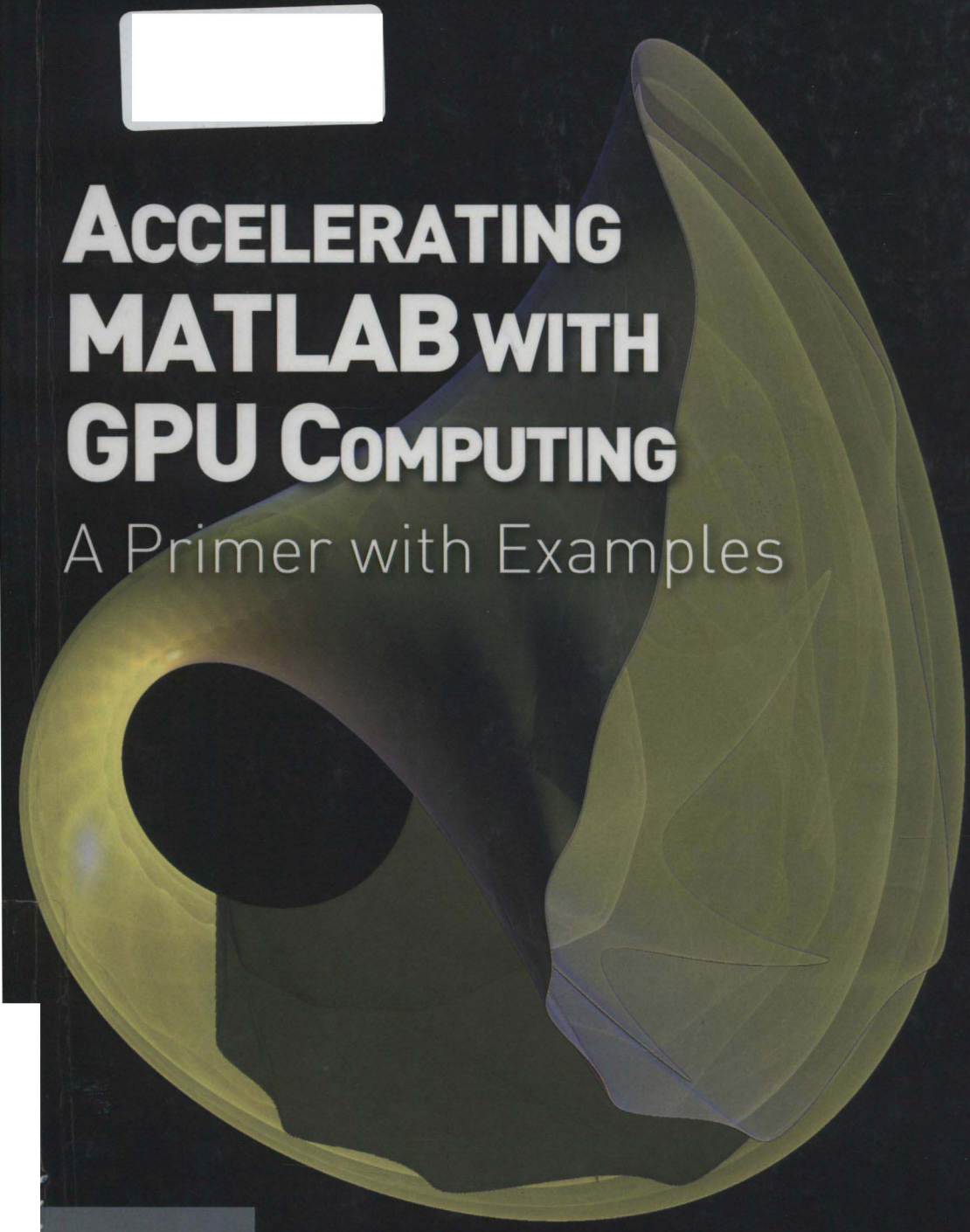
# ACCELERATING MATLAB WITH GPU COMPUTING

## A Primer with Examples



Jung W. Suh
Youngmin Kim

MK
MORGAN KAUFMANN

# Accelerating MATLAB with GPU Computing

## A Primer with Examples

*Jung W. Suh*
*Youngmin Kim*



ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Morgan Kaufmann is an imprint of Elsevier

MK
MORGAN KAUFMANN

**Acquiring Editor**: Todd Green
**Editorial Project Manager**: Lindsay Lawrence
**Project Manager**: Mohana Natarajan
**Designer**: Matthew Limbert

Morgan Kaufmann is an imprint of Elsevier

225 Wyman Street, Waltham, MA 02451, USA

First edition 2014



**British Library Cataloguing-in-Publication Data**
A catalogue record for this book is available from the British Library

**Library of Congress Cataloging-in-Publication Data**
Application Submitted

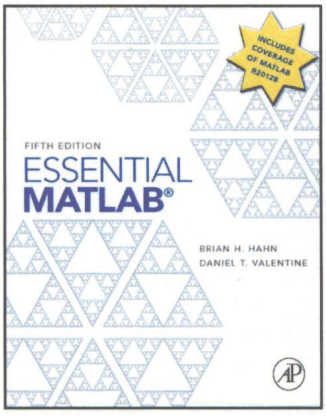
ISBN: 978-0-12-408080-5

For information on all MK publications visit our web site at *www.mkp.com*

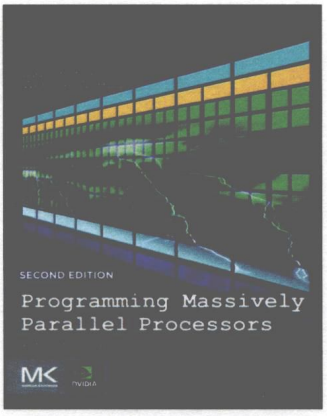
Printed and bound in USA

14 15 16 17 18 10 9 8 7 6 5 4 3 2 1

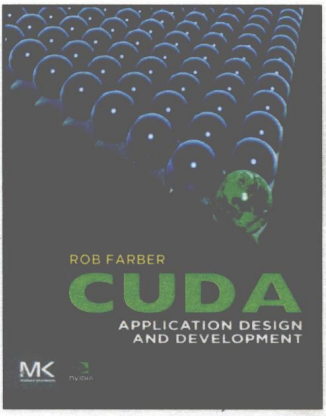
# Related Titles from Morgan Kaufmann



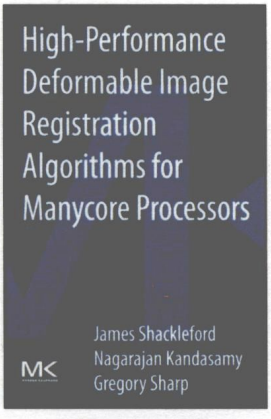

***Essential Matlab for Engineers and Scientists***
Brian Hahn and David Valentine
ISBN: 9780123943989



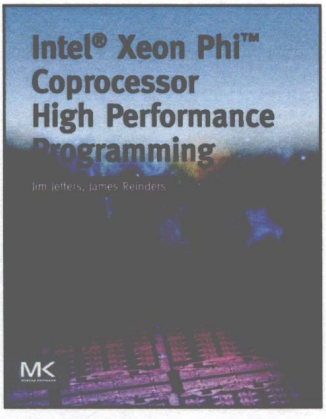

***Programming Massively Parallel Processors, 2nd Edition***
*A Hands-on Approach*
David Kirk and Wen-mei Hwu
ISBN: 9780124159921



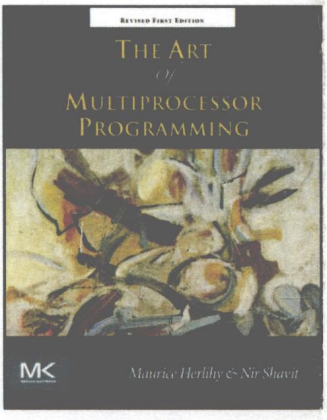

***CUDA Application Design and Development***
Rob Farber
ISBN: 9780123884268




***High Performance Deformable Image Registration Algorithms for Manycore Processors***
James Shackleford,
Nagarajan Kandasamy, Gregory Sharp
ISBN: 9780124077416




***Intel Xeon Phi Coprocessor High Performance Programming***
Jim Jeffers and James Reinders
ISBN: 9780124104143




***The Art of Multiprocessor Programming, Revised Reprint***
Maurice Herlihy and Nir Shavit
ISBN: 9780123973375

MORGAN KAUFMANN

mkp.com

# Accelerating MATLAB with GPU Computing

# Preface

MATLAB is a widely used simulation tool for rapid prototyping and algorithm development. Many laboratories and research institutions face growing demands to run their MATLAB codes faster for computationally heavy projects after simple simulations. Since MATLAB uses a vector/matrix representation of data, which is suitable for parallel processing, it can benefit a lot from GPU acceleration.

## Target Readers and Contents

This book is aimed primarily at the graduate students and researchers in the field of engineering, science, and technology who need huge data processing without losing the many benefits of MATLAB. However, MATLAB users come from various backgrounds and do not necessarily have much programming experience. For those whose backgrounds are not from programming, GPU acceleration for MATLAB may distract their algorithm development and introduce unnecessary hassles, even when setting the environment. This book targets the readers who have some or a lot of experience on MATLAB coding but not enough depth in either C coding or the computer architecture for parallelization. So readers can focus more on their research and work by avoiding non-algorithmic hassles in using GPU and CUDA in MATLAB.

As a primer, the book will start with the basics, walking through the process of setting MATLAB for CUDA (in Windows and Mac OSX), creating `c-mex` and `m`-file profiling, then guide the users through the expert-level topics such as third-party CUDA libraries. It also provides many practical ways to modify users' MATLAB codes to better utilize the immense computational power of graphics processors.

This book guides the reader to dramatically maximize the MATLAB speed using NVIDIA's Graphics Processing Unit (GPU). NVIDIA's Compute Unified Device Architecture (CUDA) is a parallel computing architecture originally designed for computer games but is getting a reputation in the general science and technology fields for its efficient massive computation power. From this book, the reader can take advantage of the parallel processing power of GPU and abundant CUDA scientific libraries for accelerating MATLAB code with no or less effort and time, and bring readers' researches and works to a higher level.

## Directions of this Book

### *GPU Utilization Using c-mex Versus Parallel Computing Toolbox*

This book deals with Mathworks's Parallel Computing Toolbox in Chapter 5. Although Mathworks's Parallel Computing Toolbox is a useful tool for speeding

up MATLAB, the current version still has its limitation in making the Parallel Computing Toolbox a general speeding-up solution, in addition to the extra cost of purchasing the toolbox. Especially, since the Parallel Computing Toolbox targets distributed computing over multicore, multiple computers and/or cluster machines as well as GPU processing, GPU optimization for speeding up the user's code is comparatively limited both in speeding-up and supporting MATLAB functions. Furthermore, if we limit to Mathworks's the Parallel Computing Toolbox only, then it is difficult to find an efficient way to utilize the abundant CUDA libraries to their maximum. In this book, we address both the strengths and the limitations of the current Parallel Computing Toolbox in Chapter 5. For the purpose of general speeding up, GPU-utilization through `c-mex` proves a better approach and provides more flexibility in current situation.

### *Tutorial Approach Versus Case Study Approach*

As the book's title says, we take more of a tutorial approach. MATLAB users may come from many different backgrounds, and web resources are scattered over Mathworks, NVIDIA, and private blogs as fragmented information. The tutorial approach from setting the GPU environment to acquiring critical (but compressed) hardware knowledge for GPU would be beneficial to prospective readers over a wide spectrum. However, this book also has two chapters (Chapters 7 and 8) that include case examples with working codes.

### *CUDA Versus OpenCL*

When we prepared the proposal of this book, we also considered OpenCL as a topic, because the inclusion of OpenCL would attract a wider range of readers. However, while CUDA is more consistent and stable, because it is solely driven by NVIDIA, the current OpenCL has no unified development environment and is still unstable in some areas, because OpenCL is not governed by one company or institution. For this reason, installing, profiling, and debugging OpenCL are not yet standardized. As a primer, this may distract the focus of this book. More importantly, for some reason Mathworks is very conservative in its support of OpenCL, unlike CUDA. Therefore, we decided not to include OpenCL in this edition of our book. However, we will again consider whether to include OpenCL in future editions if increased needs come from market or Mathworks' direction changes.

After reading this book, the reader, in no time, will experience an amazing performance boost in utilizing reader's MATLAB codes and be better equipped in research to enjoy the useful open-source resources for CUDA. The features this book covers are available on Windows and Mac.

# Contents

# 1 Accelerating MATLAB without GPU

## 1.1 Chapter Objectives

In this chapter, we deal with the basic accelerating methods for MATLAB codes in an intrinsic way – a simple code optimization without using GPU or C-MEX. You will learn about the following:

- The vectorization for parallel processing.
- The preallocation for efficient memory management.
- Other useful tips to increase your MATLAB codes.
- Examples that show the code improvements step by step.

## 1.2 Vectorization

Since MATLAB has the vector/matrix representation of its data, "vectorization" can help to make your MATLAB codes run faster. The key for vectorization is to minimize the usage of a `for-loop`.

Consider the following two `m` files, which are functionally the same:

```
% 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
```

The left `nonVec1.m` has a `for-loop` to calculate the sum, while the right `Vec1.m` has no `for-loop` in the code.

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

y =
  -1.3042e+48

>> Vec1
Elapsed time is 0.330786 seconds.

y =
  -1.3042e+48
```

The results are same but the elapsed time for `Vec1.m` is almost three times less than that for `nonVec1.m`. For better vectorization, utilize the elementwise operation and vector/matrix operation.

### 1.2.1 Elementwise Operation

The * symbol is defined as matrix multiplication when it is used on two matrices. But the `.*` symbol specifies an elementwise multiplication. For example, if `x = [1 2 3]` and `v = [4 5 6]`,

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

Many other operations can be performed elementwise:

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

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

Many functions also support this elementwise operation:

```
>> 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
```

Even the relational operators can be used elementwise:

```
>> 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
```

We can do even more complicated elementwise operations together:

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

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

### 1.2.2 Vector/Matrix Operation

Since MATLAB is based on a linear algebra software package, employing vector/matrix operation in linear algebra can effectively replace the `for-loop`, and result in speeding up. Most common vector/matrix operations are matrix multiplication for combining multiplication and addition for each element.

If we consider two column vectors, **a** and **b**, the resulting dot product is the $1 \times 1$ matrix, as follows:

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

$$\mathbf{a} \cdot \mathbf{b} = \mathbf{a}^T\mathbf{b} = \begin{bmatrix} a_x & a_y & a_z \end{bmatrix} \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = \begin{bmatrix} a_x b_x + a_y b_y + a_z b_z \end{bmatrix}$$

If two vectors, **a** and **b**, are row vectors, the $\mathbf{a} \cdot \mathbf{b}$ should be $\mathbf{a}\mathbf{b}^T$ to get the $1 \times 1$ matrix, resulting from the combination of multiplication and addition, as follows.

```
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
```

In many cases, it is useful to consider matrix multiplication in terms of vector operations. For example, we can interpret the matrix-vector multiplication $\mathbf{y} = \mathbf{A}\mathbf{x}$ as the dot products of $x$ with the rows of $A$:

$$\begin{bmatrix} \vdots \\ \mathbf{y} \\ \vdots \end{bmatrix} = \begin{bmatrix} \cdots \boldsymbol{a}_1 \cdots \\ \cdots \boldsymbol{a}_2 \cdots \\ \cdots \boldsymbol{a}_3 \cdots \end{bmatrix} \begin{bmatrix} \vdots \\ \boldsymbol{x} \\ \vdots \end{bmatrix}$$

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

### 1.2.3 Useful Tricks

In many applications, we need to set upper and lower bounds on each element. For that purpose, we often use `if` and `elseif` statements, which easily break vectorization. Instead of `if` and `elseif` statements for bounding elements, we may use `min` and `max` built-in functions:

```
% ifExample.m
clear all;
tic
```

```
% nonifExample.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
```

```
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
```

Similarly, if you need to find and replace some values in elements, you can also avoid `if` and `elseif` statements by using the `find` function to keep vectorization.

```
% 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
```

```
% 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;
```