

Julia High Performance

Second Edition

Optimizations, distributed computing, multithreading, and GPU programming with Julia 1.0 and beyond



Packt

www.packt.com

Avik Sengupta

Copyrighted material

Julia High Performance

Second Edition

Copyright © 2019 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Commissioning Editor: Aaron Lazar

Acquisition Editor: Denim Pinto

Content Development Editor: Akshita Billava

Technical Editor: Neha Pande

Copy Editor: Safis Editing

Project Coordinator: Vaidehi Sawant

Proofreader: Safis Editing

Indexer: Tejal Daruwale Soni

Production Designer: Jisha Chirayil

First published: April 2016

Second edition: June 2019

Production reference: 1070619

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78829-811-7

www.packtpub.com

Table of Contents

Preface	1
Chapter 1: Julia is Fast	7
Julia – fast and dynamic	8
Designed for speed	10
JIT and LLVM	10
Types, type inference, and code specialization	11
How fast can Julia be?	13
Summary	17
Chapter 2: Analyzing Performance	18
Timing Julia functions	18
The @time macro	19
Other time macros	20
The Julia profiler	21
Using the profiler	22
ProfileView	24
Using Juno for profiling	26
Using TimerOutputs	27
Analyzing memory allocation	28
Using the memory allocation tracker	29
Statistically accurate benchmarking	29
Using BenchmarkTools.jl	29
Summary	31
Chapter 3: Types, Type Inference, and Stability	32
The Julia type system	32
Using types	33
Multiple dispatch	34
Abstract types	35
Julia's type hierarchy	36
Composite and immutable types	37
Type parameters	38
Type inference	39
Type-stability	40
Definitions	40
Fixing type instability	41
The performance pitfalls	43
Identifying type stability	44
Loop variables	47

Kernel methods and function barriers	50
Types in storage locations	51
Arrays	51
Composite types	53
Parametric composite types	54
Summary	55
Chapter 4: Making Fast Function Calls	56
Using globals	56
The trouble with globals	57
Fixing performance issues with globals	58
Inlining	60
Default inlining	60
Controlling inlining	62
Disabling inlining	64
Constant propagation	65
Using macros for performance	66
The Julia compilation process	66
Using macros	67
Evaluating a polynomial	68
Horner's method	69
The Horner macro	70
Generated functions	71
Using generated functions	71
Using generated functions for performance	72
Using keyword arguments	74
Summary	75
Chapter 5: Fast Numbers	76
Numbers in Julia, their layout, and storage	76
Integers	76
Integer overflow	78
BigInt	81
The floating point	82
Floating point accuracy	83
Unsigned integers	84
Trading performance for accuracy	85
The @fastmath macro	86
The K-B-N summation	89
Subnormal numbers	90
Subnormal numbers to zero	90
Summary	92
Chapter 6: Using Arrays	93
Array internals in Julia	93
Array representation and storage	94

Column-wise storage	95
Adjoints	98
Array initialization	98
Bounds checking	100
Removing the cost of bounds checking	100
Configuring bound checks at startup	101
Allocations and in-place operations	101
Preallocating function output	102
sizehint!	103
Mutating functions	103
Broadcasting	104
Array views	106
SIMD parallelization (AVX2, AVX512)	109
SIMD.jl	112
Specialized array types	114
Static arrays	114
Structs of arrays	117
Yeppp!	120
Writing generic library functions with arrays	121
Summary	124
Chapter 7: Accelerating Code with the GPU	125
Technical requirements	126
Getting started with GPUs	126
CUDA and Julia	128
CuArrays	130
Monte Carlo simulation on the GPU	131
Writing your own kernels	133
Measuring GPU performance	133
Performance tips	136
Scalar iteration	136
Combining kernels	138
Processing more data	138
Deep learning on the GPU	139
ArrayFire	141
Summary	143
Chapter 8: Concurrent Programming with Tasks	144
Tasks	144
Using tasks	145
The task life cycle	147
task_local_storage	149
Communicating between tasks	149
Task iteration	151
High-performance I/O	153

Port sharing for high-performance web serving	153
Summary	154
Chapter 9: Threads	155
Threads	156
Measuring CPU cores	156
Hwloc	158
Starting threads	159
The @threads macro	160
Prefix sum	161
Thread safety and synchronization primitives	162
Multithreaded Monte Carlo simulation	162
Atomics	164
Synchronization primitives	166
Threads and GC	168
Threaded libraries	168
Over-subscription	169
The future of threading	170
Summary	171
Chapter 10: Distributed Computing with Julia	172
Creating Julia clusters	172
Starting a cluster	173
Cluster managers	174
SSHManager	174
SLURM	175
Communication between Julia processes	176
Programming parallel tasks	177
The @everywhere macro	177
The @spawn macro	178
The @spawnat macro	179
Parallel for loops	180
Parallel map	181
Distributed Monte Carlo	182
Distributed arrays	183
Conway's Game of Life	185
Shared arrays	188
Parallel prefix sum with shared arrays	189
Summary	189
Licences	190
Other Books You May Enjoy	192
Index	195

Preface

The Julia programming language has brought an innovative new approach to scientific computing, promising a combination of performance and productivity that is not usually available in the current set of languages that is commonly used. In solving the two-language problem, it has seen tremendous growth both in academia and industry. It has been used in domains from robotics, astronomy, and physics, to insurance and trading. It has particular relevance in the area of machine learning, with increasing use for the emerging field of differentiable computing.

Most new developers are attracted to the language due to its promise of high performance. This book shows you how and why that is possible. We talk about the design choices of the language's creators that allow such a high-performance compiler to be built. We also show you the steps that you, as an application developer, can take to ensure the highest possible performance for your code. We also tell you the ways in which your code can work with the compiler and runtime to fully utilize your hardware to the greatest extent possible.

Who this book is for

This book is for the beginner and intermediate Julia developer who wants to fully leverage Julia's promise of performance with productivity. We assume you are proficient with one or more programming languages and have some familiarity with Julia's syntax. We do not expect you to be expert Julia programmers yet but assume that you have written small Julia programs, or that you have taken an introductory course on the language.

What this book covers

Chapter 1, *Julia is Fast*, is your introduction to Julia's unique performance. Julia is a high-performance language, with the possibility to run code that is competitive in performance with code written in C. This chapter explains why Julia code is fast. It also provides context and sets the stage for the rest of the book.

Chapter 2, *Analyzing Performance*, shows you how to measure the speed of Julia programs and understand where the bottlenecks are. It also shows you how to measure the memory usage of Julia programs and the amount of time spent on garbage collection.

Chapter 3, *Types, Type Inference, and Stability*, covers type information. One of the principal ways in which Julia achieves its performance goals is by using type information. This chapter describes how the Julia compiler uses type information to create fast machine code. It describes ways of writing Julia code to provide effective type information to the Julia compiler.

Chapter 4, *Making Fast Function Calls*, explores functions. Functions are the primary artifacts for code organization in Julia, with multiple dispatch being the single most important design feature in the language. This chapter shows you how to use these facilities for fast code.

Chapter 5, *Fast Numbers*, describes some internals of Julia's number types in relation to performance, and helps you understand the design decisions that were made to achieve that performance.

Chapter 6, *Using Arrays*, focuses on arrays. Arrays are one of the most important data structures in scientific programming. This chapter shows you how to get the best performance out of your arrays—how to store them, and how to operate on them.

Chapter 7, *Accelerating Code with the GPU*, covers the GPU. In recent years, the general-purpose GPU has turned out to be one of the best ways of running fast parallel computations. Julia provides a unique method for compiling high-level code to the GPU. This chapter shows you how to use the GPU with Julia.

Chapter 8, *Concurrent Programming with Tasks*, looks at concurrent programming. Most programs in Julia run on a single thread, on a single processor core. However, certain concurrent primitives make it possible to run parallel, or seemingly parallel, operations, without the full complexities of shared memory multi-threading. In this chapter, we discuss how the concepts of tasks and asynchronous IO help create responsive programs.

Chapter 9, *Threads*, moves on to look at how Julia now has new experimental support for shared memory multi-threading. In this chapter, we discuss the implementation details of this mode, and see how this is different from other languages. We see how to speed up our computations using threads, and learn some of the limitations that currently exist in this model.

Chapter 10, *Distributed Computing with Julia*, recognizes that there comes a time in every large computation's life when living on a single machine is not enough. There is either too much data to fit in the memory of a single machine, or computations need to be finished quicker than can be achieved on all the cores of a single processor. At that stage, computation moves from a single machine to many. Julia comes with advanced distributed computation facilities built in, which we describe in this chapter.

To get the most out of this book

This book has been written to be a practical guide to improving the performance of your Julia code. As such, we encourage you to run the code shown in this book yourself. Running the code and inspecting the output for yourself is the best way to learn the methods suggested here. All the code is available in machine-readable format (see the following for download instructions), so we suggest having a Julia REPL open while you read this book, so that you can copy and paste code on to it.

Download the example code files

You can view and download all code for this book at <https://juliahighperformance.com>.

You can download the example code files for this book from your account at www.packt.com. If you purchased this book elsewhere, you can visit www.packt.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packt.com.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Julia-High-Performance-Second-Edition>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: https://www.packtpub.com/sites/default/files/downloads/9781788298117_ColorImages.pdf.

Code in Action

Click on the following link to see the Code in Action: <http://bit.ly/2WsM0md>

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, function or method names, folder names, and filenames. Here is an example: "Mount the downloaded WebStorm-10*.dmg disk image file as another disk in your system."

A block of code is set as follows:

```
struct Pixel{T}
    x::Int64
    y::Int64
    color::T
end
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
function sum_cols_matrix(x)
    num_cols = size(x, 2)
    s = zeros(num_cols)
    for i = 1:num_cols
        s[i] = sum_vector(x[:, i])
    end
    return s
end
```

Most code snippets in this book have been typed at the Julia REPL. This is denoted by the **julia>** prompt. Such a listing will show the output of the expression below the expression itself. If you type the code into the REPL yourself, this is exactly what you should see:

```
julia> a = fill(1, 4, 4)
4x4 Array{Int64,2}:
 1 1 1 1
 1 1 1 1
 1 1 1 1
 1 1 1 1
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select **System info** from the **Administration** panel."

Warnings or important notes appear like this.



Tips and tricks appear like this.



Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customercare@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packt.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

1 Julia is Fast

In many ways, the history of programming languages has been driven by, and certainly intertwined with, the needs of numerical and scientific computing. The first high-level programming language, Fortran, was created to solve scientific computing problems, and continues to be important in the field even to this day. In recent years, the rise of data science as a specialty has brought additional focus to numerical computing, particularly for statistical uses. In this area, somewhat counter-intuitively, both specialized languages such as R and general-purpose languages such as Python are in widespread use. The rise of Hadoop and Spark has spread the use of Java and Scala respectively among this community. In the midst of all this, Matlab has had a strong niche within engineering communities, while Mathematica remains unparalleled for symbolic operations.

A new language for scientific computing therefore has a very high barrier to overcome, and it's been only a few short years since the Julia language was introduced to the world. In that time, however, its innovative features, combining the ease of use of a dynamic language and the performance of a statically compiled language, have created a growing niche within the numerical computing world. Based on multiple dispatch as its defining paradigm, Julia is a very pleasant language to program in, making mathematical abstractions very easy to express. However, it was the claim of high performance that drew the earliest adopters.

This, then, is a book that celebrates writing high-performance programs. With Julia, this is not only possible, but also reasonably straightforward, in a low-overhead, dynamic language.

As a reader of this book, you have likely already written your first few Julia programs. We will assume that you have successfully installed Julia, and have a working programming environment available. We expect you are familiar with very basic Julia syntax, but we will discuss and review many of those concepts throughout the book as we introduce them.

In this chapter, we will describe some of the underlying design elements of Julia that contribute to its well-deserved reputation as a fast language:

- Julia – fast and dynamic
- Designed for speed
- How fast can Julia be?

Julia – fast and dynamic

It is a widely believed myth in programming language communities that high-performance languages and dynamic languages are completely disjointed sets. The perceived wisdom is that, if you want programmer productivity, you should use a dynamic language, such as Ruby, Python, or R. On the other hand, if you want fast code execution, you should use a statically typed language, such as C or Java.

There are always exceptions to this rule. However, for most mainstream programmers, this is a strongly held belief. This usually manifests itself in what is known as the two-language problem. This is something that is especially prominent in scientific computing. This is the situation where the performance-critical inner kernel is written in C, but is then wrapped and used from a dynamic, higher-level language. Code written in traditional, scientific computing environments such as R, Matlab, or NumPy follows this paradigm.

Code written in this fashion is not without its drawbacks, however. Even though it looks like it gets you the best of both worlds—fast computation, while allowing the programmer to use a high-level language—this is a path full of hidden dangers. For one, someone will have to write the low-level kernel. So, you need two different skill sets. If you are lucky enough to find the low-level code in C for your project, you are fine. However, if you are doing anything new or original, or even slightly different from the norm, you will find yourself writing both C and a high-level language. This will severely limit the number of contributors that your projects or research will get: to be really productive, those contributors really have to be familiar with two languages.

Secondly, when running code routinely written in two languages, there can be severe and unforeseen performance pitfalls. When you can drop down to C code quickly, everything is fine. However, if, for time reasons, effort, skill or changing requirements, you cannot write a performance-intensive part of your algorithm in C, you'll find your program taking hundreds or even thousands of times longer than you expected.

Julia is the first modern language to make a reasonable effort to solve the two-language problem. It is a high-level, dynamic language with powerful features that make for very productive programming. At the same time, code written in Julia usually runs very quickly, almost as quickly as code written in statically typed languages.

The rest of this chapter describes some of the underlying design decisions that make Julia such a fast language. We'll also look at some evidence of the performance claims about Julia. The rest of the book shows you how to write your Julia programs to be as fast and lean as possible. We will discuss how to measure and reason about performance in Julia, and how to avoid some potential performance roadblocks.

For all the content in this book, we will usually illustrate our points with small, self-contained programs. We hope that this will enable you grasp the crux of the issue, without getting distracted by unnecessary elements of a larger program. We expect that this methodology will therefore provide you with instinctive intuition about Julia's performance profile.

Julia has a refreshingly simple performance model—thus, writing fast Julia code is a matter of understanding a few key elements of computer architecture, and how the Julia compiler interacts with it. We hope that, by the end of this book, your instincts are developed well enough to design and write your own Julia code with the fastest possible performance.

Finally, Julia will work for you at both ends of the compute spectrum. On one hand, its performance and expressiveness allows it to run embedded use cases on low-powered processors and it is fully supported on ARM processors, and works well on the Raspberry Pi, which makes it a perfect environment for teaching programming. At the other end of the spectrum, Julia has been used to run large-scale machine learning applications on some of the world's largest super-computers. The Celeste project used Julia Build and Atlas of the Sky, where the computation ran at an amazing 1.5 petaflops (1 petaflop is 10^{15} floating point operations per second, or a thousand million million), using 1.3 million threads. This was the first time any dynamic language had broken the petaflop barrier. So, Julia can run on machines large and small, scaling massively in both directions.

Versions of Julia:



The code and examples in this book are targeted at version 1.2 of the language, which is the most recently released version at the time of publication. Since there will be no breaking changes in the 1.x series of Julia, most of the code in this book should work on version 1.0 onward, which was released in August of 2018.

Designed for speed

When the creators of Julia launched the language into the world, they said the following in a blog post entitled *Why We Created Julia*, which was published in early 2012:

"We want a language that's open source, with a liberal license. We want the speed of C with the dynamism of Ruby. We want a language that's homoiconic, with true macros like Lisp, but with obvious, familiar mathematical notation like Matlab. We want something as usable for general programming as Python, as easy for statistics as R, as natural for string processing as Perl, as powerful for linear algebra as Matlab, as good at gluing programs together as the shell. Something that is dirt simple to learn, yet keeps the most serious hackers happy. We want it interactive and we want it compiled. (Did we mention it should be as fast as C?)"

High-performance, indeed nearly C-level performance, has therefore been one of the founding principles of the language. It's built from the ground up to enable the fast execution of code.

In addition to being a core design principle, it has also been a necessity from the early stages of its development. A very large part of Julia's standard library, including very basic low-level operations, is written in Julia itself. For example, the `+` operation to add two integers is defined in Julia itself. (Refer to: <https://github.com/JuliaLang/julia/blob/e1def102429941705bc16009e35a74abcd6f88e/base/int.jl#L38>.) Similarly, the basic `for` loop uses the standard iteration mechanism available to all user-defined types.

Broadcasting, which is a fundamental low-level operation in the compiler, can be completely overridden by custom array types (this is used heavily in CUDA arrays, for example). All of this means that the compiler had to be very fast from the very beginning to create a usable language. The creators of Julia did not have the luxury of escaping to C for even the core elements of the library.

We will note throughout the book the many design decisions that have been made with an eye to high performance, but there are three main elements that create the basis for Julia's speed: a high performance Just in Time compiler, LLVM to generate machine code, and a type system that allows expressive code.

JIT and LLVM

Julia is a **Just In Time (JIT)** compiled language, rather than an interpreted one. This allows Julia to be dynamic, without having the overhead of interpretation. This compilation infrastructure is built on top of **LLVM**—more information about it is available on its website: <http://llvm.org>.

The LLVM compiler infrastructure project originated at the University of Illinois. It now has contributions from a very large number of corporate as well as independent developers. As a result of all this work, it is now a very high-quality, yet modular, system for many different compilation and code generation activities.

Julia uses LLVM for its JIT compilation needs. The Julia runtime code generator produces LLVM **Intermediate Representation (IR)** and hands it over to LLVM's JIT compiler, which in turn generates machine code that is executed on the CPU. As a result, sophisticated compilation techniques that are built into LLVM are ready and available to Julia, from simple ones (such as *Loop Unrolling* or *Loop Deletion*) to state-of-the-art ones (such as *SIMD Vectorization*). These compiler optimizations form a very large body of work and, in this sense, the existence of LLVM is very much a pre-requisite to the existence of Julia. It would have been an almost impossible task for a small team of developers to build this compiler and code generation infrastructure from scratch.

Just-In-Time compilation:

A technique in which the code in a high-level language is converted to machine code for execution on the CPU at runtime. This is in contrast to interpreted languages, whose runtime executes the source language directly.



This usually has a significantly higher overhead. On the other hand, **Ahead of Time (AOT)** compilation refers to the technique of converting a source language into machine code as a separate step prior to running the code. In this case, the converted machine code can usually be saved to disk as an executable file.

Types, type inference, and code specialization

While LLVM provides the basic infrastructure that allows fast machine code to be produced, it must be noted that adding an LLVM compiler to any language will not necessarily make it execute faster. Julia's syntax and semantics have been carefully designed to allow high-performance execution, and a large part of this is due to how Julia uses types in the language. We will, of course, have much more to say about types in Julia throughout this book. At this stage, suffice it to say that Julia's concept of types is a key ingredient of its performance.

The Julia compiler attempts to infer the type of all data used in a program, and compiles different versions of functions specialized to particular types of its arguments. To take a simple example, consider the `^` (power) function. This function can be called with integer or floating point (i.e., fractional, or decimal) arguments. The mathematical definitions and, thus, the implementation of this function are very different for integers and floats. So, Julia will compile, on demand, two versions of the code, one for integer arguments, and one for floating point arguments, and insert the appropriate call in the code when it compiles the program. This means that, at runtime, fast, straight-line code without any type checks will be executed on the CPU.

Julia allows us to introspect the native code that runs on the CPU. Using this facility, we can see that very different code is generated for integer and floating point arguments. So, let's look at the following machine code, generated for squaring an integer:

```
julia> @code_native 3^2
pushl %eax
decl %eax
movl $202927424, %eax ## imm = 0xC186D40
addl %eax, (%eax)
addb %al, (%eax)
calll *%eax
popl %ecx
retl
```



We omitted some boilerplate output when showing the result of the `@code` macro, in order to focus on the relevant parts. Run this code yourself to see the full output.

Let's now look at the following code, generated for squaring a floating point value:

```
julia> @code_native 3.5^2
vcvttsi2sdl %edi, %xmm1, %xmm1
decl %eax
movl $1993314664, %eax ## imm = 0x76CF9168
.byte 0xff .byte 0x7f .byte 0x00
addb %bh, %bh
loopne 0x68
nopw %cs:(%eax, %eax)
```

You will notice that the code looks very different (although the actual meaning of the code is not relevant for now). You will notice that there are no runtime type checks in the code. This gets to the heart of Julia's design and its performance claims.

The ability of the compiler to reason about types is due to the combination of a sophisticated dataflow-based algorithm, and careful language design that allows this information to be inferred from most programs before execution begins. Put in another way, the language is designed to make it easy to statically analyze its data types.

If there is a single reason for Julia being such a high-performance language, this is it. This is why Julia is able to run at C-like speeds while still being a dynamic language. *Type inference* and *code specialization* are as close to a secret sauce as Julia gets. It is notable that, outside this type inference mechanism, the Julia compiler is quite simple. It does not include many of the advanced *Just in Time* optimizations that Java and JavaScript compilers are known to use. When the compiler has enough information about the types within the code, it can generate optimized, straight-line code without many of these advanced techniques.



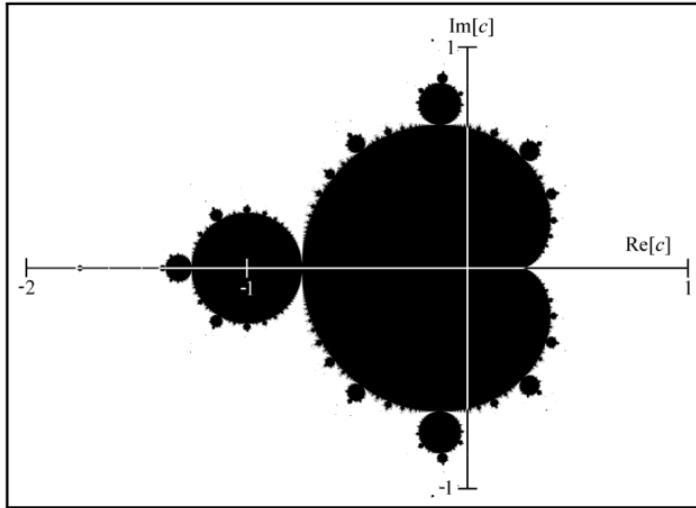
Detailed information about the implementation of type inference and code specialization in Julia can be found in the paper *Julia: A Fresh Approach to Numerical Computing*. Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah (2017) *SIAM Review*, 59: 65–98. doi: 10.1137/141000671.
URL: <https://julialang.org/research/julia-fresh-approach-BEKS.pdf>

It is useful to note here that, unlike some optionally typed dynamic languages, simply adding type annotations to your code does not make Julia go any faster. Type inference means that the compiler is usually able to figure out the types of variables when necessary. Hence, you can usually write high-level code without fighting with the compiler about types, and still achieve superior performance.

How fast can Julia be?

The best evidence of Julia's performance claims is when you write your own code. We encourage you to run and measure all the code snippets in the book. To start, we will provide an indication of how fast Julia can be by comparing a similar algorithm on multiple languages.

As an example, consider the algorithm to compute a Mandelbrot set. Given a complex number, z , the function computes whether, after a certain number of iterations, the $f_c(z) = z^2 + c$ function converges or not. Plotting the imaginary numbers where that function diverges on a 2D plane produces the following iconic fractal image that is associated with this set:



The following code computes the divergence point based on this logic. Calling this function over all points on a 2D plane will produce the Mandelbrot set:

```
function mandel(c)
    z = c
    maxiter = 80
    for n in 1:maxiter
        if abs(z) > 2
            return n - 1
        end
        z = z^2 + c
    end
    return maxiter
end
```

You will notice that this code contains no type annotations, or any special markup, even though it operates on complex numbers. It looks remarkably clean, and the idea that the same mathematical operations can apply to many different kinds of mathematical objects is key to Julia's expressiveness.

The same algorithm implemented in modern C would look as follows:

```
int mandel(double complex z) {
    int maxiter = 80;
    double complex c = z;
    for (int n = 0; n < maxiter; ++n) {
        if (cabs(z) > 2.0) {
            return n;
        }
        z = z*z+c;
    }
    return maxiter;
}
```

Downloading the example code:

You can download the example code files for this book from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files emailed directly to you.

You can download the code files by taking the following steps:

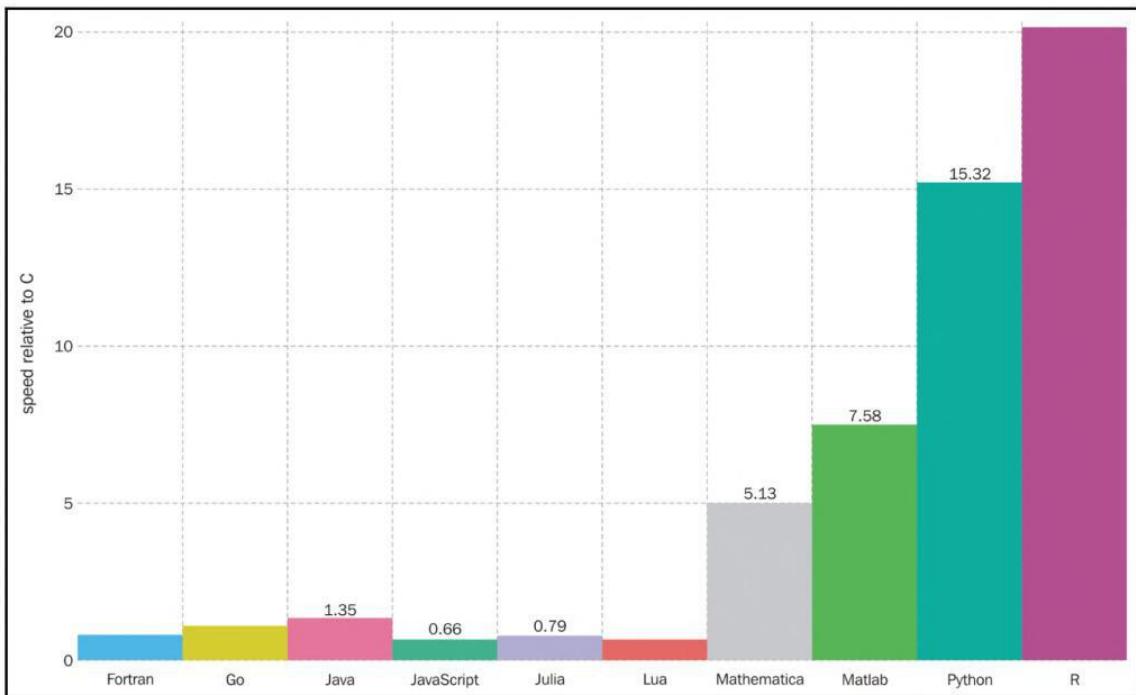
1. Log in or register on our website using your email address and password
2. Let the mouse pointer hover over the SUPPORT tab at the top
3. Click on Code Downloads & Errata
4. Enter the name of the book in the Search box
5. Select the book for which you're looking to download the code files
6. Choose from the drop-down menu where you purchased this book
7. Click on Code Download



Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of the following:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

By timing this code in Julia and C, as well as re-implementing it in many other languages (all of which are available within the *Microbenchmarks* project at <https://github.com/JuliaLang/Microbenchmarks>), we can note that Julia's performance claims are certainly borne out for this small program. Plotting these timing results in the following chart, we see that Julia can perform at a level similar to C and other statically typed and compiled languages:



This is, of course, a micro benchmark, and therefore cannot be extrapolated too much. However, I hope you will agree that it is certainly possible to achieve exceptional performance in Julia, without having to fall back to low-level languages for performance-critical code. The rest of the book will attempt to show how we can achieve performance close to this standard, for many different kinds of code bases. We will learn how to squeeze the maximum possible performance from the CPU, without any of the overhead that typically plagues dynamic languages.

Summary

In this chapter, we noted that Julia is a language that is built from the ground up for high performance. Its design and implementation have always been focused on providing the highest possible performance on a modern CPU.

The rest of the book will show you how to use the power of Julia fully, to write the fastest possible code in this language. In the next chapter, we will discuss how to measure the speed of Julia code, and identify performance bottlenecks. You will also learn about some of the tools that are built into Julia for this purpose.

2

Analyzing Performance

The aim of this book is to show us how to improve the performance of our Julia code, but before we can improve, we must measure. To try and optimize any Julia code we have written, we first need to understand its performance characteristics. Is the code fast enough for our needs? If not, is there an upper limit to how fast it can be? And finally, can we understand where the bottlenecks are, so that we can prioritize where to focus our efforts. This chapter will show us the tools available in Julia to answer these questions and more. In later chapters, we will see how to use this information to improve our code.

In this chapter, we will cover the following topics:

- Timing Julia functions
- Accurate benchmarking
- Profiling Julia functions
- Tracking detailed memory allocation

Timing Julia functions

The first step to understanding anything is to measure it. The same goes for writing high-performance Julia code. We need to measure the performance of the code as the first step to achieving that. As a high-performance language, Julia includes many tools to do this easily, effectively, and accurately. Many of these are built into the language and the standard library, while others are in external packages that can be installed with a single command. All of these tools not only make it easy to measure the performance of the code; they also make it easy to execute the measurement correctly.

When reading this book, whether in print or on screen, we encourage you to run the code and see the results for yourself. The concepts in this book will become much easier to learn if you run the code yourself. The simplest would be to copy/paste the code you see in this book into the Julia REPL.



The REPL (or the Read-Eval-Print-Loop, aka the Julia> prompt) is what you get when you run the Julia executable. It is the best command-line environment you will have seen, with features such as full history, multiline editing, and multiple modes. The code that you see in this book is written as if entered on the REPL. And in an amazing feat of user friendliness, you can copy and paste the entire line, including the text of the prompt (in other words, the julia> text). Upon pasting, the REPL will recognize this, and do the right thing. Do try this!

The @time macro

Whenever you care about the performance of your code, the `@time` macro will end up being one of your most widely used commands on the Julia prompt. Built into the base Julia runtime, this macro wraps the provided expression to calculate and print the elapsed time while running it. It also measures and prints the amount of memory allocated while running that code.

```
julia> @time sqrt(rand(1000));  
0.000023 seconds (8 allocations: 15.969 KB)
```

Any kind of Julia expressions can be wrapped by the `@time` macro. Usually, it is a function call as above, but it could be any other valid expression:

```
julia> @time for i in 1:1000  
           x = sin.(rand(1000))  
       end  
0.023210 seconds (2.00 k allocations: 15.503 MiB, 38.35% gc time)
```



Timing measurements and JIT compiling:

Recall that Julia is a JIT-compiled language. This means that the Julia compiler and runtime compiles any Julia code into machine code the first time it sees it. This means that, if you measure the execution time of any Julia expression that executes for the first time, you will end up measuring the time (and memory use) required compiling this code. So, whenever you time any piece of Julia code, it is crucial to run it at least once prior to measuring the execution time. Always measure the second or later invocation.

Other time macros

An enhanced version of the `@time` macro is also available; this is the `@timev` macro. This macro operates in a manner very similar to `@time`, but measures some additional memory statistics, and provides elapsed time measurements to nanosecond precision. The following output shows the result of running this macro:

```
julia> @timev sqrt.(rand(1000));
0.000012 seconds (8 allocations: 15.969 KB)
elapsed time (ns): 11551
bytes allocated: 16352
pool allocs: 6
non-pool GC allocs: 2
```

Both the `@time` and `@timev` macros return the value of the expression whose performance they measured (note the semicolon at the end of the preceding expression—this prevents the REPL from outputting the return value to the console). Hence, these can be added without side effects to almost any location within the Julia code.

They can be used to measure the performance of the specific expression we are interested in, and still use the computed value for further operations. For example, it could be used within a function call's arguments. In the following expression, the `@time` macro is used to time the execution of the `sqrt` function, and then the result of that function is passed as an argument to the `sum` function:

```
julia> sum(@time sqrt.(rand(1000)))
0.000373 seconds (29 allocations: 17.047 KiB)
656.069185135439
```

`@elapsed` is yet another built-in macro that can be used to measure the execution time of Julia programs. Unlike the `@time` or `@timenv` macros, which output the time information to the console, the `@elapsed` macro returns the time in seconds as a result:

```
julia> @elapsed sqrt.(rand(1000))
0.000217478
```

This means that these resulting times can be used for further processing—for example, they can be used to assert performance limits during unit testing:

```
julia> using Test
julia> @test @elapsed(sqrt.(rand(1000))) <= 10e-4
Test Passed
```

These macros are useful to measure the performance of individual expressions. To fully understand how larger codebases perform, we need a profiler.

The Julia profiler

The Julia runtime includes a built-in profiler, which can be used to measure how long each line of code takes to run, relative to a certain code base. It can therefore be used to identify bottlenecks in code, which can, in turn, be used to prioritize optimization efforts.

This built-in system implements what is known as a **sampling profiler**. As its name suggests, it samples the program call stack at certain points in time. When the profiler is run, it stops and inspects the running system every few milliseconds (by default, 1 millisecond on UNIX, and 10 milliseconds on Windows). At every point, the profiler identifies the list of function calls (and the line of code they originate), from the start of the program to the current point, and updates a counter for every line it sees on the call stack. The idea is that the lines of code that are executed most are also found more often on the call stack. Hence, over many such samples, the count of how often each line of code is encountered will be a measure of how often this code runs.

The primary advantage of a sampling profiler is that it can run without modifying the source program, and thus has very little overhead. The program runs at almost full speed when being profiled. The downside of the profiler is that the data is statistical in nature, and may not reflect exactly how the program executed. However, when sampled over a reasonable period of time (say a few hundred milliseconds at least), the results are accurate enough to form a good understanding of how the program performs, and what its bottlenecks are.

Using the profiler

The profiler lives within the *Profile* standard library package. So the first step in using the profiler is to import its namespace into the current session. You can do this using the `using` command:

```
julia> using Profile
```

This makes the `@profile` macro available. This measures and stores the performance profile of the expression supplied to it.

Do not profile the JIT:



As with measuring the time of execution, remember to run your code at least once before attempting to profile it. Otherwise, you will end up profiling the Julia JIT compiler, rather than your code. If you see many instances of `inference.jl` in your profiler output, that means you are profiling the compiler instead of your code. If you see this, clear the profile data, and run your code again; on the second run, you will get the correct profile results.

To see how the profiler works, let's start with a function that creates 1,000 sets of 10,000 random numbers, and then computes the mean of the squares for each set:

```
using Statistics
function randmsq()
    x = rand(10000, 1000)
    y = mean(x.^2, dims=1)
    return y
end
```

After calling the function once to ensure that all the code is compiled, we can run the profiler over this code as follows:

```
julia> randmsq();
julia> @profile randmsq()
```

This will execute the function while collecting profile information. The function will return as normal, and the collected profile information will be stored in memory.

The output from the profiler is a hierarchical list of code locations, representing the call stack for the program. The number, against each line, counts the number of times this line was sampled by the profiler. Therefore, the higher the number, the greater the contribution of that line to the total runtime of the program. It indicates the time spent on the line, and all its *callees*. If the hierarchy is too deeply nested, thereby making the output confusing, you can get a flat output by calling `Profile.print(format=:flat)`.

The profile information can be printed via the `print` method as follows:

```
julia> Profile.print()
115 ./task.jl:257; (::getfield(REPL,
Symbol("##28#29")){REPL.REPLBackend})()
115 ...r/share/julia/stdlib/v0.7/REPL/src/REPL.jl:116; macro expansion
115 ...r/share/julia/stdlib/v0.7/REPL/src/REPL.jl:85;
eval_user_input(::Any, ::REPL.REPLBackend)
115 ./boot.jl:316; eval(::Module, ::Any)
115 ./<missing>:0; top-level scope
115 .../julia/stdlib/v0.7/Profile/src/Profile.jl:27; macro expansion
53 ./REPL[11]:2; randmsq()
53 ...e/julia/stdlib/v0.7/Random/src/Random.jl:224; rand
53 .../julia/stdlib/v0.7/Random/src/Random.jl:236; rand
53 .../julia/stdlib/v0.7/Random/src/Random.jl:235; rand
8 ./boot.jl:407; Type
8 ./boot.jl:400; Type
8 ./boot.jl:392; Type
45 .../julia/stdlib/v0.7/Random/src/Random.jl:214; rand!
45 ...e/julia/stdlib/v0.7/Random/src/RNGs.jl:447; rand!
...
...
```

What does this output tell us? Well, among other things, it shows that the creation of the random arrays took a majority of the execution time; over a third. Of the remaining time, the majority was spent on the squaring, and a minority on the mean.

There are a few options to the profiler that are sometimes useful, although the defaults are a good choice for most uses. Primary among them is the *sampling interval*. This can be provided as keyword arguments to the `Profile.init()` method. The default delay is 1 millisecond on Linux, and should be increased for very long-running programs through a line of code such as the following (which sets the delay to 100 ms):

```
julia> Profile.init(delay=.01)
```

The delay may be reduced as well, but the overhead of profiling can increase significantly if it is lowered too much.

Finally, you may have realized that the profiler stores its samples in memory in order to be viewed later. In order to profile a different program during a Julia session that is already running, it may be necessary to clear the stored profile from memory. The `Profile.clear()` function does this, and must therefore be run between any two invocations of `@profile` within the same Julia process.

ProfileView

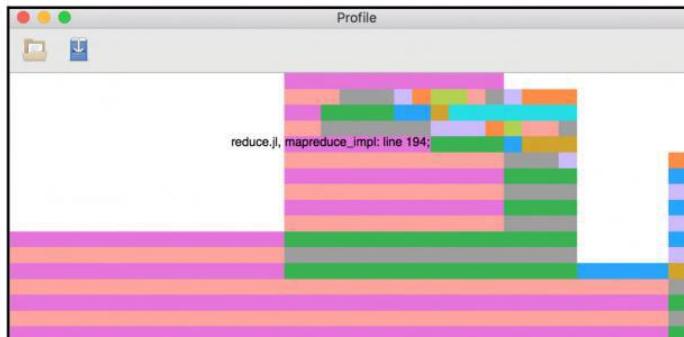
The textual display of the profiler output shown previously is useful and explanatory in many cases, but can become confusing to read for long, or deeply nested call graphs. In this case, or in general, if you prefer a graphical output, the `ProfileView` package is the answer. However, this is not included in the base Julia distribution, and must be installed as an external package via the Julia package manager:

```
julia> using Pkg  
julia> Pkg.add("ProfileView")
```

This will install the `ProfileView` package and its dependencies (which include the Gtk graphical environment). Once installed, it is very simple to use. Simply load the package and call its `view()` function instead of `Profile.print()` after the profile samples have been collected using `@profile`:

```
julia> using ProfileView  
julia> ProfileView.view()
```

A user interface window will pop up, with the profile displayed as a *flame graph*, similar to the following screenshot. Move your cursor over the blocks to note a hover containing the details of the call location:



This view provides the same information as the tree view seen earlier, but may be easier to navigate and understand, particularly for larger programs. In this chart, elapsed time goes from left to right, while the call stack goes from bottom to top. The width of the bar therefore shows the time spent by the program in a particular call location, along with its callees. The bars stacked on top of one another show the hierarchy of function calls. This view of a program's execution profile is commonly known as a **flame graph**.

The `ProfileView` UI provides a few nifty utilities to work with the profile data. The profile itself can be saved to disk using the save icon at the top of the window, while a previously saved profile can be opened by clicking the folder icon. Right-clicking on a bar will cause an editor to open the program at that line, and left-clicking will cause a line describing the call to be printed in the Julia REPL. The latter is an easy way to quickly mark interesting lines, for subsequent analysis.

`ProfileView` also has the ability to create the flame graph in SVG format, which makes it easy to share profiling results with others. SVG is also the default format when `ProfileView` is called from within an IJulia notebook:

```
In [9]: using ProfileView
In [10]: @profile testfunc()
Out[10]: 1x1000 Array{Float64,2}:
 0.289969  0.290125  0.288434  0.288483  ...  0.28958  0.286604  0.288659

In [11]: ProfileView.view()

Out[11]:
```

Profile results

```
dsfmt_fill_array_close_open! in dSFMT.jl:84
rand! in random.jl:433
testfunc in In[1]:2
macro expansion in profile.jl:23; anonymous in &lt;missing&gt;;-1
include_string in loading.jl:515
include_string in Compat.jl:407
execute_request in execute_request.jl:154
eventloop in eventloop.jl:8
#14 in task.jl:335

Function: dsfmt_fill_array_close_open! in ./dSFMT.jl:84
```

From the REPL, the `svgwrite` function can be used to output the graph in SVG format.

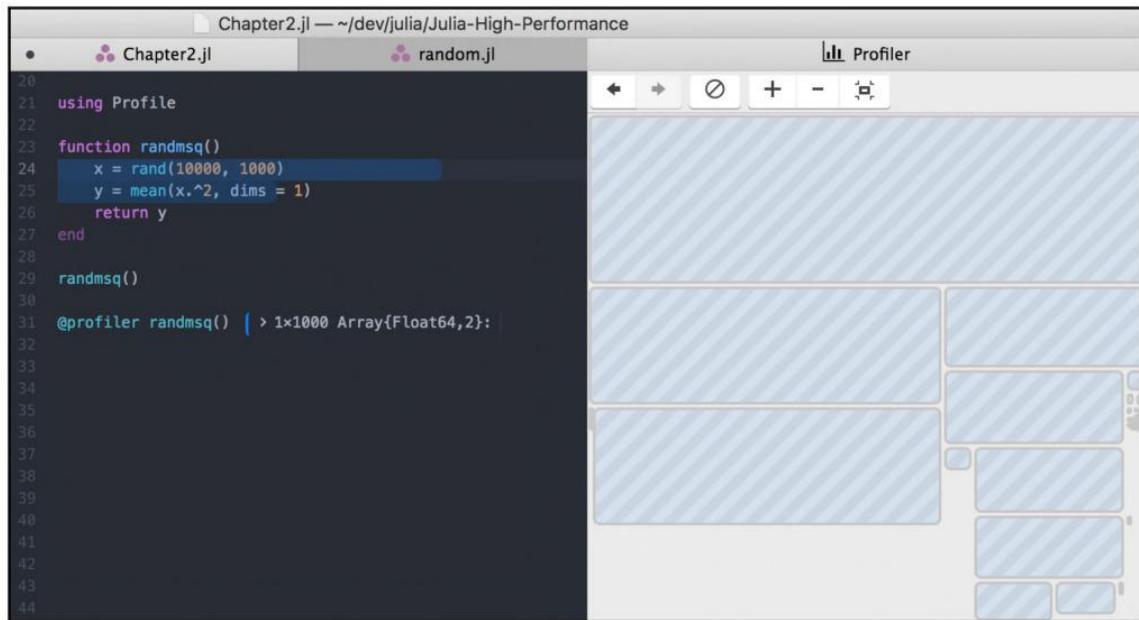
```
julia> ProfileView.svgwrite("profile_results.svg")
```

Using the profiler on the Julia REPL is simple, as we saw in this section, but it is also possible to use it in an integrated manner within an IDE.

Using Juno for profiling

The Juno IDE is a popular environment for developing Julia code. It is bundled with JuliaPro, and can also be installed directly. Among its many productivity enhancing features (such as inline evaluation, code completion, and a built-in debugger) is an integrated profiler. This provides a display of the profiler output on top of the source code view, making it easy to visualize the relative contribution of each line of code to the overall execution time.

While the display is more sophisticated, using the profiler in Juno is similar: use the `@profiler` macro (note the extra `r` at the end). The following screenshot shows an example of the profiler view inside Juno:

A screenshot of the Juno IDE interface. On the left, there is a code editor window titled "Chapter2.jl — ~/dev/julia/Julia-High-Performance". It contains the following Julia code:

```
20
21 using Profile
22
23 function randmsq()
24     x = rand(10000, 1000)
25     y = mean(x.^2, dims = 1)
26     return y
27 end
28
29 randmsq()
30
31 @profiler randmsq() | > 1x1000 Array{Float64,2}:
32
33
34
35
36
37
38
39
40
41
42
43
44
```

Line 24, which contains the line `x = rand(10000, 1000)`, is highlighted with a blue bar above the code. To the right of the code editor is a "Profiler" view window. This window has a toolbar with icons for back, forward, search, and zoom. Below the toolbar is a large area containing a flame graph. The flame graph consists of several nested rectangular boxes of varying sizes, all with a light blue diagonal striped pattern. The largest box is at the top level, and smaller boxes are nested within it, representing the call stack or execution flow.

You will notice the highlight on top of the source code of the function being analyzed, depicting the performance cost of each line as an inline bar. On the right, there is a more traditional flame graph. Hovering on the boxes in the flame graph displays the file and function that it denotes, and clicking on the box will open the relevant source code in the IDE.

Using TimerOutputs

For some complicated and long-running programs, a full profiler run can be complex and confusing. Scrolling through many hundreds of stack frames becomes tedious. In these situations, the `TimerOutput` package helps to easily measure the constituent parts of a program.

To use this package, first install it using Julia's package manager:

```
julia> using Pkg  
julia> Pkg.add("TimerOutputs");
```

Once installed, the package can be loaded. Next, a global `TimerOutput` object is created. This will store the results of our timing runs:

```
julia> using TimerOutputs  
  
julia> const to = TimerOutput();
```

Now, we can time individual parts of the computation. We will reuse the `randmsq` function we wrote previously. We annotate the code inside the function via the `@timeit` macro, which takes as an argument the `TimerOutput` object, a name to refer to each invocation, and the expression to be measured:

```
function randmsq_timed()  
    @timeit to "randmsq" begin  
        x = @timeit to "rand" rand(10000, 1000)  
        y = @timeit to "mean" mean(x.^2, dims=1)  
        return y  
    end  
end
```

We now run this function, and then view the timer outputs using the `print_time` function:

```
julia> randmsg_timed();
```

```
julia> print_timer(to)
```

		Time			Allocations			
Tot / % measured:	Section	ncalls	time	%tot	avg	alloc	%tot	avg
	randmsg	2	438ms	100%	219ms	153MiB	100%	76.3MiB
	rand	1	250ms	57.0%	250ms	76.3MiB	50.0%	76.3MiB
	mean	1	188ms	43.0%	188ms	76.3MiB	50.0%	76.3MiB

The output nicely summarizes the timings for the nested calls, and calculates the aggregates. For long simulations, or complex optimization problems, this way of measuring timings can be very useful.

Analyzing memory allocation

The amount of memory used by a program is sometimes as important to track as the amount of time taken to run it. This is not only because memory is a limited resource that can be in short supply, but also because excessive allocation can easily lead to excessive execution time. The time taken to allocate and deallocate memory and run the garbage collection can become quite significant when a program uses large amounts of memory.

The `@time` macro seen in the previous sections provides information about memory allocation for the expression or function being timed. In some cases, however, it may be difficult to predict where exactly the memory allocation occurs. For these situations, Julia's track allocation functionality is just what is needed.

Using the memory allocation tracker

To get Julia to track memory allocation, start the `julia` process from your command or shell prompt with the `-track-allocation=user` option as follows:

```
$ julia track-allocation=user
```

This will start a normal Julia session in which you can run your code as usual. However, in the background, Julia will track all the memory used, which will be written to `.mem` files when Julia exits. There will be a new `.mem` file for each `.jl` file that is loaded and executed. These files will contain the Julia code from their corresponding source files, with each line annotated with the total amount of memory that was allocated as a result of executing this line.

As we discussed earlier, when running Julia code, the compiler will compile the user code at runtime. Once again, we do not want to measure the memory allocation due to the compiler. To achieve this, first run the code under measurement once, after starting the Julia process. Then, run the `Profile.clear_malloc_data()` function to restart the allocation measurement counters. Finally, run the code under measurement once again, and then exit the process. This way, we will get the most accurate memory measurements.

Statistically accurate benchmarking

The tools described in this chapter, particularly the `@time` macro, are useful for identifying and investigating bottlenecks in our program. However, they are not very accurate in terms of a fine-grained analysis of fast programs. If you want to, for example, compare two functions that take a few milliseconds to run, the amount of error and variability in the measurement will easily swamp the running time of this function.

Using BenchmarkTools.jl

The solution, then, is to use the `BenchmarkTools.jl` package for statistically accurate benchmarking. Install the package via the Julia package manager, and thereafter it is simple to use. Instead of using `@time`, use the `@benchmark` macro. Unlike `@time`, however, this macro can only be used in front of function calls, rather than any expression. It will evaluate the parameters of the function separately, and then call the function multiple times to build up a sample of execution times.

The output will show the mean time taken to run the code, but with statistically accurate upper and lower bounds. These statistics are estimated by evaluating the expression multiple times, with the number of evaluations determined in order to maximize the accuracy of the measurements. These estimates attempt to account for the noise inherent in running benchmarks on real machines, while also minimizing the time taken to measure it accurately. As an example, we measure the running time of creating a random array and calculating the square root of all its elements:

```
julia> using BenchmarkTools

julia> @benchmark sqrt.(rand(1000))
BenchmarkTools.Trial:
  memory estimate: 15.88 KiB
  allocs estimate: 2
-----
  minimum time: 6.266 μs (0.00% GC)
  median time: 7.225 μs (0.00% GC)
  mean time: 9.417 μs (13.12% GC)
  maximum time: 612.404 μs (96.45% GC)
-----
  samples: 10000
  evals/sample: 5
```

A simpler version of the output can be obtained by using the `@btime` macro. This macro does the same operations as the `@benchmark` macro, but provides simpler output that is similar to the basic `@time` macro. Furthermore, it also returns the value of the expression that it evaluated. For the rest of the book, this is what we will use for all time measurements for the code that we write and evaluate. Using the `@btime` macro from the `BenchmarkTools` package will allow us to be confident that any performance improvements to our code that we measure are real, and not noise:

```
julia> @btime mean(rand(1000));
  1.665 μs (1 allocation: 7.94 KiB)e
```



The `BenchmarkTools` package consists of sophisticated machinery to provide statistically accurate benchmarking. The theory behind the code is explained in Jarett Ravel and Jiahao Chen's paper, *Robust benchmarking in noisy environments*, available at <https://arxiv.org/abs/1608.04295>.

These two macros, `@benchmark` and `@bt_time`, should be your standard method to measure performance in Julia. They should be used in almost all cases in which you need to benchmark any code. We will use them almost exclusively throughout this book. All the code in subsequent chapters will assume that the package has been loaded in the session by using `BenchmarkTools`. In rare cases, such as for long-running programs that take too long and cannot be executed multiple times, you may fall back to the `@time` macro. However, such occasions should be rare.

Summary

In this chapter, we discussed how to use the available tools to measure the performance of Julia code. We learned to measure the time and memory resources used by code, and understood how to arrive at the hotspots for any program.

In subsequent chapters, we will learn how to remedy the issues we encounter using the tools of this chapter, and hence improve the performance measurements for our code.

3

Types, Type Inference, and Stability

Julia is a dynamically typed language. Unlike languages such as Java or C, the programmer does not need to specify the fixed type of every variable in the program. Yet, somewhat counterintuitively, Julia achieves its impressive performance characteristics by inferring and using the type information for all the data in the program. In this chapter, we will start with a brief look at the type system in the language and then explain how to use this type system to write high-performance code.

This chapter will cover the following topics:

- The Julia type system
- Type inference
- Type stability
- Types at storage locations

The Julia type system

Types in Julia are essentially tags, on values, that restrict the range of potential values that can possibly be stored at that location. Being a dynamic language, these tags are relevant only to runtime values. Types are usually not enforced at compile time; rather, they are checked at runtime. However, type information is used at compile time to generate specialized methods for different kinds of function arguments.

Using types

In most dynamic languages, types are implicit in terms of how values are created. Julia can be—and usually is—written in this way, with no explicit type annotations. However, optionally, you can indicate variables or function parameters to be restricted to specific types using the `::` symbol. Here are a few examples.

We define two versions of the `iam` function, one for integer arguments and another for string arguments. We also define a single method for the function `addme`, which takes two unrestricted values of any kind as an argument, as follows:

```
#Declare type of function argument
iam(x::Integer) = "an integer"
iam(x::String) = "a string"

function addme(a, b)
    #Declare type of local variable x
    x::Int64 = 2
    #Type of variable y will be inferred
    y = (a+b) / x
    return y
end
```

Having defined these functions, we can now call them. These calls should make clear how Julia dispatches function calls based on the types of the argument values, as follows:

```
julia> iam(1)                                #Dispatch on type of
argument
"an integer"

julia> iam("1")                               #Dispatch on type of
argument
"a string"

julia> iam(1.5)                               #Dispatch fails
ERROR: `iam` has no method matching iam(::Float64)
```

A note on terminology

In Julia, the abstract operation represented by a name is called a function, while the individual implementations for specific types are called methods. Thus, in the preceding code, we can use the `iam` function and the `iam` methods for `Integer` and `String`. In an object-oriented language, objects have methods; in Julia, functions have methods.

