

**FACULTY  
OF MATHEMATICS  
AND PHYSICS**  
Charles University

**MASTER THESIS**

Roman Borufka

**Performance Testing Suite for Unity  
DOTS**

Department of Software and Computer Science Education

Supervisor of the master thesis: Mgr. Jakub Gemrot, Ph.D.

Study programme: Computer Science

Study branch: Computer Graphics and Game Development

Prague 2020

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Prague, 6th January 2020

signature of the author

I would like to express my gratitude to Mgr. Jakub Gemrot, Ph.D. for his help and valuable advice during writing this thesis.

Title: Performance Testing Suite for Unity DOTS

Author: Roman Borufka

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Jakub Gemrot, Ph.D., Department of Software and Computer Science Education

Abstract: Game developers are searching for new ways of writing high performance code in order to adapt to trends in hardware development. Unity's relatively new DOTS system has introduced a new way, how to write code in order to fully exploit all aspects of modern processors, e.g. multithreading or SIMD instructions. The thesis focuses on creation of a generally-usable performance testing suite in order to benchmark the performance of various features of Unity DOTS system. Based on the results of the benchmarks a list of recommendations for writing high-performance solutions in Unity is compiled. The recommendations are evaluated in a real-time boids simulation.

Keywords: high-performance systems, performance testing, video games development, Unity, DOTS

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Thesis Structure . . . . .	3
<b>2</b>	<b>Background</b>	<b>4</b>
2.1	Concurrency and Parallelism . . . . .	4
2.1.1	Data and Task Parallelism . . . . .	4
2.1.2	Flynn’s Taxonomy . . . . .	4
2.1.3	SIMD Parallelism . . . . .	5
2.1.4	Multithreading . . . . .	7
2.2	Writing High-Performance Code in C# . . . . .	8
2.2.1	Structs . . . . .	8
2.2.2	Other Features . . . . .	9
2.3	Unity DOTS . . . . .	10
2.3.1	Entity Component System . . . . .	11
2.3.2	C# Job System . . . . .	12
2.3.3	Burst Compiler . . . . .	15
2.4	Measuring Time . . . . .	17
2.4.1	Stopwatch . . . . .	18
2.4.2	Unity Profiler . . . . .	19
2.4.3	Unity Performance Testing . . . . .	20
2.4.4	Unity Editor vs Standalone Build . . . . .	22
<b>3</b>	<b>Related Work</b>	<b>23</b>
3.1	Unity DOTS Benchmark . . . . .	23
3.2	Analyzing Burst Generated Assemblies . . . . .	24
3.3	Burst Benchmarks . . . . .	24
3.4	Native Memory Allocators . . . . .	25
3.5	Conclusion . . . . .	25
<b>4</b>	<b>Implementation</b>	<b>26</b>
4.1	Tests and Test Sets . . . . .	26
4.2	Format of the Performance Testing Suite . . . . .	26
4.3	Test Framework . . . . .	28
4.3.1	Tests and Test Sets . . . . .	29
4.3.2	Test Set Runner . . . . .	29
4.4	Measurement Procedure . . . . .	29
4.5	Extensibility . . . . .	30
<b>5</b>	<b>Results</b>	<b>31</b>
5.1	Wrapper Test Set . . . . .	32
5.2	Accessor Test Set . . . . .	34
5.3	Allocator Test Set . . . . .	36
5.4	Batch Size Test Sets . . . . .	38
5.4.1	Float Assignment . . . . .	38
5.4.2	Matrix Multiplication . . . . .	41

5.5	Job Type Test Sets . . . . .	43
5.5.1	Float Assignment . . . . .	43
5.5.2	Matrix Multiplication . . . . .	45
5.6	Float Optimization Test Sets . . . . .	47
5.6.1	Float Mode Test Sets . . . . .	47
5.6.2	Float Precision Test Sets . . . . .	50
5.7	Math Test Sets . . . . .	52
5.8	Recommendations . . . . .	55
5.9	Evaluation of the recommendations . . . . .	56
<b>6</b>	<b>Conclusion</b>	<b>58</b>
6.1	Future Work . . . . .	58
	<b>Bibliography</b>	<b>59</b>
	<b>List of Figures</b>	<b>61</b>
	<b>List of Tables</b>	<b>62</b>
	<b>List of Abbreviations</b>	<b>63</b>
<b>A</b>	<b>User Documentation</b>	<b>64</b>
A.1	Unity Editor . . . . .	64
A.1.1	System Requirements . . . . .	64
A.1.2	Installation Instructions . . . . .	65
A.2	PTS Windows Standalone Build . . . . .	65
A.2.1	System Requirements . . . . .	65
A.2.2	Installation Instructions . . . . .	65
A.3	PTS Android Standalone Build . . . . .	66
A.3.1	System Requirements . . . . .	66
A.3.2	Installation Instructions . . . . .	66
A.4	Boid Windows Standalone Build . . . . .	66
A.4.1	System Requirements . . . . .	66
A.4.2	Installation Instructions . . . . .	66
<b>B</b>	<b>All Test Results</b>	<b>67</b>

# 1. Introduction

In past decades, there were great hardware advancements in generally available PCs and mobile devices. These improvements came mostly in the form of hardware parallelism - SIMD instructions and multi-core processors. Many game developers adapted to this trend by making their game engines more focused on data-oriented design and multithreading in order to get the best performance for their games. One such example is the recently introduced Unity DOTS system, which is a part of Unity game engine. The source code of Unity DOTS is not available and the performance of its various features is not perfectly clear. Moreover, the system is not fully documented, since some its features are still in preview.

The aim of the thesis is to create a generally usable performance testing suite for running custom user tests, in which the performance of Unity DOTS will be evaluated. The evaluation will be done by creating a set of tests, which stress the respective parts of the system. Based on the results of the tests, a list of recommendations for implementing high-performance solutions in Unity will be compiled. Finally, the recommendations will be evaluated within a real-time simulation of boids.

## 1.1 Thesis Structure

The thesis is structured as follows: Chapter 2 will describe the theoretical background in the field of concurrency and parallelism. Moreover, it presents the high performance features of C# and introduces the Unity DOTS system in general. Chapter 3 will review the related work in the field of the Unity DOTS. Chapter 4 will describe format and the implementation details of the performance testing suite for Unity DOTS. Chapter 5 describes the tests, which are part of the performance testing suite. It presents and compares its results and suggests the recommendations to be taken into account, when writing high-performance solutions in Unity. Subsequently, it evaluates the recommendations on a real-time boids simulation. Finally, Chapter 6 makes a conclusion and proposes possible directions for future work in this field.

# 2. Background

## 2.1 Concurrency and Parallelism

In order to take full advantage of modern multicore CPUs with vector instruction sets, it is necessary to understand two different terms, related to development for these platforms, which are nowadays often used to complement each other - concurrency and parallelism.

Concurrent computing executes multiple computations on shared data in overlapping time periods, in contrast to sequential computing, which runs operations only from one flow of control. The implementation of concurrent software might be a single process running multiple threads or multiple processes running on one or several computers. Reading and writing shared data, however, can yield incorrect results in some cases. One example of that might be a race condition, which arises when two or more flows of control are capable of writing into shared data in different order and therefore cause different results (Gregory [2018]).

Parallel computing executes computations on multiple hardware components at the same time, meaning it is able to run operations on multiple tasks at the same time simultaneously. Nowadays, modern computers contain parallel hardware in many forms, e.g. multicore CPU processors or GPUs (Gregory [2018]).

In general, parallelism can be divided into categories by many factors. We will classify forms of parallelism firstly by division into data and task parallelism and secondly by division according to the Flynn's taxonomy.

### 2.1.1 Data and Task Parallelism

Data parallelism refers to a single operation being run on a multiple data sets in parallel, e.g. matrix multiplication on many matrices in the animation module of the game engine. On the other hand, task parallelism means performing several operations on different tasks, e.g. running collision and animation tasks in parallel (Gregory [2018]).

### 2.1.2 Flynn's Taxonomy

Another approach how to classify parallelism is according to the Flynn's taxonomy, proposed by Michael J. Flynn in 1966. Parallelism can be divided into four categories defined by the number of parallel instruction streams and number of data streams (Gregory [2018]).

- **Single instruction, single data (SISD)** - typical on older single core

PCs, only single data item is processed at a time

- **Single instruction, multiple data (SIMD)** - CPUs with vector instructions performing computations on multiple data at once
- **Multiple instruction, single data (MISD)** - very rarely used in game engines
- **Multiple instruction, multiple data (MIMD)** - multicore CPUs and distributed systems

### 2.1.3 SIMD Parallelism

Single instruction multiple data (SIMD) instructions or vector instructions are modern CPU instructions, which are capable of performing operation on multiple data in parallel. In order to utilise the SIMD parallelism, processor manufacturers introduced several vector instruction sets with their processors.

In following subsections, first, SIMD instructions for Intel processors will be described. Moreover, the data alignment for SIMD instructions will be outlined. Finally mobile vector instruction sets will be mentioned.

#### Intel Vector Instruction Sets and Registers

The first generation of vector instruction set by Intel is the MMX instruction set, which appeared in 1994. The width of MMX registers is 64 bits and vector instructions could operate on either two 32-bit integers, four 16-bit integers or eight 8-bit integers (Gregory [2018]).

In 1999, Intel introduced its next instruction set, which is named Streaming SIMD Extensions (SSE). The SSE registers are 128-bits wide and the supported data types for the instructions are 32-bit floats. As a result, the calculations, which are running on four 32-bit floats, are frequently used by the game engines. Furthermore, several extensions to the SSE instruction set were released - SSE2, SSE3, SSSE3 and SSE4. The SSE2 extension permitted the instructions to perform calculations additionally on two 64-bit floats, two 64-bit integers, four 32-bit integers, eight 16-bit integers or sixteen 8-bit bytes. SSE registers are called XMM0 through XMM15 (Gregory [2018]).

Another instruction set, released in 2011, was the Advanced Vector Extensions (AVX). The width of AVX registers is 256 bits, which allows the calculations to run on eight 32-bit floats or four 64-bit floats. As with the SSE, two extensions were released - AVX2 and AVX-512. AVX2 extended the AVX instruction set to run calculations also on integers. AVX-512 introduced a 512-bit wide registers, which can store sixteen 32-bit floats or eight 64-bit floats. AVX registers are known as YMM0 through YMM15, while AVX-512 registers are ZMM0 through ZMM32 (Gregory [2018]).

## Data Types and Alignment

Modern compilers supply specific data types to facilitate working with the SSE and AVX data, e.g. in C++, the `_m128` data type is an array of four floats used by SSE and the `_m256` is an array of eight floats, that is used by AVX. These data types are used for transferring the data from RAM to the vector registers in CPU. Prior to the transfer, data must be aligned on a particular boundary. For XMM registers, data must be 16-byte aligned, data for YMM registers has to be 32-byte aligned and ZMM register data must be aligned on 64-byte boundary. Local variables, global variables and member variables of types `_m128`, `_m256`, etc. are aligned automatically by the compiler. However, compiler does not ensure proper alignment for variables allocated dynamically on the heap and therefore the data must be aligned manually (Gregory [2018]).

## Compiler Intrinsic Functions

In order to simplify writing SIMD code, modern compilers introduce intrinsic functions (also called intrinsics). Intrinsic is inlined assembly code generated by the compiler, which is called as a regular C function and therefore it replaces the need of writing error-prone assembly code (Gregory [2018]).

Below are some sample intrinsic functions:

Sample SSE intrinsic function prototypes in C++ that add, subtract, multiply or divide four pairs of 32-bit floats:

```
_m128 _mm_add_ps(_m128 a, _m128 b);  
_m128 _mm_sub_ps(_m128 a, _m128 b);  
_m128 _mm_mul_ps(_m128 a, _m128 b);  
_m128 _mm_div_ps(_m128 a, _m128 b);
```

Sample C++ AVX intrinsic function prototypes that add, subtract, multiply or divide eight pairs of 32-bit floats:

```
_m256 _mm256_add_ps(_m256 a, _m256 b);  
_m256 _mm256_sub_ps(_m256 a, _m256 b);  
_m256 _mm256_mul_ps(_m256 a, _m256 b);  
_m256 _mm256_div_ps(_m256 a, _m256 b);
```

## Vector Instruction Sets on Mobile Devices

On portable devices, such as smartphones and tablets, where ARM processors are typically used, it is possible to make use of vector instruction sets and intrinsic functions as well. On Android platform it is possible to use ARM Advanced SIMD, also known as Neon, which is an extension of the instruction set for ARMv7 and ARMv8 (Google [2019]).

Neon provides 32 registers, each 128-bit wide. Apart from the auto-vectorization feature, which makes code optimizations, Neon also comes with intrinsic functions, which are accessible from C or C++ (Arm Limited [2019]).

Below are Neon intrinsic function prototypes in C++ that add or subtract four pairs of 32-bit floats:

```
float32x4_t vaddq_f32 (float32x4_t a, float32x4_t b)
float32x4_t vsubq_f32 (float32x4_t a, float32x4_t b)
```

## 2.1.4 Multithreading

In order to utilise parallelism in the game engine, it is necessary to shift our sequential program into a concurrent program. Various tasks have to be decomposed into several subtasks, in order to execute them in parallel. Tasks can be decomposed in several ways, however, in general the decomposition could be done either by task parallelism or data parallelism (Gregory [2018]).

Simple example of the task parallelism approach would be assigning a thread to each subsystem of the game engine, e.g. collision subsystem, render subsystem, animation subsystem etc. Master thread would be responsible for controlling and synchronizing subsystem threads, while having its share on the game logic as well. Running this logic on a CPU with multiple cores allows for the parallelization. However, this approach results in having more threads than CPU cores and therefore each core would have to share several threads. Furthermore, different engine subsystems require different amount of time for running its tasks and some subsystems depend on other subsystems. As a result, some threads might not be fully utilized for long periods of time (Gregory [2018]).

An example of data parallelism would be a scatter and gather method. This approach divides the work on the large amount of data into several batches and therefore it is usable for processing large amount of data, e.g. animation data. In the context of the game engine, several threads are spawned, the calculations are executed and then the threads are joined with the master thread. However, creating a new thread is an expensive operation and therefore creating a new thread for every scatter and gather computation is not practical (Gregory [2018]).

## Job Systems

A job system is a system used to run multiple general tasks on multiple CPU cores in parallel. Instead of creating a thread for every task, the job system is responsible for storing a queue of jobs and planning their execution on currently available CPU cores. The interface of a typical job system contains a function for creating the job and the function, which waits for another job termination. As a result, it is similar to the interface for running threads (Gregory [2018]).

## 2.2 Writing High-Performance Code in C#

Nowadays, writing C#/.NET code in managed environment is much easier in comparison to writing code in languages such as C++, which do not provide the benefits of managed environment such as automatic garbage collection, strong type checking or checking of array bounds.

Using the abstractions provided by the .NET libraries means an additional overhead. However, if programmers give up on certain features of the language and use only a high-performance subset of C#, it is possible to write a code, which is comparable in performance to its C++ equivalent. Below, several features of C# language will be described and their performance will be assessed.

### 2.2.1 Structs

When writing high-performance code in C#, using struct instead of class keyword when appropriate might have dramatical impact on the code performance.

Struct only allocates memory for its members and therefore there is no additional memory overhead. When struct is used as a local variable, it is allocated on the stack. If it is a member of a class, it is a part of its memory and exists on the heap. Therefore, structs do not have to be garbage collected. Passing it to a method, causes the struct to be passed by value. As a result, passing large structures can be quite inefficient (Watson [2014]).

Class, on the other hand, allocates 4 bytes for the pointer in 32-bit processes or 8 bytes in 64-bit processes. Furthermore, it allocates memory for its members and it has an extra memory overhead - 8 bytes (32-bit processes) or 16 bytes (64-bit processes). Passing objects to methods is via copy of the pointer, therefore the cost of passing it is very cheap and irrespective of the size of the object. Since, objects reside on the heap, the garbage collection is applied to them (Watson [2014]).

When using arrays, there is a large difference in memory requirements and efficiency when using an array of structs versus an array of objects. The array of structs not only uses less memory, but also garbage collections occurs less often. Moreover, there is benefit of cache-friendly memory layout, which results in shorter access times for larger arrays. When using array of objects, it is necessary to dereference the pointer for each object, which points to a different parts of heap memory and therefore cache cannot be utilised efficiently (Watson [2014]).

Due to the different behaviour of structs and classes when passed to a method, it is recommended to keep the struct size relatively small.

## Ref Keyword with Structs

In order to be able to avoid passing structures by value. It is possible to use the `ref C#` keyword, which has several meanings. Firstly, it can be used to pass the method argument by reference by using it in the signature of the method and the method call. When a struct is passed to a method by reference, no boxing occurs (Microsoft [2019a]).

Moreover, the `ref` keyword can be used to return a value from the method by reference by using it in the method signature. Furthermore, it can be used to store a local variable as a reference. The `ref` local variables and `ref` return values are supported from C# 7.0. Finally, it can be used in conjunction with the `struct` keyword to indicate that the struct is created on the stack and cannot be used on the heap (Microsoft [2019a]; (Microsoft [2019b])).

### 2.2.2 Other Features

It is recommended to avoid usage of virtual methods, as they cannot be inlined in every case by the JIT compiler and this might introduce unnecessary overhead (Watson [2014]).

Boxing is another C# feature that users who are writing C# code with high-performance in mind should avoid. It refers to a situation, when a primitive type or a struct is wrapped into an object, which is created on the heap, e.g. in order to be passed to methods as an object reference parameter. On the other hand, unboxing means obtaining the value type from the object. Boxing requires an additional overhead for allocation, copying and casting of the object. Moreover, the creation of the object on the heap, results in more work for the garbage collector (Watson [2014]).

Another case worth mentioning is using `for vs foreach` loops. Often, `foreach` loops are converted into standard `for` loops. However, when using `foreach` loops (abstraction of a loop) with `IEnumerable<T>` (abstraction of an array) it results in code with virtual calls, `try-finally` block and other overhead. Consequently, using standard `for` loops is faster or equally fast as `foreach` loops in all cases (Watson [2014]).

Casting objects causes an additional overhead as well, when the object is cast to a valid child object. However, casting to an incorrect child object can have significant impact, since it throws an exception. High-performance C# code should avoid throwing exceptions as much as possible, as it is very expensive. Using them on regular basis might cause major performance issues (Watson [2014]).

## 2.3 Unity DOTS

Unity is a game engine, developed by Unity Technologies, used for creation of 2D and 3D video games. Unity supports several platforms ranging from Windows and MacOS through consoles such as PlayStation 4 and Xbox One, mobile devices such as Android and iOS to web platform WebGL.

Traditional game design in Unity involves creating a Game Object and then attaching components to it. Components contain both data and behaviour, e.g. Transform component, Collision component or Renderer component. To define custom components, user can create C# MonoBehaviour scripts, where custom data and behaviour can be defined. However, this approach has several disadvantages. Firstly, less frequent reuse of code, owing to the tight coupling of data and behaviour. Moreover, components are usually referencing other components, which are reference types, and therefore components are spread on many places on heap in memory. Consequently, data cannot be processed by SIMD instructions, which in turn prevents programmers from writing high-performance code (Ferreira and Geig [2018]).

Recently, Unity introduced its Data-Oriented Technology Stack (DOTS), which is a set of features for Unity game engine. It aims to tackle performance problems in traditional game design workflow in Unity. DOTS components comprise:

- **Entity Component System (ECS)** for writing data-oriented and high-performance code
- **C# Job System** for executing safe and efficient multithreaded code by splitting the work into jobs, which are processed by worker threads, as described in subsection 2.1.4.
- **Burst Compiler** for generating optimized code using SIMD instructions for several platforms, as described in subsection 2.1.3.

By using DOTS features, users are able to write optimized data-oriented code running on multiple threads and using SIMD instructions. As a result, using DOTS brings several advantages such as substantial boost in performance, increased number of objects in game world, decreased battery consumption on mobile devices and better code reusability coming from data-oriented design (Unity Technologies [2018b]).

In order to be able to use features of the DOTS system, it is required to install several Unity packages - Burst, Collections, Entities, Jobs and Mathematics. For the installation guide refer to Appendix A.1.2.

### 2.3.1 Entity Component System

Entity Component System is a different approach to code writing. Specifically, it focuses on writing C# code in data-oriented manner as opposed to traditional object-oriented programming (Unity Technologies [2018b]).

As the name suggest, ECS consists of three parts:

- **Entity** is a mere unique identifier, therefore an integer.
- **Component** contain only raw data, e.g. Movement component or Render component. Components do not contain any behaviour.
- **System** performs the operations on components, e.g. Movement system, Sound system, etc.

By writing data-oriented ECS code, the data and the behavior are strictly separated. Referencing objects in various parts of memory is not needed, since data is stored in a contiguous piece of memory. Therefore, data access time is shorter and it is possible to run SIMD instructions on it (Ferreira and Geig [2018]).

Furthermore, with ECS, the programmer is able to exploit other features of DOTS - C# Job System and Burst Compiler and therefore utilise full potential of modern processors with multiple cores and SIMD instructions (Unity Technologies [2018b]).

Below is an example of traditional MonoBehaviour component, which updates the position of the object. The Update method of the component is called every frame and moves the owning game object in the direction of axis x by the number of units stored in the speed variable:

```
public class MovementComponent : MonoBehaviour
{
    private float speed;

    void Update()
    {
        Vector3 position = transform.position;
        position.x += speed;
        transform.position = position;
    }
}
```

On contrary, compare it to the simplified ECS code below, which stores only data in the components and the logic is contained within systems (for the definition of NativeArray type, see subsection 2.3.2). The OnUpdate method is, again, called every frame and updates the Translation component of all entities with Translation and Movement components. The value of Translation component is shifted in the direction of axis x by the number of units stored in the

speed variable of Movement component. This, in turn, results into moving the entities:

```
public struct Movement : IComponentData
{
    public float speed;
}

public struct Translation : IComponentData
{
    public float3 Value;
}

public class MovementSystem : ComponentSystem
{
    public NativeArray<Translation> translation;
    public NativeArray<Movement> movement;

    protected override void OnUpdate()
    {
        for (int i = 0; i < translation.Length; i++)
        {
            float3 position = translation.Value[i];
            position.x += movement[i].speed;
            translation.Value[i] = position;
        }
    }
}
```

### 2.3.2 C# Job System

C# Job System gives users the opportunity to utilise multiple cores in modern processors. It provides a safe instruments to write high-performance multithreaded code next to native engine jobs, since C# jobs are running inside Unity Native C++ Job system. As a result, user can avoid bugs arising from multithreading such as race conditions and the shared job environment prevents from instantiating excessive number of threads, which negatively impacts code execution time, because of context switching (Unity Technologies [2018b]).

#### Safety System and Native Container

In order to avoid race conditions, the C# Job System ensures that the data are transferred from main thread to the worker thread of the job via copy. It copies a block of data byte by byte and therefore it can copy only blittable types, which are the types with the same representation in the managed and native code (Unity Technologies [2018d]).

NativeContainer is a native memory wrapper, which can be used in a managed environment. This provides jobs with a way to access data on the main thread instead of copying. Unity provides the users with a NativeArray type by default. Moreover, the Unity ECS comes with several other native containers such as NativeList (resizable native array), NativeHashMap (native container for key, value pairs), etc (Unity Technologies [2018c]).

The safety system of the C# Job System tracks the reads and writes into NativeContainer types and performs checks race condition checks, it also runs checks on deallocation and out of bounds access. In case any check fails, Unity outputs an error describing, how it occurred and how to solve it. This feature is, however, available only in the Unity Editor (Unity Technologies [2018c]).

NativeContainer can be decorated with a [ReadOnly] attribute in order to allow multiple jobs reading from it simultaneously. Otherwise, job has both read and write access to the NativeContainer, and the jobs have to be scheduled with dependencies to other jobs using the same NativeContainer, which can decrease performance (Unity Technologies [2018c]).

Upon creation of NativeContainer an allocator type has to be specified. Currently, three allocator types are available (Unity Technologies [2018c]):

- Allocator.Temp - the fastest allocator, the NativeContainer has to be disposed before another callback from native to managed code
- Allocator.TempJob - slower allocator, thread-safe, to be used with jobs, maximum lifetime of four frames
- Allocator.Persistent - the slowest allocator, no limited lifetime, should not be used in performance-critical code

## Job Creation and Scheduling

In order to create a job, which is meant to be used within C# Job System, users have to create a struct implementing the IJob interface. Moreover, the structure needs to contain variables of blittable types or NativeContainer types, which represent the data processed by the job. Finally, the Execute method responsible for processing the data must be created (Unity Technologies [2018a]).

The job has to be scheduled on main thread in order to run in on a worker thread. In order to schedule a job, firstly, it needs to be instantiated, secondly the job member variables have to be filled with appropriate data and finally, the Schedule method should be called (Unity Technologies [2018e]).

Below, a simple example of a job, which assigns floats from one NativeArray to another, can be seen:

```

public struct Job : IJob
{
    [ReadOnly]
    public NativeArray<float> input;
    [WriteOnly]
    public NativeArray<float> output;

    public void Execute()
    {
        for (int i = 0; i < input.Length; i++)
        {
            output[i] = input[i];
        }
    }
}

```

In the example below, there is a parallel job doing the same operation as the previous job. However, it is running on multiple worker threads:

```

public struct ParallelJob : IJobParallelFor
{
    [ReadOnly]
    public NativeArray<float> input;
    [WriteOnly]
    public NativeArray<float> output;

    public void Execute(int i)
    {
        output[i] = input[i];
    }
}

```

The following example shows, how the the aforementioned jobs are scheduled from the main thread. The job implementing the IJob interface can be scheduled by calling the Schedule method without any parameters. However, to schedule a parallel job, it is required to call the Schedule method with the number of indices to be processed and the size of the batches it should be divided into. Based on the value of the batch size parameter, the work is automatically divided into worker threads <sup>1</sup>:

```

// Create input job data
var src = new NativeArray<float>(1000, Allocator.TempJob);

// Create output job data
var dst = new NativeArray<float>(1000, Allocator.TempJob);
var parallelDst = new NativeArray<float>(1000,
                                         Allocator.TempJob);

```

<sup>1</sup><https://docs.unity3d.com/ScriptReference/Unity.Jobs.IJobParallelFor.html> (Accessed 22th December 2019)

```

// Create the jobs and fill its data
var job = new Job
{
    input = src ,
    output = dst
}

var parallelJob = new ParallelJob
{
    input = src ,
    output = parallelDst
}

// Schedule the jobs
var handle = job.Schedule();
var parallelHandle = parallelJob.Schedule(src.Length , 64);

// Wait for the jobs to complete
handle.Complete();
parallelHandle.Complete();

// Free the memory int native containers
src.Dispose();
dst.Dispose();
parallelDst.Dispose();

```

### 2.3.3 Burst Compiler

Burst compiler generates highly optimized native code from Intermediate Language (IL)/.NET bytecode using C# jobs as an input. Burst compiler optimizes for the targeted platform without the need to manually tweak assembly code or using intrinsics. Aim of the burst compiler is to be used effectively with the high-performance C# code and the C# Job system. To compile a job by Burst compiler, it suffices to add the [BurstCompile] attribute to the job, see the example below (Unity Technologies [2019c]).

```

[BurstCompile]
public struct Job : IJob
{
    public void Execute()
    {
        // Actual job code
    }
}

```

Burst compiler optimizes the code based on the properties of the user's processor and therefore the code is capable of running computation on multiple data at once

by using SIMD instructions (Ferreira and Geig [2018]).

Burst has several compiler options, which can be applied to the [BurstCompile] attribute. One of them is an option of different precision for the math functions, e.g. sine or cosine, which can be set through the FloatPrecision flag, e.g. FloatPrecision.High. Another option is to use different FloatMode flag to allow Burst rearrange the instructions or use less precise SIMD instructions, e.g. FloatMode.Fast (Unity Technologies [2019c]).

Burst compiler supports only a subset of .NET using only elements of high-performance C# code, which does not support any reference type objects (see section 2.2). More specifically, the subset contains primitive types:

- bool
- sbyte and byte
- short and ushort
- int and uint
- long and ulong
- float
- double

However, Burst does not work on strings, because string is a managed type. Moreover, it is not possible to use char type as well, although it should be available in a future release. Additionally, specific storage enum types are also supported. Furthermore, struct types are also allowed given that its fields are also supported. Regarding the struct layout, LayoutKind.Sequential and LayoutKind.Explicit are usable in this context. Managed arrays are not allowed, for the exception of reading from a readonly static field. Instead of managed arrays, a native container, which can be used by Burst such as `NativeArray<T>`, should be used in this case. Burst also allows for its own vector types from `Unity.Mathematics` namespace, e.g. `float3`, `int4`, etc. The vector types can then be easily used by SIMD instructions. Generic types are available to use inside structs. Pointer types to types supported by burst are also allowed as well as `System.IntPtr` and `System.UIntPtr` (Unity Technologies [2019c]).

Regarding the language support, basic C# control flows, e.g. `if`, `else`, `switch`, `for`, `while`, `break` and `continue` are supported. Coding in C# with `unsafe` and pointers is allowed as well. It is also possible to pass structs to functions by `ref` or `out` parameters. Internal calls and `DllImport` calls are also available. Extension methods can be used as well in the context of Burst. However, some C# features are not supported by the Burst such as `try`, `catch` and `finally`. `Foreach` iteration cannot be used as well, since it requires `try`, `catch`, `finally` flow (Unity Technologies [2019c]).

It is possible to remove methods inside a job from the burst compilation by adding a [BurstDiscard] attribute to it. This way, managed code can be accessed, i.e. Unity log. However, the method cannot contain any ref or out parameter nor have a return type (Unity Technologies [2019c]).

Burst is using a dynamic dispatch at runtime for its jobs by considering CPU features. Currently, Burst compiler supports SIMD instructions up to SSE4 for Intel CPUs. For desktop platforms (Windows, macOS or Linux) Burst selects SSE2 or SSE4 instruction set, e.g. for CPU with SSE3, it opts for SSE2. For other platforms refer to Burst User Guide<sup>2</sup> (Unity Technologies [2019c]).

Another feature together with Burst that Unity provides is Burst Inspector. Burst Inspector displays to users all the compilable jobs, which can be viewed in following modes (Unity Technologies [2019c]):

- Burst generated optimized native assembly code
- .NET Intermediate Language (IL) code of the job
- Intermediate representation (IR) of LLVM compiler without optimizations
- IR of LLVM compiler with optimizations
- LLVM compiler optimization diagnostics

In contrast with the Burst compiler runtime, Burst Inspector additionally supports AVX, AVX2 and AVX512 disassembly view (Unity Technologies [2019c]).

Figure 2.1 depicts Burst inspector. The left part displays all the jobs in the Unity project. The Burst jobs are highlighted by black color and user can select one of them. In the upper right part, the job display mode can be chosen. The bottom right part displays the Burst job code depending on the mode selected. In this specific example, the Burst generated native code is displayed.

## 2.4 Measuring Time

A usual part of every game engine is a set of tools for profiling the code, i.e. obtaining information about performance of the application. The tools range from a simple timer to a complex profiler displaying information in UI.

It is possible to query system time on most operating systems, e.g. `time()` function in C standard library. However, the usability in game engines is questionable, since it lacks adequate time resolution. Another option is to use a high-resolution timer, which is nowadays included in all modern processors. The implementation is usually a hardware register measuring the CPU cycle count and therefore the

---

<sup>2</sup><https://docs.unity3d.com/Packages/com.unity.burst@1.1/manual/index.html#burst-aot-requirements>

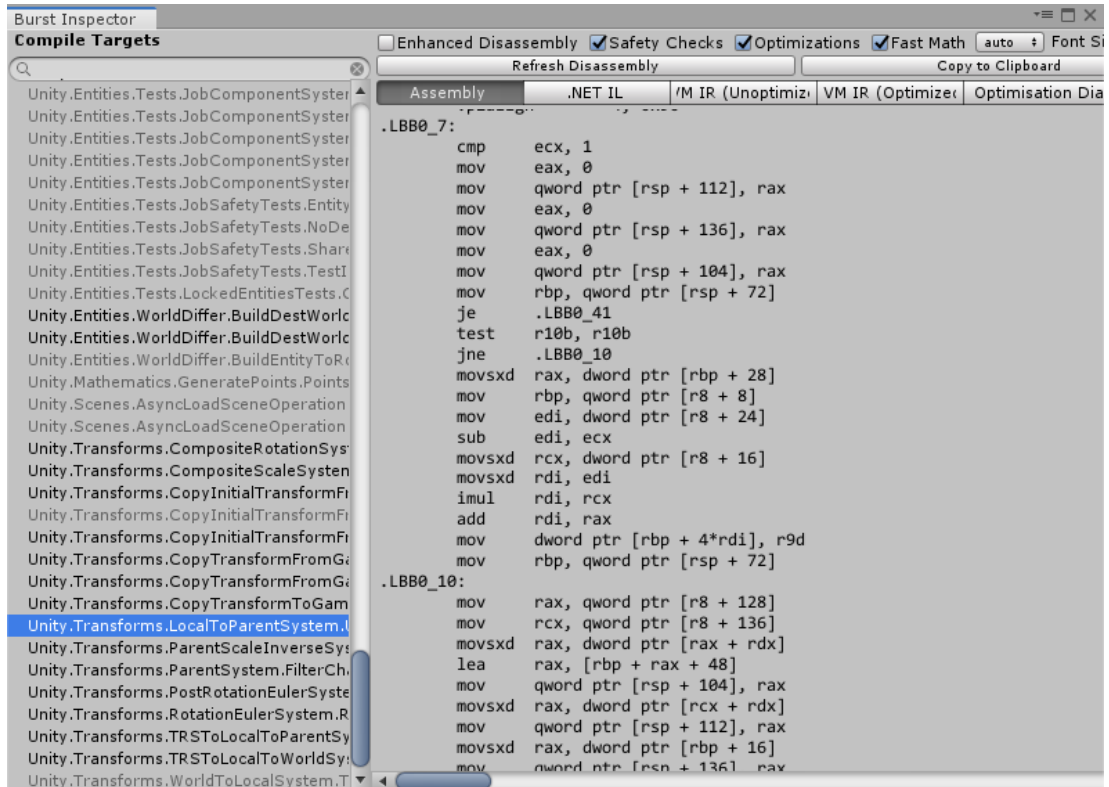


Figure 2.1: Burst Inspector.

resolution is sufficient for usage in game engines. The Win32 API provides the access to the high-resolution timer in the form of `QueryPerformanceCounter()` and `QueryPerformanceFrequency()` functions. The former returns the elapsed number of CPU cycles (or ticks), while the latter returns the number of cycles per second on the particular CPU (Gregory [2018]).

The Unity game engine provides several tools to measure time, while executing the code. Each of them is described in the following subsections.

### 2.4.1 Stopwatch

Firstly, the `Stopwatch` class, in the `System.Diagnostics` namespace of .NET libraries, measures the elapsed time with minimal overhead by using system's high-resolution performance counter. The elapsed time can be accessed from the `Stopwatch` instance by `Elapsed`, `ElapsedTicks` or `ElapsedMilliseconds` properties. The first returns a `TimeSpan` structure, which contains a detailed information about time in hours, minutes, seconds, milliseconds, etc. The second one returns time measured in milliseconds. The last one is measured in timer ticks (Microsoft [2019c]).

Below is a sample usage of the `Stopwatch`:

```
var stopwatch = new Stopwatch();
stopwatch.Start();
Run(); // Function for which we want to measure the time
stopwatch.Stop();
var ticks = stopwatch.ElapsedTicks;
var milliseconds = stopwatch.ElapsedMilliseconds;
```

## 2.4.2 Unity Profiler

Moreover, the Unity Profiler tool contains several modules to measure performance, e.g. CPU Usage Profiler module, Memory Profiler module, GPU Profiler module, UI Profiler module, etc. It can be run in Editor mode during application development or it can be connected to devices connected to the development PC or even network devices during application runtime. The Unity Profiler tool gathers the data for each module and displays them inside a Profiler Window in a user friendly format (Unity Technologies [2019d]).

Figure 2.2 shows the Unity Profiler window. The upper part of the window shows the controls for various profiler options, e.g. enabling the profiling, enabling the deep profiling, clearing the profiler window or loading and saving the profiler data. The left part displays the profiler modules, which can be disabled as needed. The modules can be added by the drop down menu at the top of the this part. On the right of the profiler modules, charts for each modules can be seen after the profiling starts. In the bottom part, the information according to the selected module selected in the left part is displayed. In this specific example, it shows the timeline with the details of the profiling on main thread, render thread, jobs, etc. This can be particularly useful to display profiling information of jobs on worker threads.

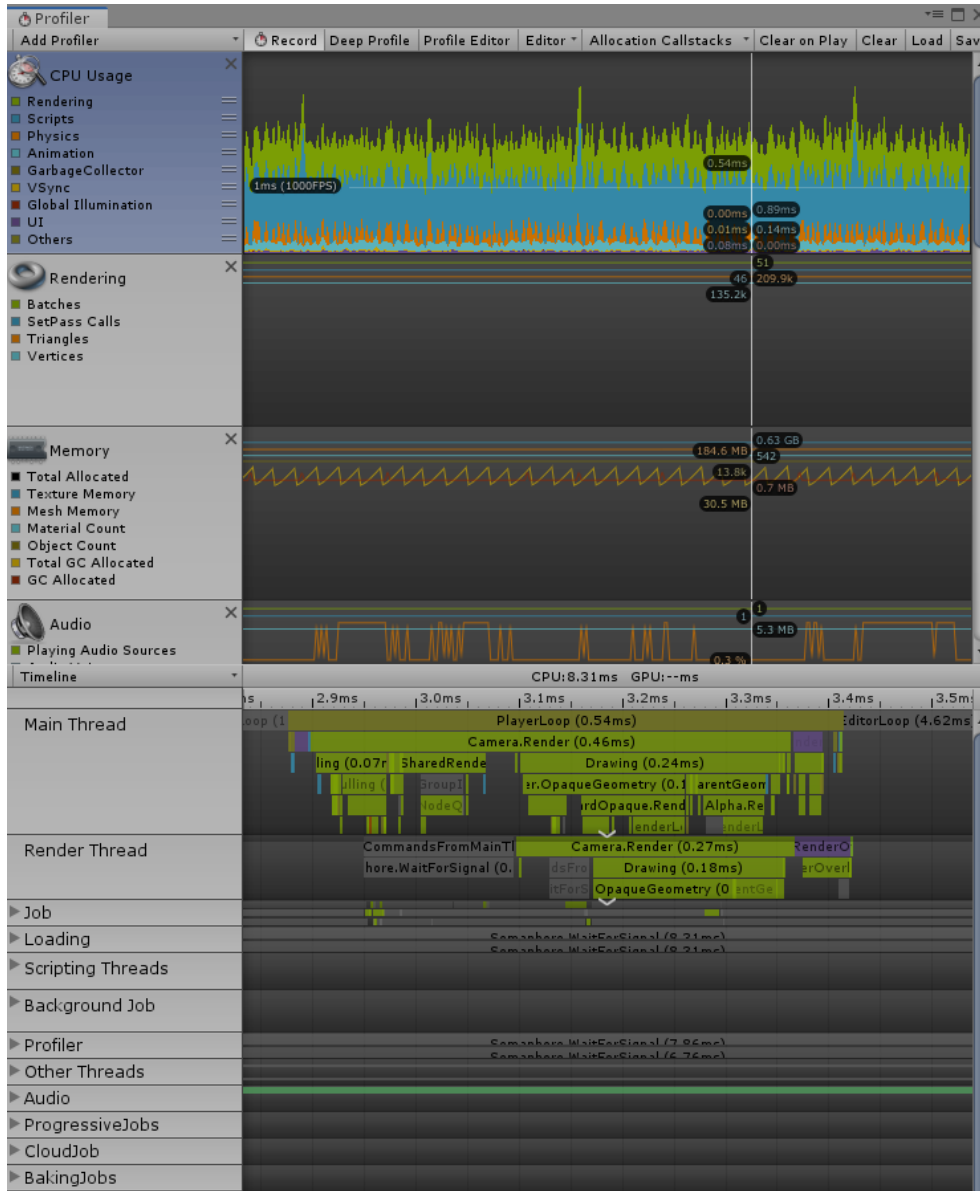


Figure 2.2: Unity Profiler Window.

### 2.4.3 Unity Performance Testing

Inside Unity Editor, users can use the Unity Test Runner framework, which provides users with the possibility to select and run various unit tests and performance tests in edit mode and play mode of Unity Editor and also in standalone builds (Unity Technologies [2019b]).

It is possible to measure time for specific parts of the code in Unity Editor or in standalone build by using a Unity Editor package named the Unity Performance Testing Extension, which can be connected to the Unity Test Runner framework. The Unity Performance Testing Extension comes with API methods and custom method attributes. The custom attributes are following (Unity Technologies [2019a]):

- [Test] - General test used for measuring within a single frame
- [UnityTest] - General yielding test used for measuring in multiple frames
- [Performance] - Sets up the performance test, when used with [Test] or [UnityTest] attribute

In order to take measurement for single methods, it is possible via API method called `Measure.Method()` in the `Unity.PerformanceTesting` namespace, see basic usage below (Unity Technologies [2019a]):

```
public void Fn()
{
    // Contents of measured function
}

[Test]
[Performance]
public void Test()
{
    var measure = Measure.Method(Fn);
    measure.Run();
}
```

When a performance test is executed the summary of the test is displayed in the Unity Editor. The summary consist of statistical information, e.g. min, max, average, median, etc. Furthermore, more detailed summary can be found in Performance Test Report window (Unity Technologies [2019a]).

Figure 2.3 depicts the Unity Test Runner window. In the top part, users can select the mode, for which the tests should be displayed - edit mode or play mode. It also contains buttons to run selected tests or all tests. The middle part shows all the runnable tests in the currently selected mode. Finally, the bottom part displays the summary of the selected test measurement.

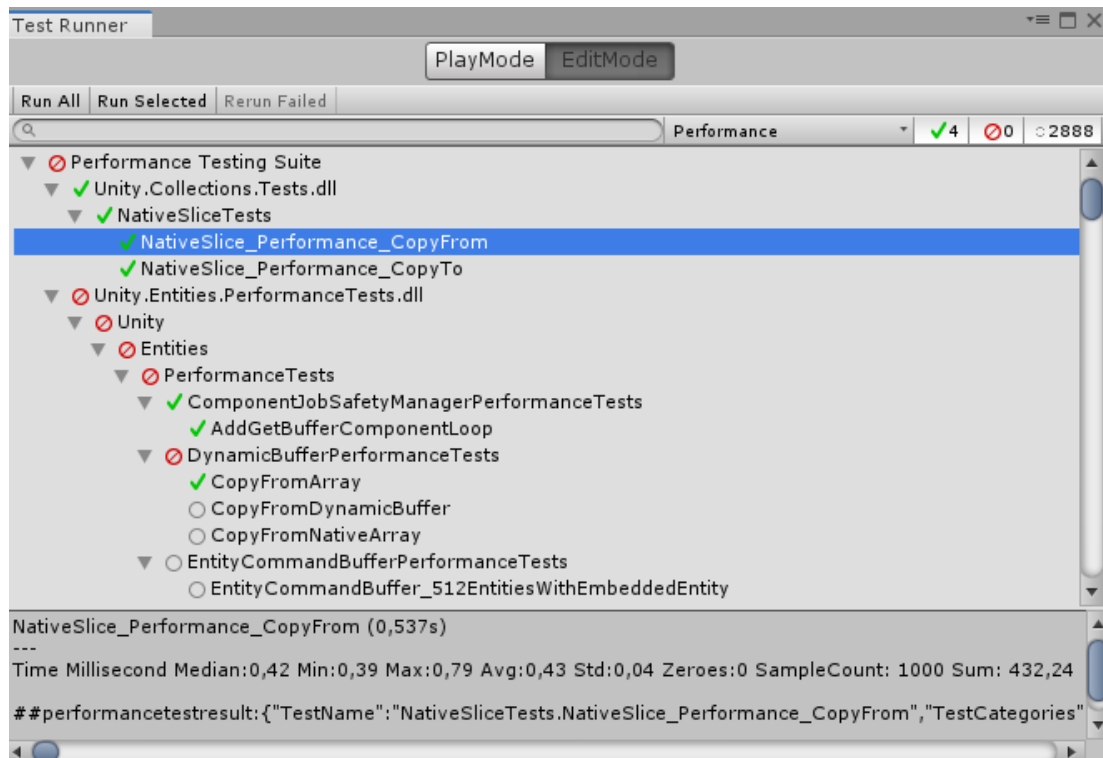


Figure 2.3: Unity Test Runner.

#### 2.4.4 Unity Editor vs Standalone Build

Undoubtedly, it is easier to run all kinds of performance tests and benchmarks inside Unity Editor without having to build the standalone executable after every change in code. However, for performance reasons it is recommended to use Standalone Build over Unity Editor.

The reasoning behind that is Unity Editor can introduce certain overhead for some operations. In the Unity DOTS system, the overhead might be caused by Native arrays checks for race conditions, Burst safety checks, jobs debugger or leak detection<sup>3</sup>.

<sup>3</sup><https://forum.unity.com/threads/job-system-not-as-fast-as-mine-why-not.518374/#post-3397587> (Accessed December 15th, 2019)

## 3. Related Work

There is a limited amount available work and research done in the field of the Unity DOTS. The technology was released relatively recently, some features of the system are still in preview, the source code behind the Unity C# Job System and Burst compiler is not available to the public and the the system is not fully documented. Therefore, there are only a few works testing the performance and efficiency of the Unity DOTS framework as a whole. In this chapter, the list of the most relevant works done in this field will be compiled.

### 3.1 Unity DOTS Benchmark

The first example is a benchmark described and performed by Ferreira and Geig [2018]. This benchmark compares several approaches to moving and rendering large number of ships, namely:

- Classic approach with a MonoBehaviour component per each Game Object.
- Classic approach combined with C# Job System
- ECS approach combined with C# Job System
- ECS approach combined with C# Job System and Burst compiler

The unit of measurement is the amount of ships, which can be instantiated to the scene to achieve frame time of approximately 33ms. The benchmark was run on Intel Core i7-8700K processor (4.7GHz) and NVIDIA GeForce GTX 1080 graphics card.

The results have shown that performance of ECS with C# Job System and Burst is far more superior than the traditional approach (more than 9x objects are instantiated in this case). Using jobs in the classic MonoBehaviour approach results in approximately 1.7x more objects. The ship count in the case of ECS is more than 3x bigger than the non-ECS version with the C# Job System. The Burst itself gives another boost against the non-Burst alternative - almost 1.65x more ships. For the results of the benchmark by Ferreira and Geig [2018], see table 3.1.

	Classic	Classic + Jobs	ECS + Jobs	ECS + Jobs + Burst
Ship count	16,500	28,000	91,000	150,000+

Table 3.1: Unity DOTS Benchmark Results

## 3.2 Analyzing Burst Generated Assemblies

The next piece of related work is an analysis by Pitaksarit, S. [2019], which tested and examined the code generated by Burst compiler. The analysis was based on comparing the pieces of C# code with similar functionality and assessing the level of optimizations done in each assembly code generated by Burst, which is accessible via Burst Inspector (see subsection 2.3.3). The author experimented with the sizes of job structs, the arrangement of the member variables and their manual alignment. Furthermore, the Burst assemblies of several math functions inside Unity.Mathematics namespace were compared against their naive implementation counterparts.

Results have shown that Burst was able to effectively vectorize a job performing an assignment of struct, which contains float4 type (an array of four floats), by using the xmm register. However, for the assignment of the struct with float3 type, Burst did not generate any SIMD instructions. Nevertheless, the vectorization was enabled when explicit padding of 4 bytes was used, see below:

```
[ StructLayout ( LayoutKind . Explicit ) ]  
public struct PaddedStruct : IComponentData  
{  
    [ FieldOffset ( 4 ) ]  
    public float3 x;  
}
```

Moreover, the author demonstrated that the structure of four floats and the structure of one float4 results into the exactly same assembly. Furthermore, it has been shown that simple C# properties accessing member variables do not cause any additional assembly code generated. Additionally, the comparison of the author's inlined implementation of dot product computation with math.dot method from Unity.Mathematics namespace, was in favor of the latter. Therefore, it was suggested to use the math functions from Unity.Mathematics, where it is possible.

## 3.3 Burst Benchmarks

Another interesting work is a benchmark, which compares the performance of 11 famous algorithms, e.g. Sieve of Eratosthenes, Mandelbrot or RayTracer depending on the compilers. The benchmark is accessible on GitHub<sup>1</sup>. This source compares Burst compiler against other scripting backends in Unity - IL2CPP<sup>2</sup> (Intermediate Language To C++) and MonoJIT<sup>3</sup>. Furthermore, it compares the compilers for the C# code against C compilers, namely GCC<sup>4</sup> and Clang<sup>5</sup>.

---

<sup>1</sup><https://github.com/nxrighthere/BurstBenchmarks> (Accessed November 18th, 2019)

<sup>2</sup><https://docs.unity3d.com/Manual/IL2CPP.html> (Accessed November 18th, 2019)

<sup>3</sup><https://www.mono-project.com/docs/about-mono/> (Accessed November 18th, 2019)

<sup>4</sup><https://gcc.gnu.org/> (Accessed November 18th, 2019)

<sup>5</sup><https://clang.llvm.org/> (Accessed November 18th, 2019)

The performance was measured in CPU ticks, see section 2.4. The author of the benchmark states that it was executed on a standalone build on AMD FX-4300 processor (4GHz).

The results of the benchmark have shown that Burst performance is superior when it is compared with its C# counterparts, in majority cases. Regarding the performance comparison of Burst and C compilers, results show that Burst competes well with Clang and GCC for the majority of the algorithms.

## 3.4 Native Memory Allocators

Another work<sup>6</sup> compares the native memory allocators described in subsection 2.3.2. The author focuses on comparison of the allocation and deallocation times of the allocators and also observes other allocator properties, e.g. memory distance of different allocations using the same allocator.

It has been shown, that the `Allocator.Temp` dominates the `Allocator.TempJob`, and `Allocator.TempJob` outperforms `Allocator.Persistent` in case of allocation and deallocation times of small native containers, e.g. allocation of 4 bytes. The same is valid for the memory distance of different allocations. `Allocator.Temp` shows very cache-friendly behaviour for subsequent small allocations, whereas the results for `Allocator.Persistent` show very distant memory addresses for subsequent allocations.

## 3.5 Conclusion

The aforementioned examples demonstrate the performance aspects of some features of the Unity DOTS system. This work will focus on creation of its own performance testing suite for running benchmark tests for Unity DOTS, but also for all kinds of user tests in general. Furthermore, the goal is to create a set of tests for the suite to benchmark the performance of various features of the Unity DOTS system.

More specifically, the tests will focus on assessing which C# features can be optimized by Burst and their performance gains. Moreover, the comparison of single-threaded jobs and parallel jobs divided into worker threads with or without Burst will be studied. Furthermore, we will examine the performance of mathematical functions within `Unity.Mathematics` namespace, which are meant to be used with Burst. Based on the evaluation of the results a list of recommendations, which should be taken into account when writing high-performance code within Unity DOTS, will be generated. The recommendations will be then evaluated in a real-time application scenario - boids simulation.

---

<sup>6</sup><https://jacksondunstan.com/articles/5406> (Accessed December 15th, 2019)

## 4. Implementation

Although Unity provides the users with its own Performance Testing Suite (see subsection 2.4.3), it lacks several features, which could serve us. Firstly, it does not come with a unified interface for initialization and cleanup of the data for the tests. Moreover, it might be useful to run the tests in standalone builds on various platforms. Furthermore, we would like to be able to export the results to a simple comma-separated value (CSV) text file in order to be able to process the results in any 3rd party statistical software. Additionally, the tests can be divided into groups to compare the results with the other tests of the group. As a result, our own Performance Testing Suite will be implemented in order to overcome all the aforementioned shortcomings.

This chapter describes the implementation of the Performance Testing Suite in Unity. The structure of the tests is described, the high-level format of the application is covered, the implementation of the test framework is described, the measurement procedure is outlined and the option of extensibility of the suite by writing own tests is shown.

### 4.1 Tests and Test Sets

In order to be able to write various kinds of tests, we need to introduce a unified way of writing benchmark tests. Let's say a test is a calculation we want to measure. It is comprised by initialization of its values, actual execution of the test, which is measured and finally the cleanup of the values. Moreover, a test set is a group of similar tests with minor differences. The benchmarks of the tests in a test set are supposed to be compared against each other.

The test set also defines the default values of test sizes, for which the tests should be executed. Furthermore, the test set contains the information, how many times the measurement should be repeated by default.

### 4.2 Format of the Performance Testing Suite

There are two possibilities to launch the Performance Testing Suite and run the test sets. First of them is by using a PTS plugin inside Unity Editor, see appendix A.1.2. The second option is to run the standalone build of the Performance Testing Suite, which provides the user with the opportunity to run the benchmark on several devices, as the executable can be built for multiple platforms in Unity. For the purpose of this thesis, the Windows and Android standalone builds are provided, see appendix A.2 and A.3. For differences in measuring performance in Editor vs standalone application refer to section 2.4.4.

When using Unity Editor, the user can navigate to the tab PTS, where the option Test Set Runner can be selected, see figure 4.1. This opens the Test Set Runner tab, where the all the available test sets are displayed and user can select which test sets he wants to execute. Furthermore, the user can select the output directory, where he wants to save the text files with the benchmark results. At the bottom of the window, there is a Run button, which triggers the benchmark upon clicking. The Test Set Runner tab is displayed in figure 4.2.

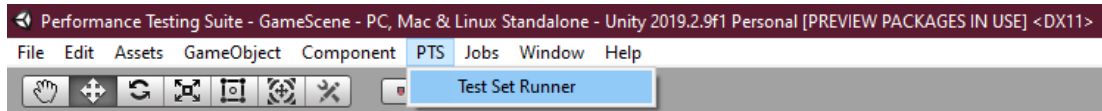


Figure 4.1: Unity Editor Menu.

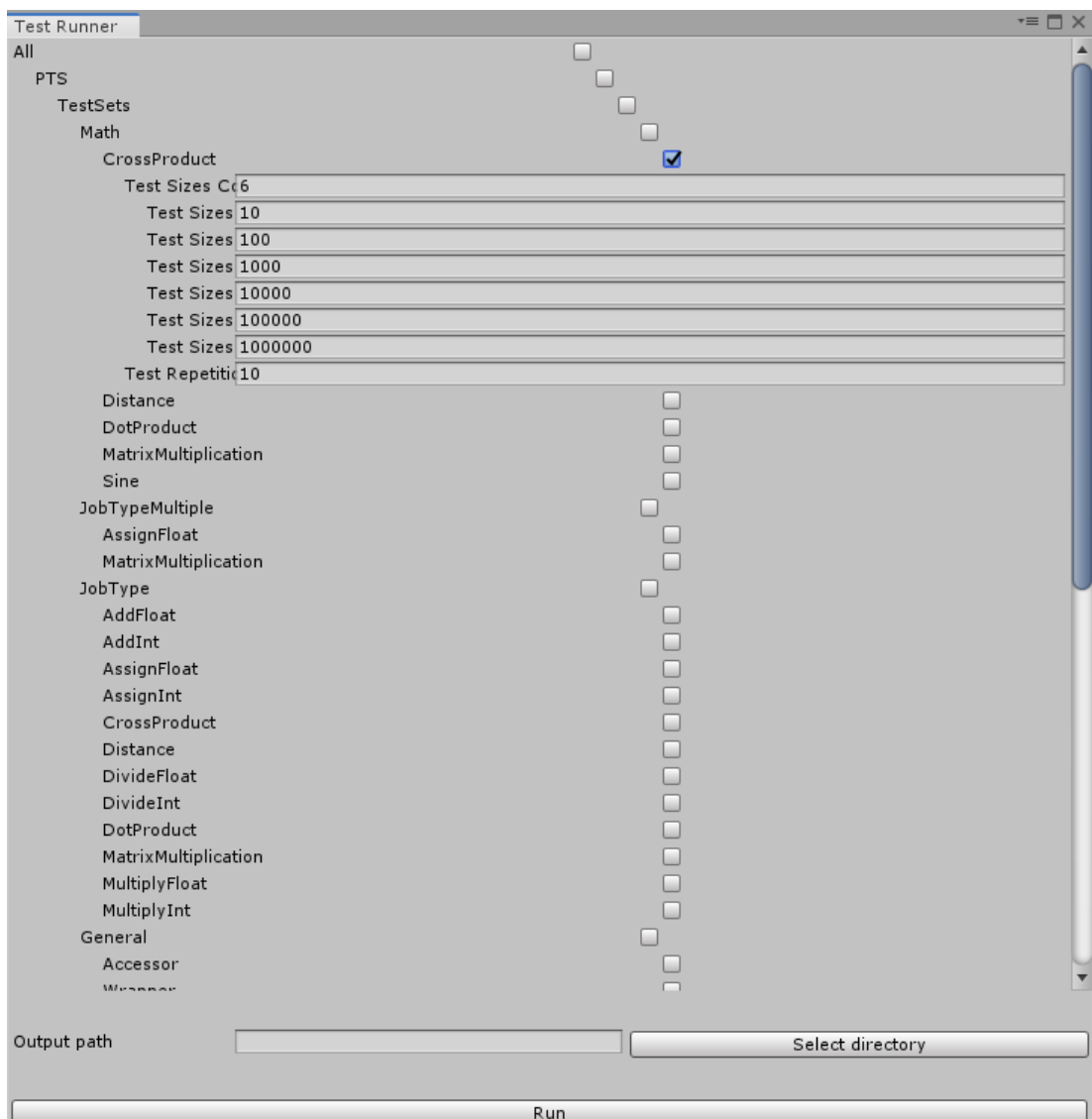


Figure 4.2: Unity Editor Test Set Runner.

When using the standalone version, upon the startup of the application, user is presented with a window. Similarly to the Unity Editor version, test sets are displayed inside a window and they can be selected for the benchmark, see figure

4.3. After clicking the Run button at the bottom of the window, the selected test set benchmarks are executed and the results are saved into a persistent data path directory of the application, which is defined in the Unity documentation<sup>1</sup>.

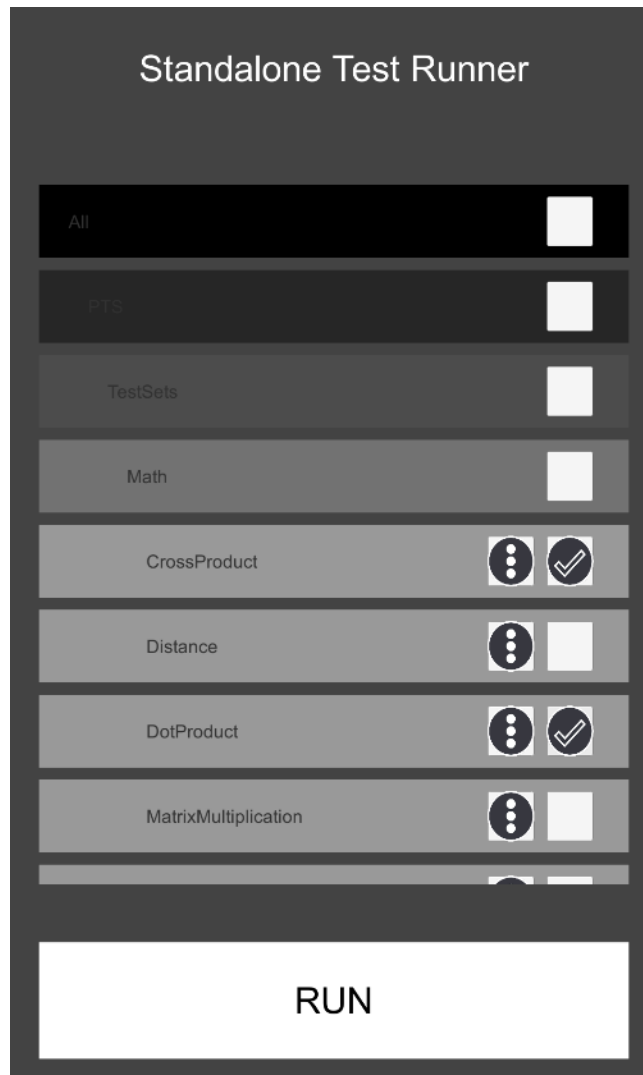


Figure 4.3: Standalone Test Set Runner.

## 4.3 Test Framework

The test framework consists of the interface for the tests and test sets. Moreover, it comprises the measured tests and test sets itself. Furthermore, it contains the test runner, which is responsible for running the tests and outputting the results. The last part of the framework is the user interface wrapper for selecting and running the tests in Unity Editor or standalone build.

<sup>1</sup><https://docs.unity3d.com/ScriptReference/Application-persistentDataPath.html>  
(Accessed November 2nd, 2019)

### 4.3.1 Tests and Test Sets

The representation of tests and test sets in C# consists of two interfaces `ITest` (which represents a test) and `ITestSet` (which stands for a test set) defined in the namespace `PTS`. The required functionalities of `ITest`, which a class or struct must implement are shown below:

```
public interface ITest
{
    // Initialization of the values for given test size
    void Init(int size);
    // Measured execution of the test
    void Run();
    // Cleanup
    void Terminate();
}
```

The mandatory interface members of `ITestSet` can be seen below:

```
public interface ITestSet
{
    IReadOnlyList<ITest> Tests { get; }
    List<int> TestSizes { get; set; }
    int TestRepetitions { get; set; }
}
```

### 4.3.2 Test Set Runner

The `TestSetRunner` is the class responsible for running and measuring the test sets. It initializes sequentially all the tests provided, measures the execution time of their `Run()` function and handles the proper cleanup.

There are two wrappers, which provide the user interface for the test runner in Unity Editor and inside standalone build. Namely, classes `EditorTestSetRunner` and `StandaloneTestSetRunner`. Both wrappers are capable of running the `TestSetRunner` with the list of the test sets selected by the user.

## 4.4 Measurement Procedure

In order to reduce variations in the elapsed ticks measured, which might be caused by garbage collection or by any means unrelated to the running application, each test is measured the number of times defined in the test set, and the significant statistics for the test runs are collected. These collected statistics are mean, min, max, median, 25th percentile and 75th percentile and they are available to be further processed by a 3rd party statistical software. Before and after each test

iteration, the garbage collection is invoked. Another technique, which contributes to increased precision of the benchmark is a warm-up run of the test with the size parameter of 1, which is performed before each test is executed. This allows for the JIT compilation of the Burst job, which takes several milliseconds in general. After that, the measured tests do not require any additional overhead by Burst.

## 4.5 Extensibility

The Performance Testing Suite was written with the possibility of extensibility in mind and therefore it is possible for users to write and benchmark their own tests. Custom tests can be written for Burst jobs, but also for any code in general.

In order to write custom tests, user needs to create a class implementing the `ITest` interface and put the calculations he wants to measure into the body of the `Run` method. The initialization of values (which depend on the size of the test) should be inserted into the body of the `Init` method as it is called before the actual benchmark. The cleanup of the values should be placed into the `Terminate` method. Furthermore, the user is required to create another class implementing the `ITestSet` interface, which stores a list of tests and the recommended values of the size parameters, e.g. to benchmark only a single test, it suffices to create one test set, which contains the test.

# 5. Results

In section 3.5, we roughly described the tests we were implementing for the performance testing suite. In this chapter, the results of the measured tests will be presented and analyzed. Furthermore, for some test sets, we will take into account and compare the Burst generated assemblies for its tests, which are accessible through the Burst Inspector, see subsection 2.3.3. Given the results of the tests, we will compile a list of recommendations for writing high-performance code in Unity DOTS. The results will be then evaluated in a real-time simulation of boids.

All the tests were executed within Standalone build on 2 desktop devices and 1 mobile device:

- Desktop A
  - Windows
  - CPU - Intel Core i5-7400, 4-core 3.5 GHz, SSE4.1, SSE4.2, AVX2
  - RAM - 8GB
  - GPU - GeForce GTX 1050Ti
- Desktop B
  - Windows
  - CPU - Intel Core i5-4210U, 2-core 2.7 GHz, SSE4.1, SSE4.2, AVX2
  - RAM - 8GB
  - GPU - Intel HD Graphics 4400
- Android
  - CPU - 4-core Samsung Exynos M1 2.6GHz, 4-core ARM Cortex-A53 1.6GHz, Advanced SIMD (NEON) - 128-bit registers
  - RAM - 4GB
  - GPU - Mali-T880

The tables and figures with the results for the Desktop A platform will be presented in this chapter. For more information about the results for the platforms Desktop B and Android, refer to Appendix B.

Our tests are measured on test sizes, which are multiplies of 10 and therefore it allows for easier interpretation of the results from the figures, since the scales of the x and y axes in the figures are 10-logarithmic.

The entire list of the measured tests is following:

- **Wrapper Test Set** - examines how Burst handles nested structures and structures implementing an interface as opposed to plain structures and built-in C# value types
- **Accessor Test Set** - compares the behaviour of Burst when accessing member variables directly, through properties or by using methods
- **Allocator Test Set** - measures the performance of the allocators, described in section 3.4, in conjunction with Native Containers
- **Batch Size Test Sets** - determines of the optimal value of the batch size parameter for parallel jobs in general
- **Job Type Test Sets** - compares scenarios, where it is appropriate to prefer parallel jobs over single-threaded jobs and Burst vs non-Burst versions
- **Float Optimization Test Sets** - examines the different floating point number optimizations made by Burst
- **Math Test Sets** - the target is to compare the performance of the mathematical functions from the traditional Unity implementation and the new implementation from Unity.Mathematics namespace

## 5.1 Wrapper Test Set

The first test set, named Wrapper Test Set, contains tests, which are running operations of assignment of values from one Native Array to another one, on different data structures. Each data structure is different, however, its size is exactly four bytes. Let's describe them below:

- Primitive Type - int
- Struct - struct, which contains a single int member
- Nested Struct - struct, which contains another nested struct with a single int member
- IComponentData - struct, which implements IComponentData interface

Table 5.1 presents the median elapsed ticks of each test in the test set based on the test size. Similarly, figure 5.1 displays this relationship in a line chart.

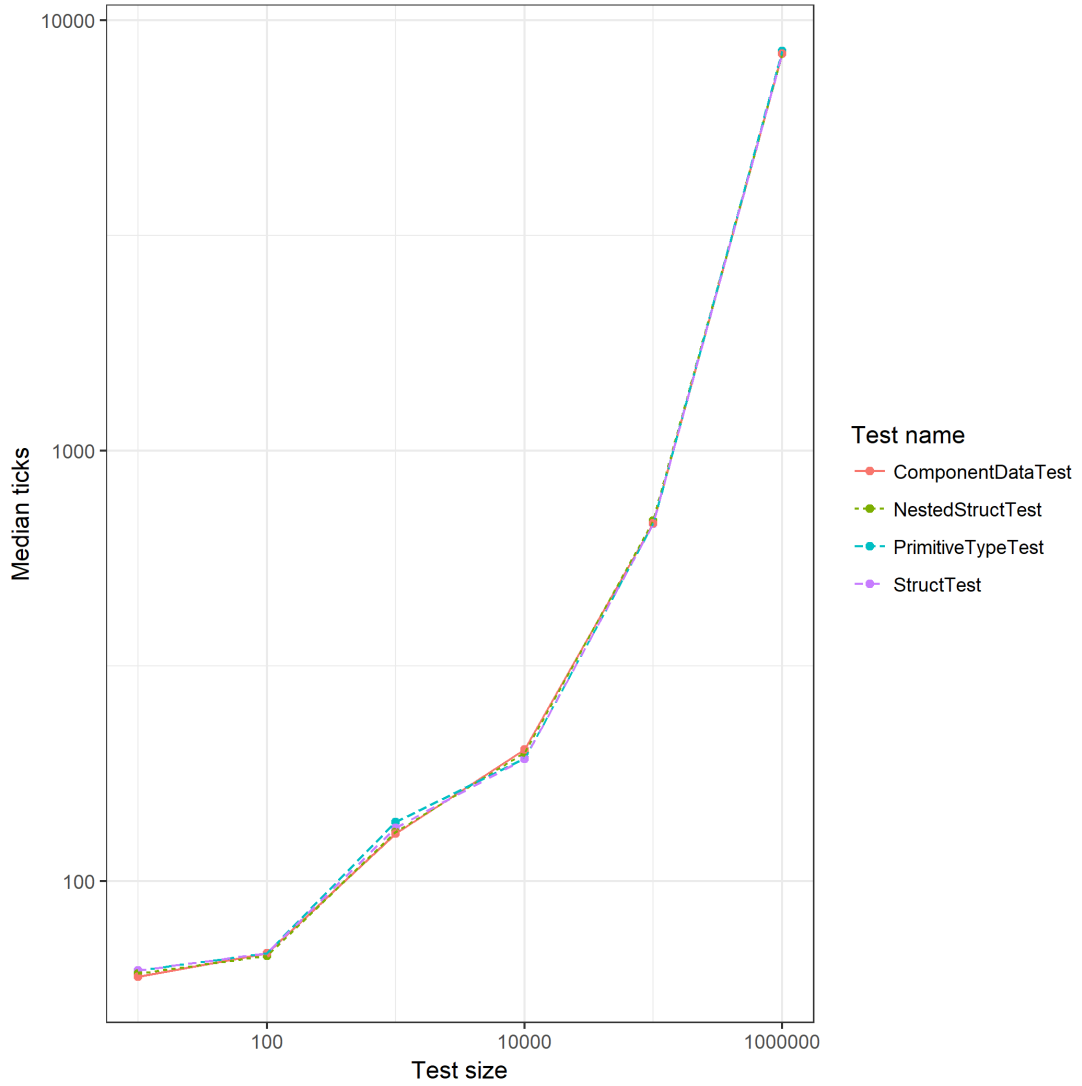


Figure 5.1: Wrapper Test Set.

Test Size	Primitive Type	Struct	Nested Struct	IComponentData
10	62	62	61	60
100	68	68	67	68
1 000	137	133	130	129
10 000	193	192	199	202
100 000	676	676	688	678
1 000 000	8483	8362	8367	8372

Table 5.1: Wrapper Test Set

The results of this test set show that wrapping value types inside another structure causes no additional overhead, as Burst generates identical Assembly code for the assignment of the structure as for the assignment of the nested type. Furthermore, wrapping the aforementioned structure into another structure or using a structure, which implements an interface does not require any additional CPU time and the generated Burst Assembly is equivalent to previous assemblies.

Since, the Burst generated assemblies are identical for all types of tests, the tiny differences in median elapsed ticks can be attributed to different ratio of cache hits and cache misses during each test and to the computation of median.

As a result, using plain structures instead of nested structures or structures implementing the IComponentData interface in the jobs gives the user no additional performance benefit and therefore it is appropriate to use the latter, when running calculations within the ECS.

## 5.2 Accessor Test Set

Another test set, called Accessor Test Set, compares the assignment of structures with one float from one Native Array to another. However, the assignment is done differently for each test. The assignment approaches are following:

- Copy Test - copy of the struct
- Field Test - copy of the struct by copying the member directly
- Property Test - copy of the struct by copying the member through property
- Auto-Property Test - copy of the struct by copying the member through auto-implemented property
- Method Test - copy of the struct by copying the member through getter and setter methods
- Inline Method Test - copy of the struct by methods with aggressive inlining attribute

Table 5.2, again, shows the median elapsed ticks of each test based on the test size. Futhermore, figure 5.2 displays this relationship in a line chart.

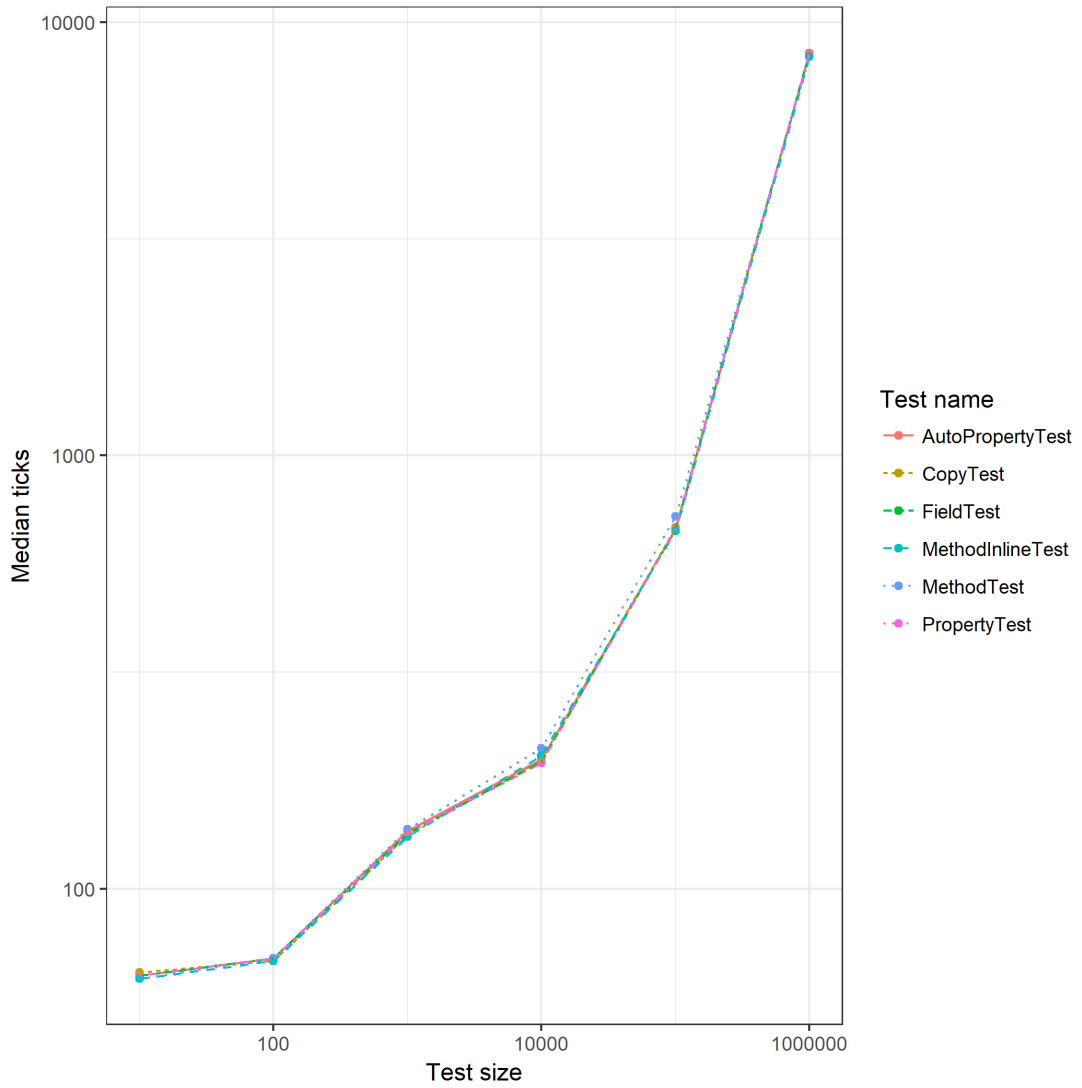


Figure 5.2: Accessor Test Set.

Test Size	Copy Test	Field Test	Property Test
10	64	63	63
100	68	69	69
1 000	133	134	134
10 000	198	196	195
100 000	682	675	675
1 000 000	8495	8484	8487
Test Size	Auto-Property Test	Method Test	Inline Method Test
10	63	62	62
100	69	69	68
1 000	136	137	132
10 000	199	211	203
100 000	679	724	670
1 000 000	8478	8341	8321

Table 5.2: Accessor Test Set

The results, show that using direct user-implemented or auto-implemented properties has no additional overhead over direct manipulation with the fields. This confirmed the findings of work in section 3.2. Furthermore, the same is valid for using getter and setter methods instead of a direct member access, as it seems to be inlined in all cases, due to the very small size of the method. Moreover, the generated Burst Assembly is identical in each test of the test set.

Again, since all types of tests generate identical Burst assembly, the negligible differences in median elapsed ticks are affected by cache hits, cache misses and computation of median.

As a result, it is appropriate to abstract from using direct member access by using properties (user-implemented or auto-implemented) or methods as long as its only purpose is to access the member.

### 5.3 Allocator Test Set

The following test set is the Allocator Test Set, which compares the performance of allocation of NativeCollection, based on the type of allocator. All three allocators (Allocator.Temp, Allocator.TempJob and Allocator.Persistent), as described in subsection 2.3.2, are tested.

In table 5.3 the median elapsed ticks of each test based on the test size is depicted. Additionally, the relationship is displayed in a line chart in figure 5.3.

Test Size	Temp Allocator	Temp Job Allocator	Persistent Allocator
10	11	13	13
100	12	17	16
1 000	14	20	17
10 000	15	19	17
100 000	15	18	18
1 000 000	222	223	220

Table 5.3: Allocator Test Set.

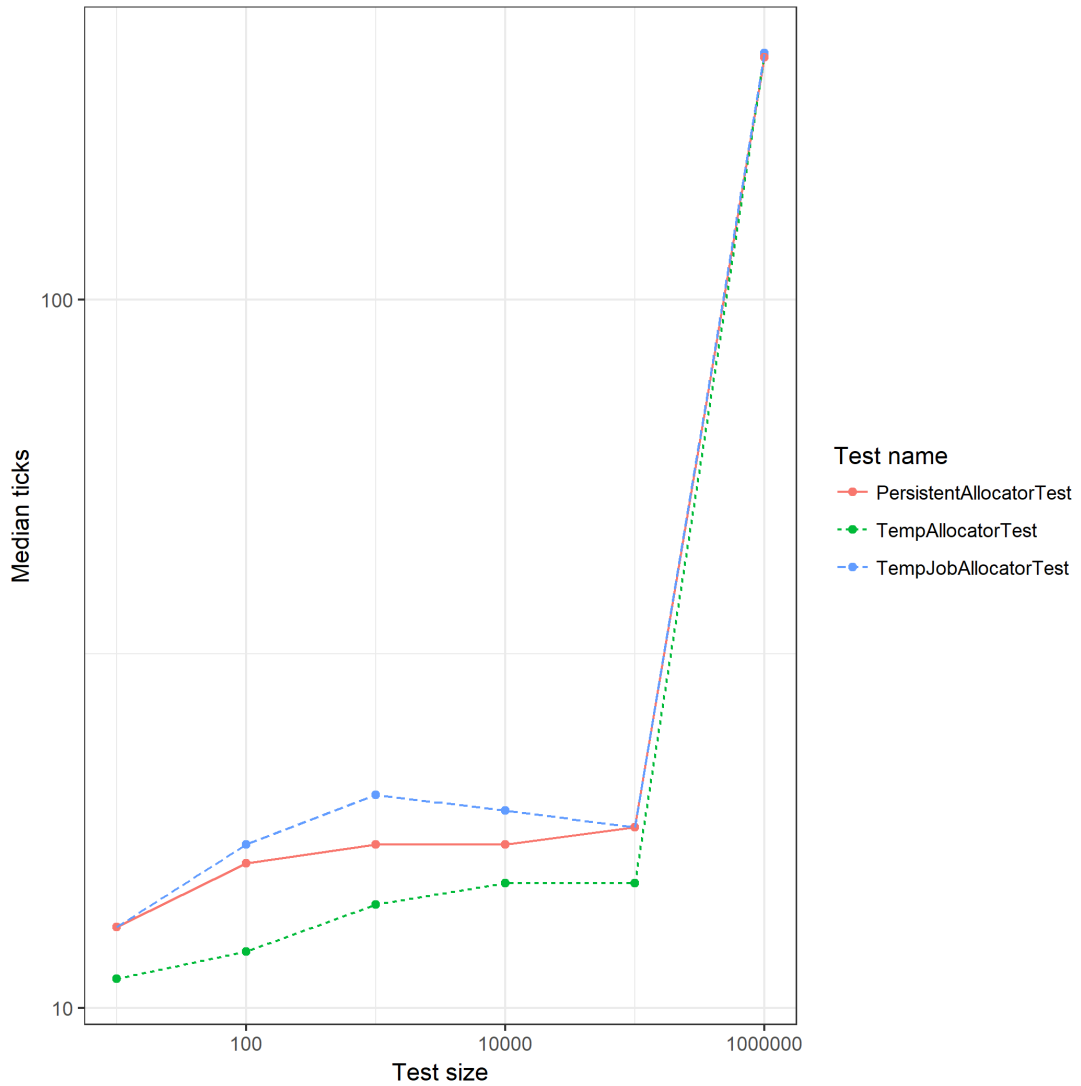


Figure 5.3: Allocator Test Set.

According to the results, for `NativeArray` allocations up to 100 000 floats, the `Allocator.Temp` is the fastest, `Allocator.Persistent` is the second fastest and `Allocator.TempJob` is the slowest of all three. This, however, does not confirm our expectations based on information in subsection 2.3.2 and the work from 3.4. Asymptotically, all the allocators have comparable performance.

Consequently, in general, it is recommended to use `Allocator.Temp` whenever possible. However, since it is not possible to use `Allocator.Temp` with jobs, it is not perfectly clear, whether `Allocator.TempJob` should be preferred over `Allocator.Persistent`, because of the discrepancy between our findings and findings in 3.4, especially when `Allocator.Persistent` can be reused in consecutive executions of the same operation.

## 5.4 Batch Size Test Sets

The next test sets, named Batch Size Test Sets, are comprised by tests measuring the time of parallel calculation using `IJobParallelFor`. The tests differ in the value of the batch size parameter (numbers from 1 to 256, which are the power of four) provided to the `Schedule` method of the job, as described in subsection 2.3.2. Based on the value of this parameter, Unity divides the work into worker threads.

### 5.4.1 Float Assignment

This subsection compares the tests of float assignment from one Native Array to another based on the batch size parameter.

Firstly, only the parallel version with `IJobParallelFor` without Burst is tested. The median elapsed ticks of each test based on the test size are presented in table 5.4, while figure 5.4 depicts the relationship in a line chart.

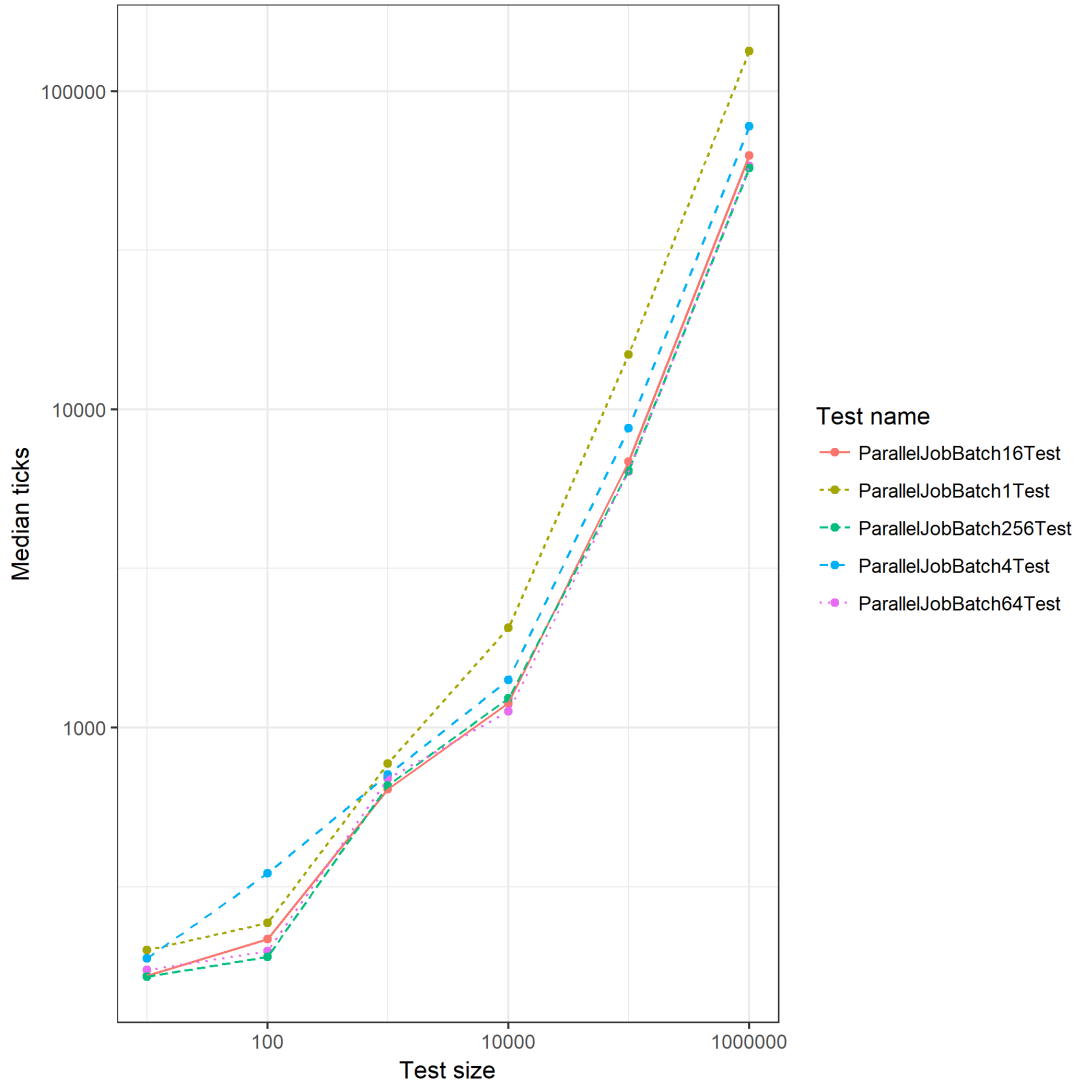


Figure 5.4: Batch Size Test Set - Assign Float - Parallel.

Test Size	Batch of 1	Batch of 4	Batch of 16	Batch of 64	Batch of 256
10	200	188	166	173	165
100	243	348	216	198	190
1 000	769	712	640	690	658
10 000	2059	1409	1196	1123	1236
100 000	14851	8736	6835	6382	6410
1 000 000	133643	77508	62666	58058	57216

Table 5.4: Batch Size Test Set - Assign Float - Parallel

Moreover, the parallel version using Burst is tested. Again, the median elapsed ticks can be seen in table 5.5 and the respective graph in figure 5.5.

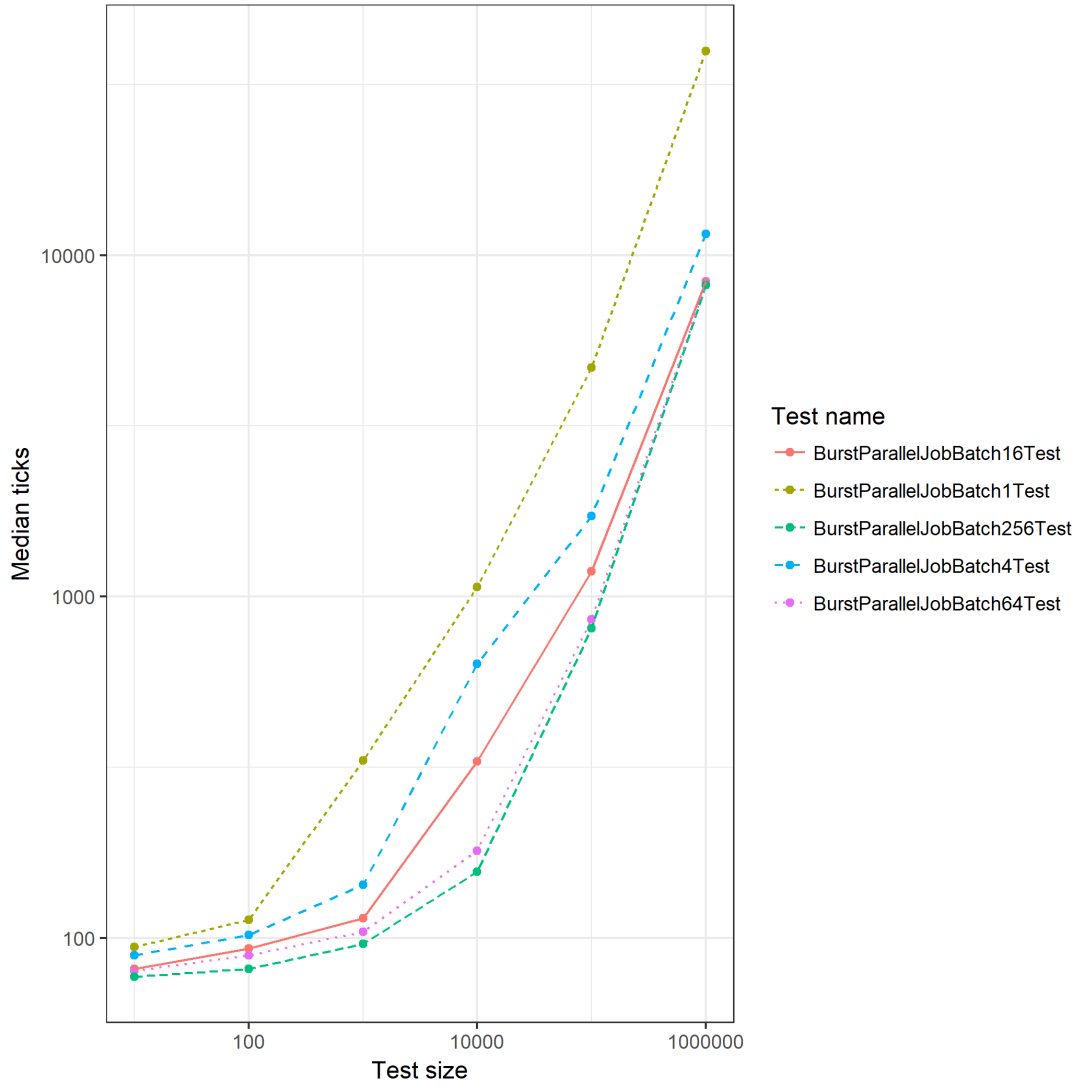


Figure 5.5: Batch Size Test Set - Assign Float - Burst Parallel.

Test Size	Batch of 1	Batch of 4	Batch of 16	Batch of 64	Batch of 256
10	94	89	81	80	77
100	113	102	93	89	81
1 000	331	143	114	104	96
10 000	1067	635	328	180	156
100 000	4679	1723	1185	855	806
1 000 000	39606	11521	8367	8279	8180

Table 5.5: Batch Size Test Set - Assign Float - Burst Parallel

The results of this test set have shown that it is efficient to set batch size parameter to a higher value, e.g. 64 or 256, when each iteration of the job is running only a small number of operations.

## 5.4.2 Matrix Multiplication

On contrary, if the underlying calculation of the Batch Size Test is a multiplication of two 4x4 matrices instead of a mere float assignment, the differences in median elapsed ticks are less noticeable for different values of the batch size parameter.

Parallel matrix multiplication median elapsed ticks without Burst are shown in table 5.6 and the respective graph in figure 5.6.

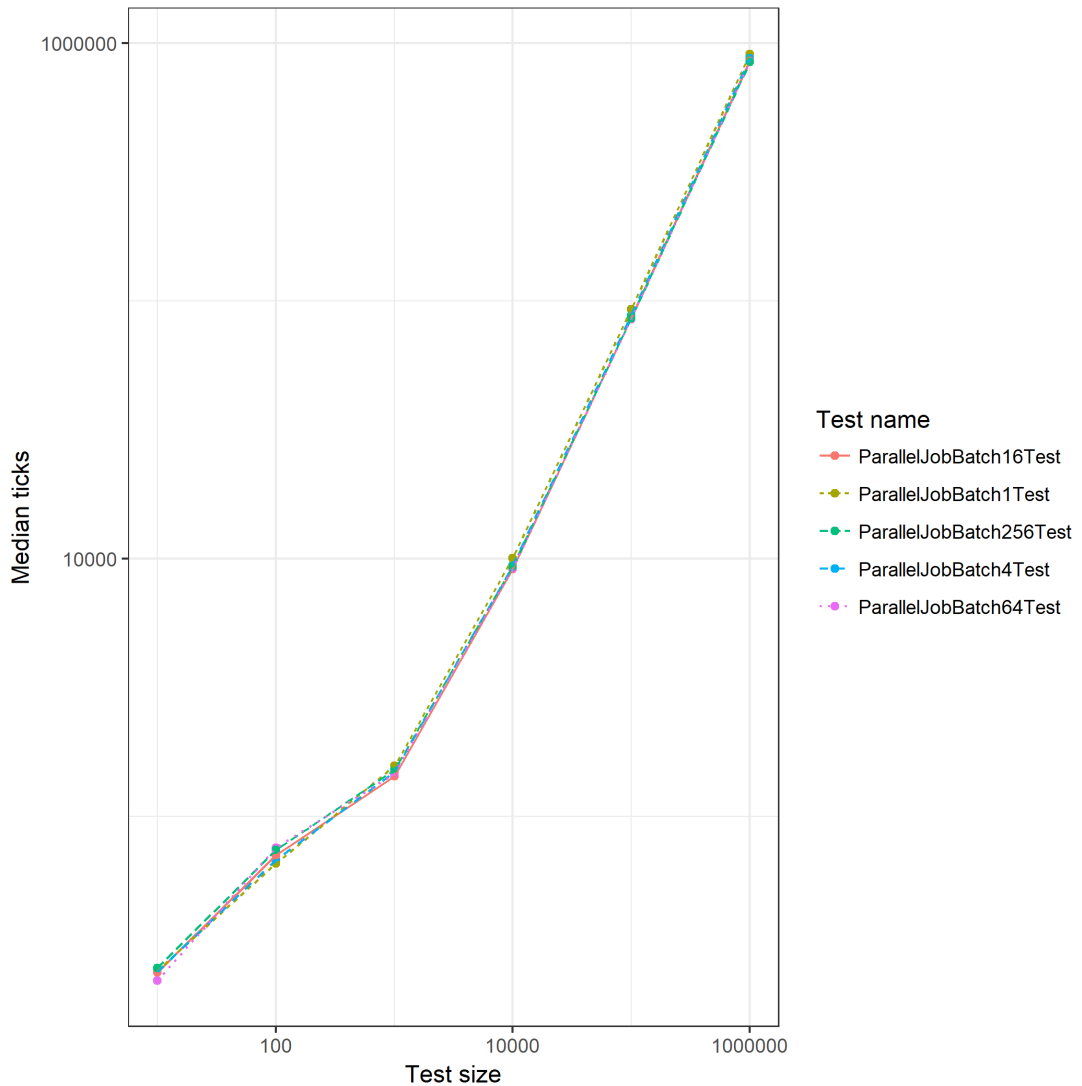


Figure 5.6: Batch Size Test Set - Matrix Multiplication - Parallel.

Test Size	Batch of 1	Batch of 4	Batch of 16	Batch of 64	Batch of 256
10	252	250	249	231	258
100	658	680	706	754	741
1 000	1571	1490	1434	1468	1518
10 000	10059	9399	9125	9148	9334
100 000	92882	87726	85588	84990	85580
1 000 000	906616	873082	851504	847954	843941

Table 5.6: Batch Size Test Set - Matrix Multiplication - Parallel

The median elapsed ticks for the matrix multiplication in burst parallel mode can be seen in table 5.7 and the respective graph in figure 5.7.

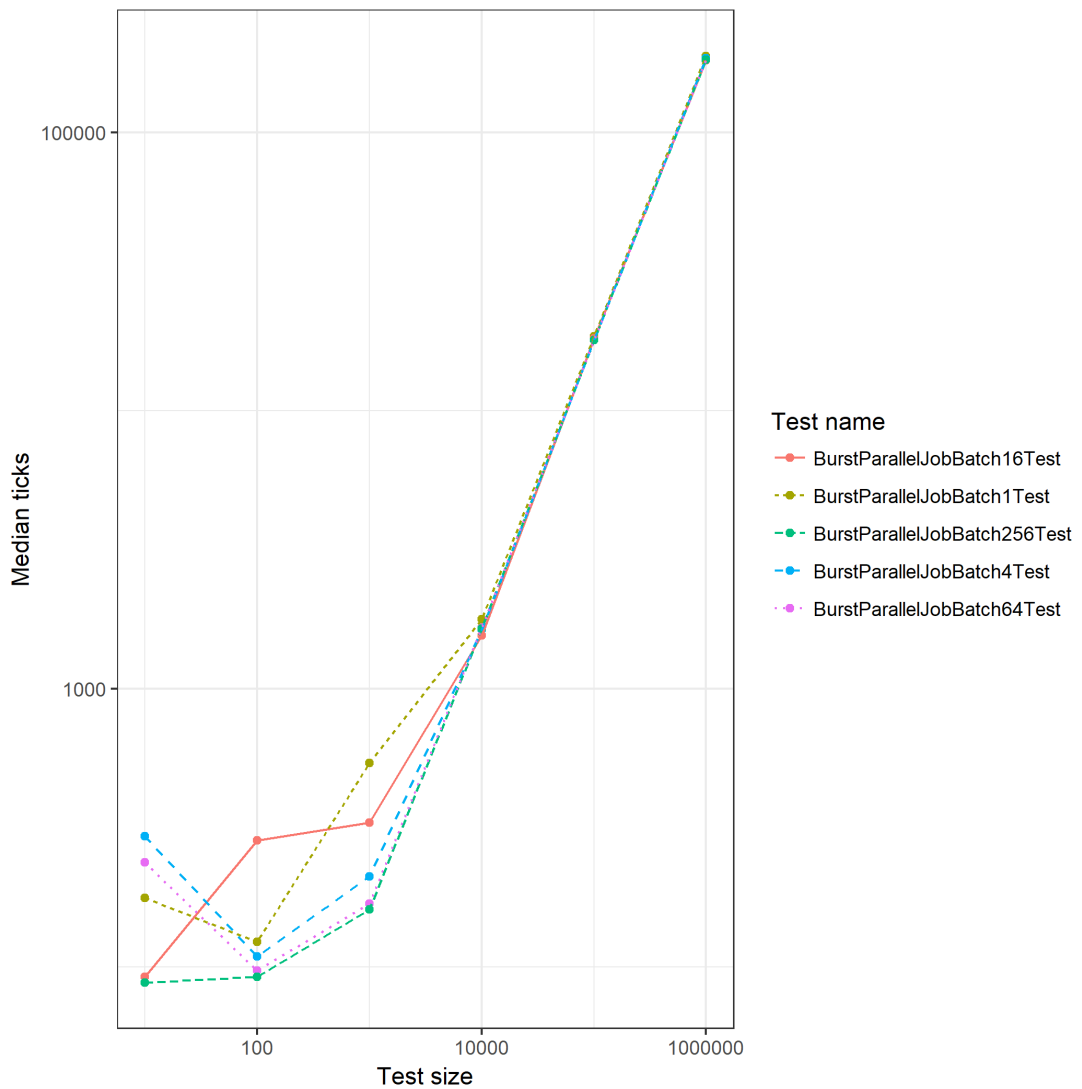


Figure 5.7: Batch Size Test Set - Matrix Multiplication - Burst Parallel.

Test Size	Batch of 1	Batch of 4	Batch of 16	Batch of 64	Batch of 256
10	177	295	92	238	88
100	123	109	285	97	92
1 000	540	212	330	169	161
10 000	1781	1633	1555	1661	1646
100 000	18457	18035	18133	17914	17929
1 000 000	188080	184397	181905	182046	182026

Table 5.7: Batch Size Test Set - Matrix Multiplication - Burst Parallel

Based on the results from both float assignment and matrix multiplication test sets, it is sensible to set the value of the batch size parameter to 64 when running the parallel jobs for calculations, where each iteration of the calculations is small. This is valid independently of whether the burst compilation is on or off. Only for more extensive calculations it makes sense to lower the value of the parameter.

Furthermore, the performance testing suite also contains other Batch Size Test Sets, e.g. dot product, cross product, etc., see Appendix B.

## 5.5 Job Type Test Sets

Job Type Test Sets, is the name of the group of test sets, which compare a specific calculation on same data, based on the type of the job used to transmit the data:

- No Job - method, which passes data between C# arrays
- Job - IJob using NativeArray
- Parallel Job - IJobParallelFor using NativeArray
- Burst Job - Burst compiled IJob using NativeArray
- Burst Parallel Job - Burst compiled IJobParallelFor using NativeArray
- Burst 4 Jobs - four Burst compiled IJobs running on four smaller NativeArrays in parallel

For jobs using IJobParallelFor we set the value of the batch size to 64, based on the findings from 5.4.

### 5.5.1 Float Assignment

Firstly, the underlying calculation is assignment of float from one data structure to another.

Table 5.8 displays the median elapsed ticks of each test based on the test size. The figure 5.8 depicts the relationship in a line chart.

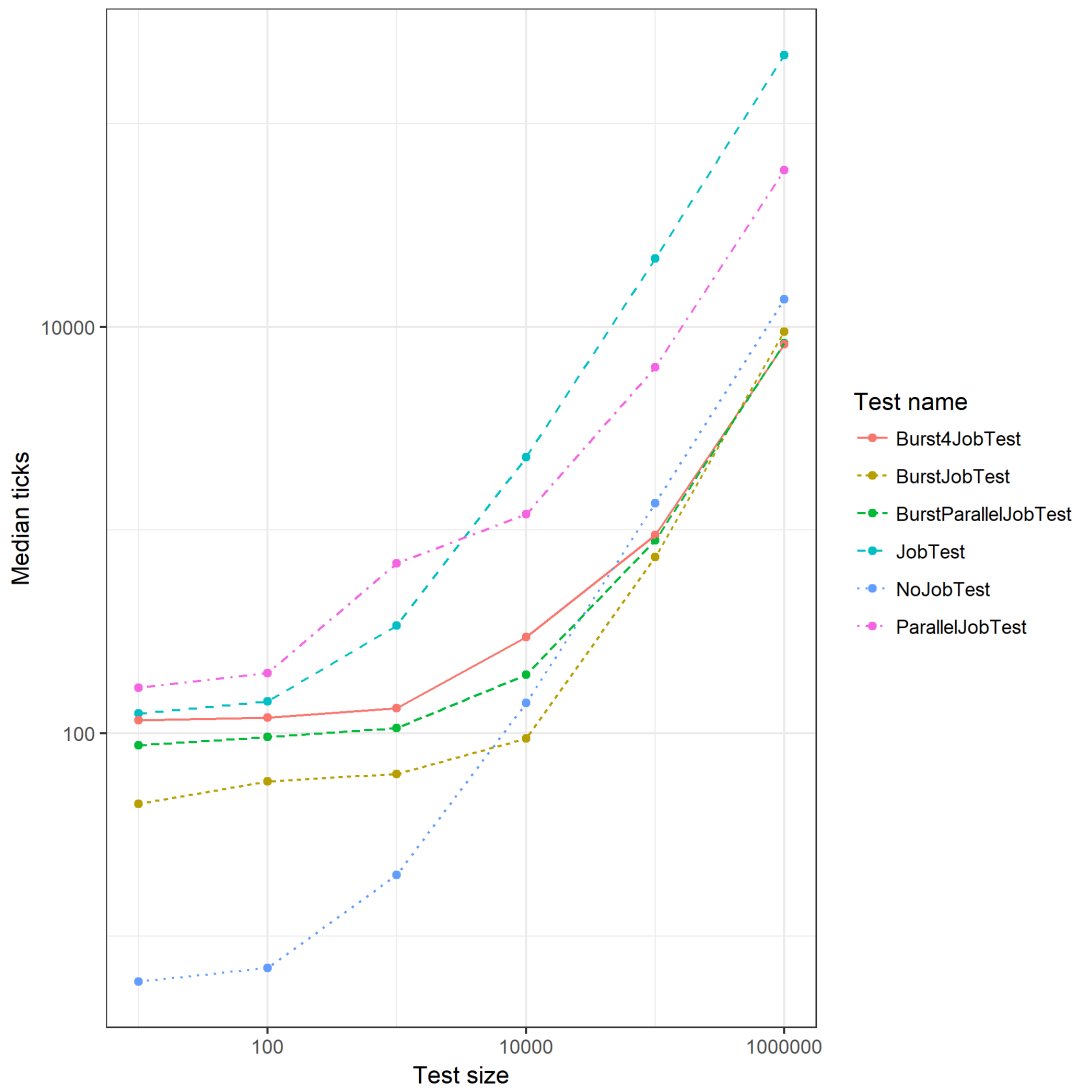


Figure 5.8: Job Test Set - Assign Float.

Test Size	No Job	Job	Parallel Job
10	6	125	167
100	7	143	198
1 000	20	338	685
10 000	141	2278	1198
100 000	1357	21699	6337
1 000 000	13657	217274	58878
Test Size	Burst Job	Burst Parallel Job	Burst 4 Jobs
10	45	87	116
100	58	96	119
1 000	63	106	133
10 000	94	194	298
100 000	737	891	941
1 000 000	9430	8303	8201

Table 5.8: Job Test Set - Assign Float

## 5.5.2 Matrix Multiplication

Furthermore, the next calculation is the matrix multiplication of two 4x4 matrices.

Table 5.9 presents the median elapsed ticks of each test based on the test size. The figure 5.9, again, depicts the relationship in a line chart.

Test Size	No Job	Job	Parallel Job
10	39	195	210
100	287	453	846
1 000	2771	3339	1515
10 000	27647	32283	9136
100 000	277874	323290	85245
1 000 000	2801598	3236772	843471
Test Size	Burst Job	Burst Parallel Job	Burst 4 Jobs
10	40	103	291
100	47	97	132
1 000	107	176	190
10 000	1390	1662	1815
100 000	18304	18207	18317
1 000 000	185398	182035	184182

Table 5.9: Job Test Set - Matrix Multiplication

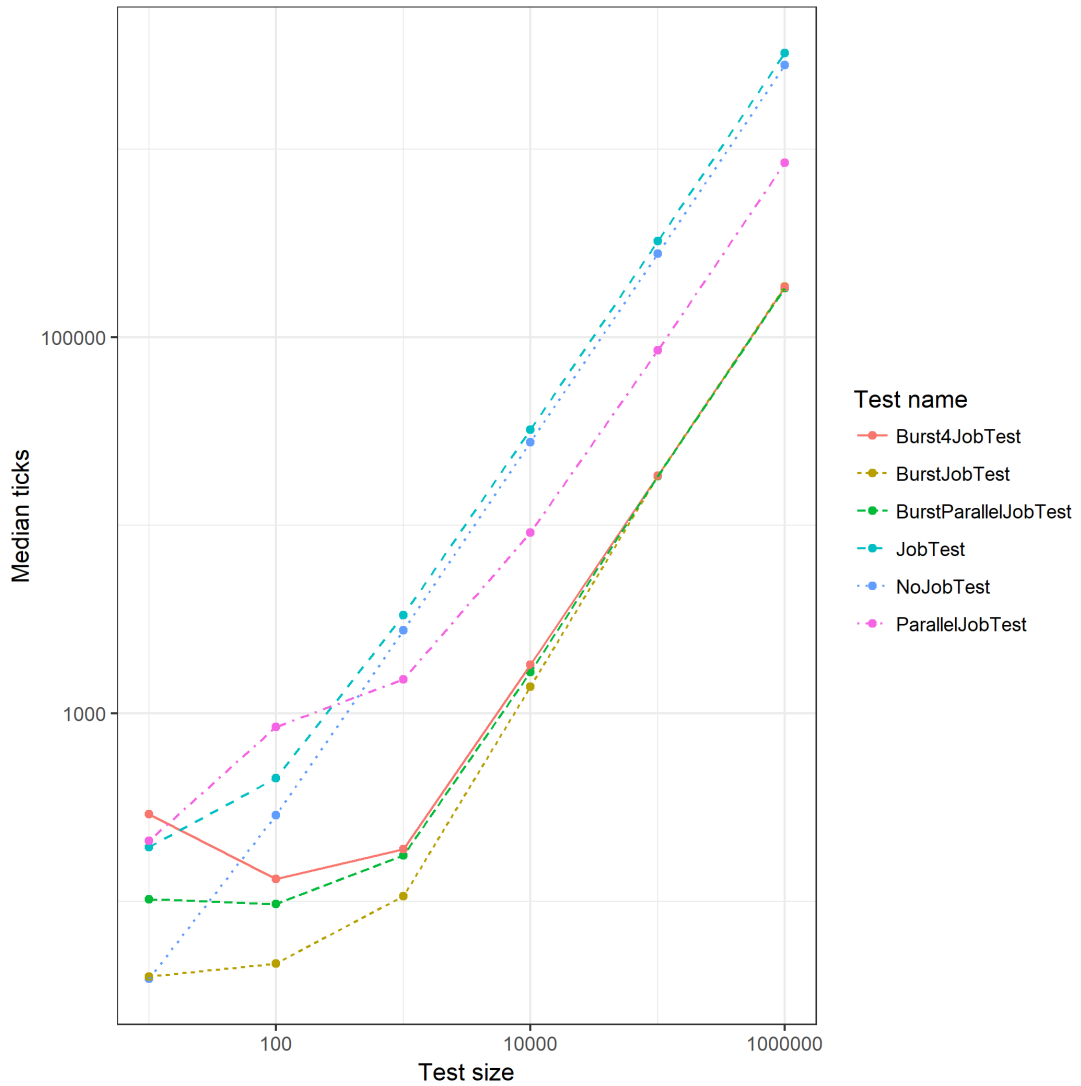


Figure 5.9: Job Test Set - Matrix Multiplication.

As the results of the tests show, running the burst compiled versions of the jobs are significantly faster than their counterparts. This is attributed to the fact that Burst takes advantage of data parallelism thanks to SIMD instructions.

Moreover, it has been shown that in case of a small number of simple operations, e.g. assignment of 1000 floats or less from one array to another, the IJob outperforms IJobParallelFor. However, for assignment of 10 000 floats, the parallel version of the job gives better results than the single-threaded job. On the other hand, for a more complex operation, e.g. matrix multiplication, the parallel alternative outperforms its non-parallel counterpart even when running on 1000 matrices. However, when these calculations are run with Burst enabled, IJobParallelFor starts to surpass the performance of IJob at much higher tests sizes (1 000 000 for float assignment and 100 000 for matrix multiplication). Next, the manual scheduling of 4 IJob jobs turned out to be even slower than the IJobParallelFor in both examples.

Furthermore, the performance of assignment using C# array dominates the per-

formance of the assignment using jobs and NativeArrays for small data sets. However, C# arrays can be not jobified without managed allocations neither compiled by Burst and therefore it is recommended to use NativeArrays with Burst jobs.

Consequently, it is recommended to use the Burst compiler whenever possible and in case of very large calculations, it is advisable to use IJobParallelFor jobs. Otherwise, for small datasets and small calculations the IJob jobs should be preferred. The manual splitting of the dataset and scheduling multiple IJob jobs instead of a IJobParallelFor should be avoided.

Moreover, the performance testing suite also contains examples of Job Type Test Sets, e.g. cross product, distance computation, etc., see Appendix B.

## 5.6 Float Optimization Test Sets

Float Optimization Test Sets consists of two test sets - Float Mode Test Set and Float Precision Test Set. These test sets compare tests with different degree of optimizations, which Burst compiler is able to perform, see subsection 2.3.3.

### 5.6.1 Float Mode Test Sets

The Float Mode Test Sets compare two tests with identical underlying calculations with different Burst option of FloatMode flag. One is using FloatMode.Strict (the default mode), while the other is using FloatMode.Fast.

First calculation for this test is the matrix multiplication of two 4x4 matrices. Table 5.10 shows the median elapsed ticks of each test based on the test size, whereas figure 5.10 depicts the relationship in a line chart.

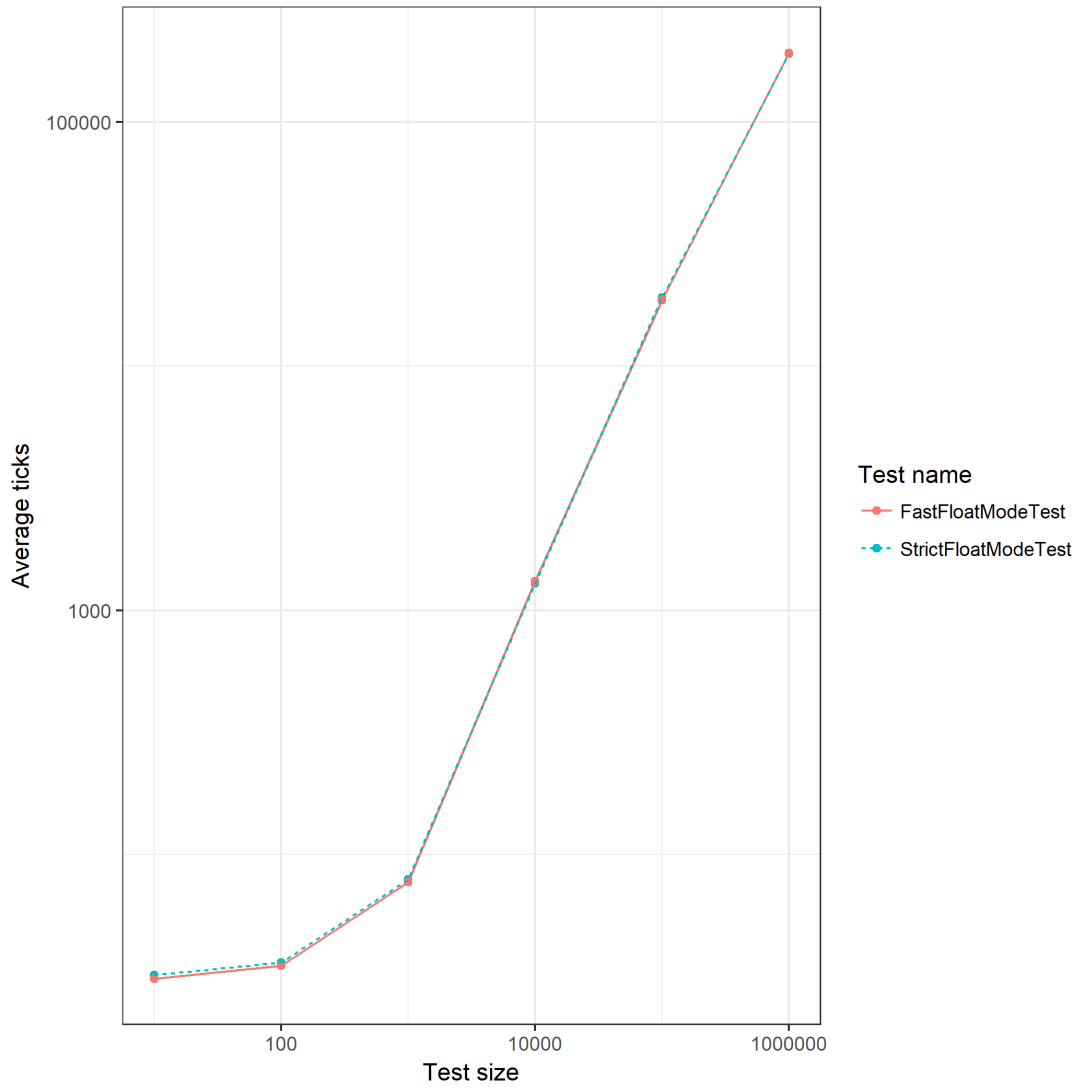


Figure 5.10: Float Mode Test Set - Matrix Multiplication.

Test Size	Strict Float Mode	Fast Float Mode
10	32	31
100	36	35
1 000	79	77
10 000	1289	1312
100 000	19078	18603
1 000 000	190616	191462

Table 5.10: Float Mode Test Set - Matrix Multiplication.

The next calculation for the Float Mode Test Set is the computation of sine of a float. Table 5.11 depicts the median elapsed ticks of each test based on the test size, while figure 5.11 visualizes the relationship in a line chart.

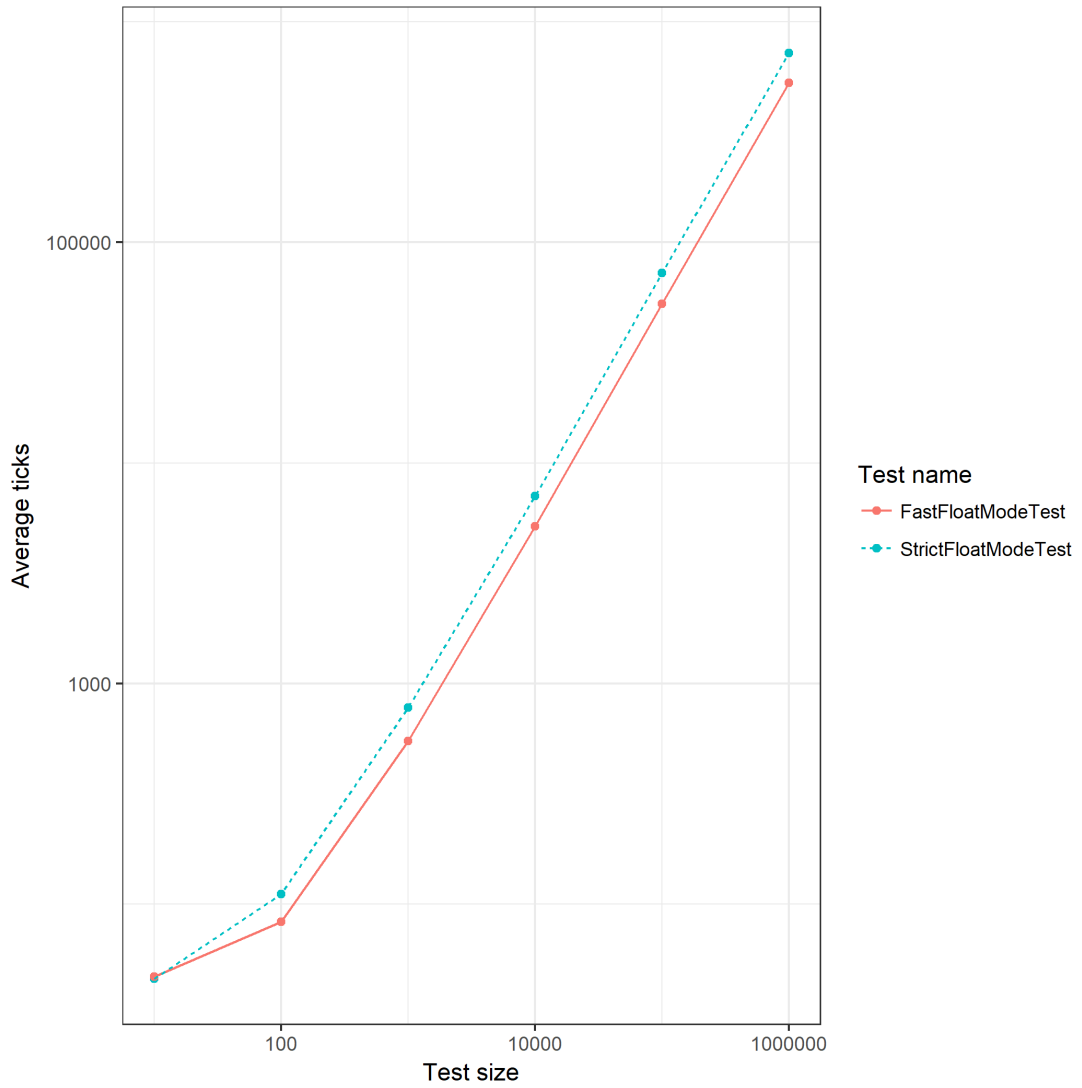


Figure 5.11: Float Mode Test Set - Sine.

Test Size	Strict Float Mode	Fast Float Mode
10	46	47
100	111	83
1 000	780	549
10 000	7072	5165
100 000	72508	52631
1 000 000	721557	527904

Table 5.11: Float Mode Test Set - Sine.

It has been shown that using different float modes has performance impact in case of the sine computation. In this case, the fast float mode outperforms the strict float mode. However, for matrix multiplication calculation, there were no significant performance differences measured between the float modes.

## 5.6.2 Float Precision Test Sets

The Float Precision Test Sets, again, compare two tests with identical underlying calculations. One is using `FloatPrecision.Medium` (the default precision mode), while the other is using `FloatPrecision.High`. We compare the results for matrix multiplication of 4x4 matrices at first and then the results for sine of a float are evaluated.

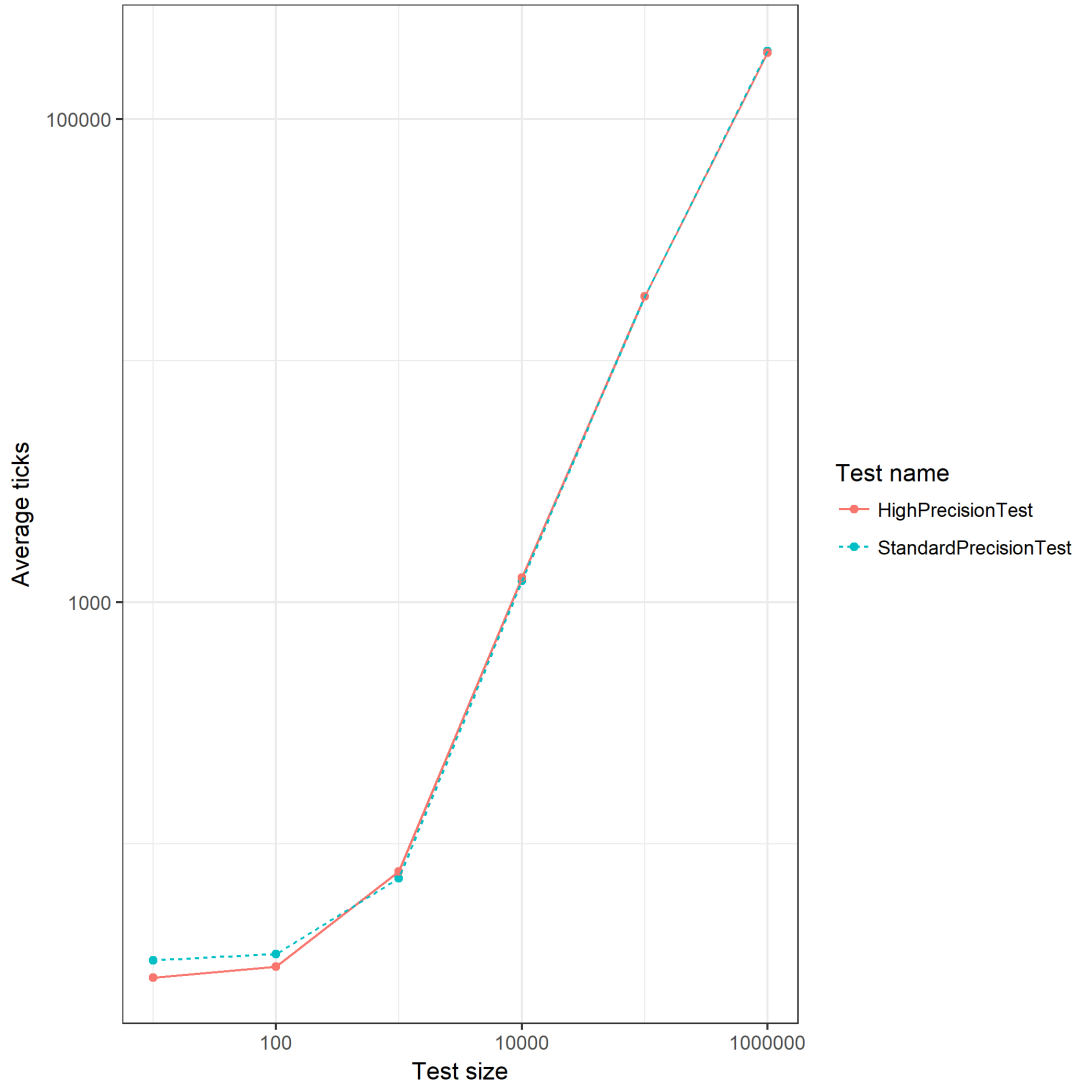


Figure 5.12: Float Precision Test Set - Matrix Multiplication.

Test Size	Standard Float Precision	High Float Precision
10	33	28
100	35	31
1 000	72	77
10 000	1231	1267
100 000	18508	18509
1 000 000	191453	188912

Table 5.12: Float Precision Test Set - Matrix Multiplication.

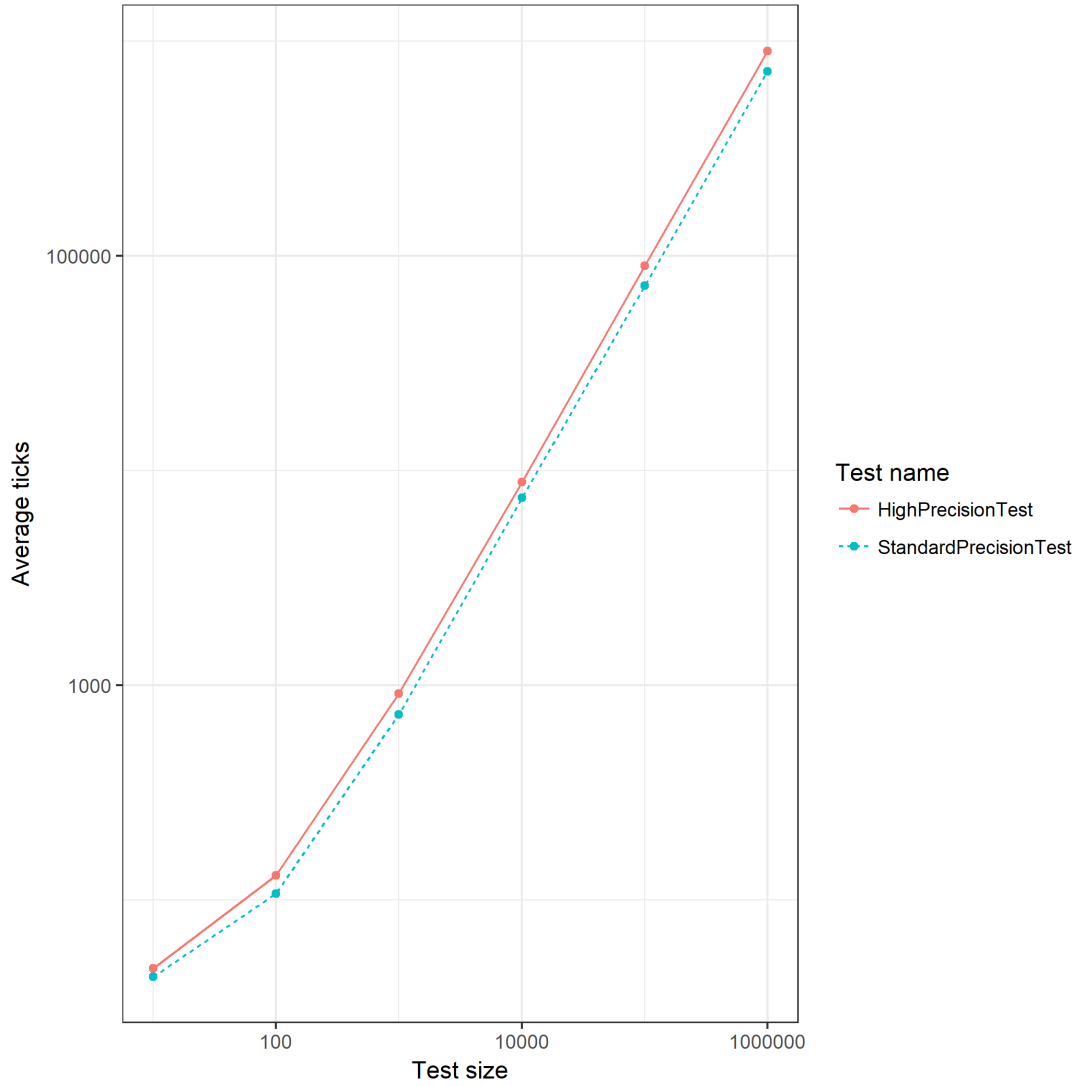


Figure 5.13: Float Precision Test Set - Sine.

Test Size	Standard Float Precision	High Float Precision
10	44	48
100	107	130
1 000	733	913
10 000	7472	8835
100 000	72608	90083
1 000 000	722549	900389

Table 5.13: Float Precision Test Set - Sine.

The results show that different settings for float precision do not matter in case of matrix multiplication, whereas it shows that using standard float precision results in better performance in case of computation of sine, similarly to the Float Mode Test Sets.

As a result, high-performance solutions, for which very high precision is not re-

quired, it is recommended to use the `FloatMode.Fast` and `FloatPrecision.Standard` flags for the Burst compiler as it could boost performance in certain cases.

## 5.7 Math Test Sets

Math Test Sets, compare the implementation of mathematical functions from `UnityEngine` namespace, against the new implementation inside `Unity.Mathematics` namespace. The implementation from `Unity.Mathematics` namespace is supposed to be optimized for Burst to generate efficient SIMD instructions<sup>1</sup>.

In this scenario, two test sets will be taken as an example - calculation of dot product and matrix multiplication. The results for dot product are presented in table 5.14 and figure 5.14, while the results for matrix multiplication are shown in table 5.15 and figure 5.15.

---

<sup>1</sup><https://github.com/Unity-Technologies/Unity.Mathematics> (Accessed 15th December, 2019)

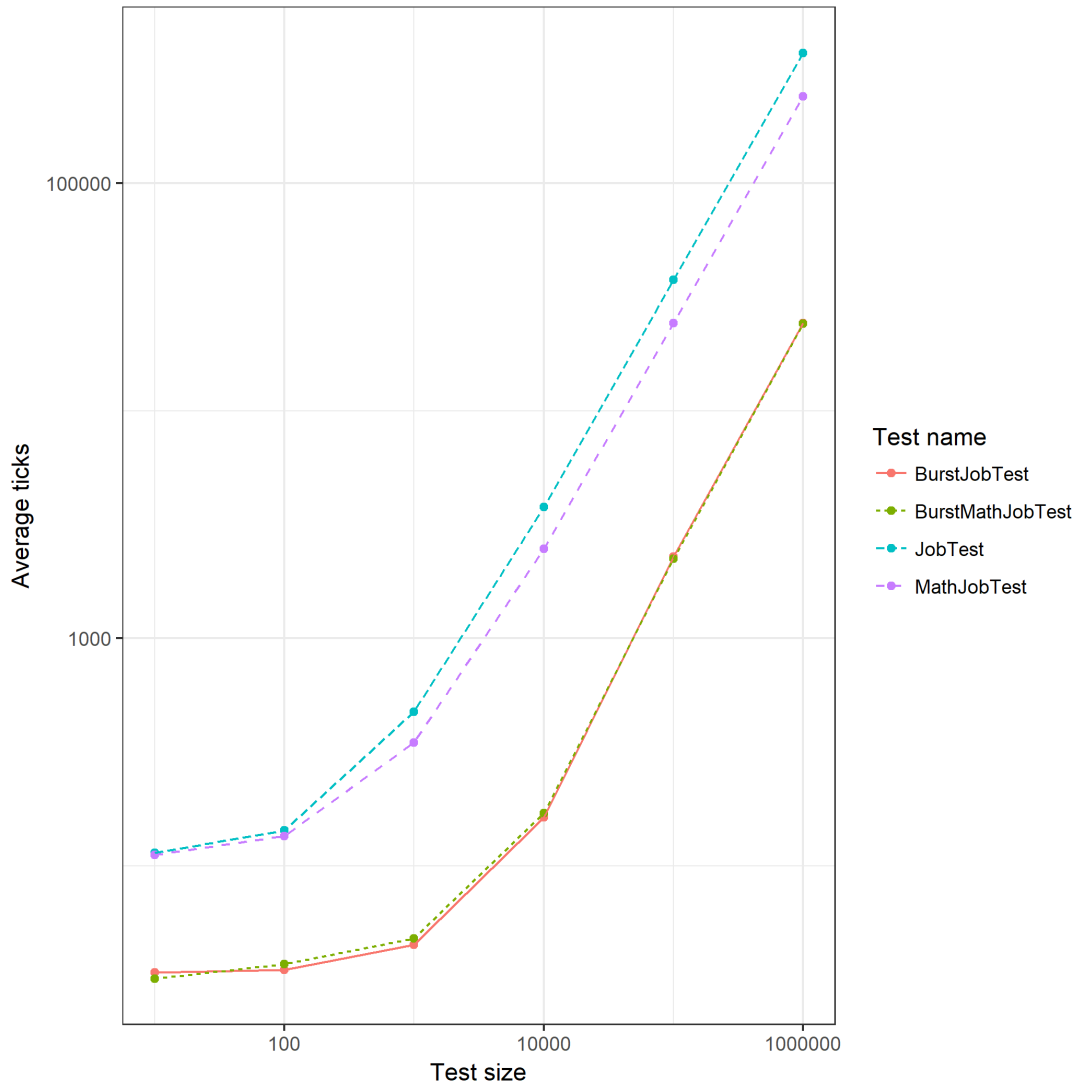


Figure 5.14: Math Test Set - Dot Product.

Test Size	Job	Math Job	Burst Job	Burst Math Job
10	114	112	34	32
100	143	135	35	37
1 000	477	348	45	48
10 000	3774	2474	164	171
100 000	37573	24215	2281	2242
1 000 000	373227	240390	24211	24116

Table 5.14: Math Test Set - Dot Product.

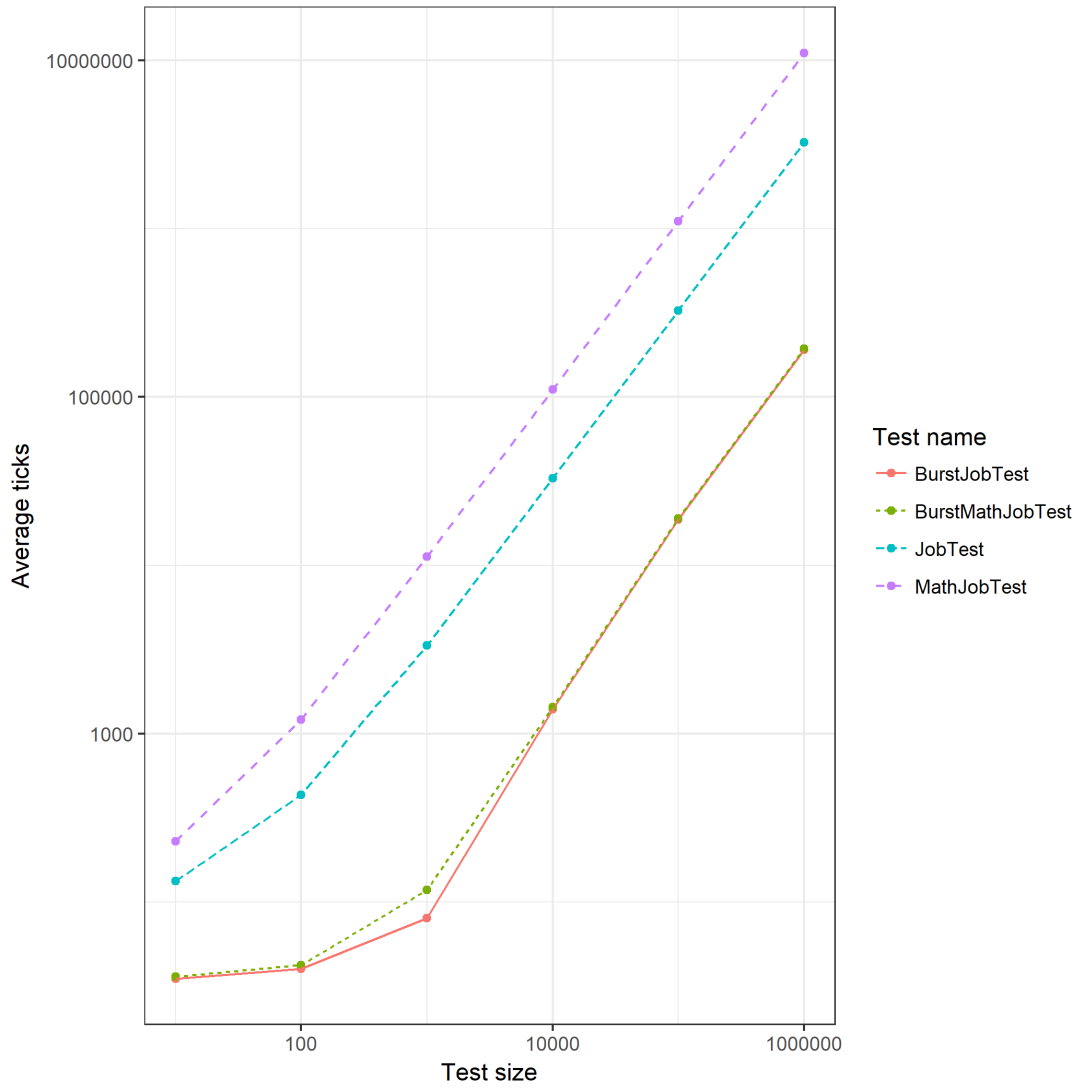


Figure 5.15: Math Test Set - Matrix Multiplication.

Test Size	Job	Math Job	Burst Job	Burst Math Job
10	133	229	35	36
100	432	1206	40	42
1 000	3331	11262	80	118
10 000	32823	110990	1398	1437
100 000	326817	1107354	18649	18940
1 000 000	3267917	11071588	190629	192930

Table 5.15: Math Test Set - Matrix Multiplication.

The results for the computation of the dot product show that using the algorithm from the Unity.Mathematics namespace gives better results than the older version from UnityEngine namespace, when the Burst compiler is not used. However, when using Burst, both algorithms have comparable performance.

In case of matrix multiplication, it comes as a surprise that the non-Burst version

produces exactly opposite results as in the case of computation of dot product. The old implementation is faster than the algorithm from `Unity.Mathematics` namespace. In case when using Burst, both implementations have, again, comparable performance.

As a result, it is not straightforward, which implementation of the same algorithm should be used when implementing high-performance solutions.

The performance testing suite also contains other Math Test Sets, such as computation of distance, etc., see Appendix B.

## 5.8 Recommendations

Based on the evaluations of the test sets in previous section, we can create a list of recommendations for implementing high-performance solutions within Unity DOTS. Below the recommendations from previous sections are summarized:

- Accessing member variables by simple getter and setter properties, auto-implemented properties and methods does not cause any additional overhead by Burst
- Burst operates equally on plain primitive value types and primitive value types wrapped in a structure. Using nested structures or structures implementing an interface results in equal performance as well.
- Use `Allocator.Temp` with Native Arrays if possible. When using Native Arrays in jobs, consider using `Allocator.Persistent` over `Allocator.TempJob`.
- Use Burst compiler with jobs whenever possible
- Choose `IJob` jobs over `IJobParallelFor` jobs when each iteration of the calculation is small, such as assignment of values from one Native Array to another or matrix multiplication. For more extensive calculations opt for `IJobParallelFor` jobs.
- When using `IJobParallelFor` set the value of the batch size parameter to 64 for standard operations, such as assignment of values or matrix multiplication. For larger calculations, use lower values of this parameter.
- Prefer `IJobParallelFor` job over several `IJob` jobs operating on several smaller datasets
- In cases for which high performance is more important than very high float precision, consider using `FloatMode.Fast` and `FloatPrecision.Standard` options for the Burst compilation
- In some cases, using the old implementation of mathematical functions from `UnityEngine` namespace yields better performance than the newer implementation from `Unity.Mathematics` namespace. However, this condition proved to be valid only if Burst compiler is not used.

## 5.9 Evaluation of the recommendations

In this section, we will evaluate the recommendations for the system on a larger real-time simulation. For the purpose of our evaluation, we chose the boids simulation.

The boids simulation, is similar to the behaviour of bird flocking. There are three forces, that influence the boid behavior<sup>2</sup>:

- Alignment - force leading boids to the average direction of nearby boids
- Cohesion - force pointing boids to the average position of nearby boids
- Separation - force preventing nearby boids from overcrowding

In our simulation, we use the aforementioned forces and we also add the barrier force, which prevents the boids from flying away from the viewport.

The boids simulation will be run with 500 boids and it will compare three scenarios:

- **Naive implementation using MonoBehaviors** - traditional approach with MonoBehaviors without using DOTS system
- **Implementation using DOTS** - the DOTS implementation using ECS, C# Job System and Burst compiler
- **Implementation using DOTS with recommendations** - DOTS implementation using ECS, C# Job System and Burst compiler, which takes into account the recommendations from section 5.8

Regarding the implementation using DOTS with recommendations, we will manually schedule `IJobParallelFor` jobs for neighbour boid collection, computation of each force and subsequent application of the forces. Because all the calculations for each job need to iterate over the other boids, they are rather extensive and therefore we will set batch size parameter of the jobs to 1. Finally, all jobs will be compiled by Burst. However, we refrain from using `FloatMode.Fast` flag for Burst compilation, as it produces worse results in our simulation.

Firstly, we will measure the frame times of 100 frames after 5 seconds elapsed. This is done in order to avoid the costs of initialization of the simulation in the first frame. At this time, the boids are starting to flock with other boids. Moreover, the frame times of 100 frames after 180 seconds elapsed will be measured. This is done to account for the case, when boids are already flocked into groups and more data need to be processed, because of the increased number of neighbours.

---

<sup>2</sup><https://cs.stanford.edu/people/eroberts/courses/soco/projects/2008-09/modeling-natural-systems/boids.html> (Accessed 20th December, 2019)

Table 5.16 depicts the results of the boids simulation. The measured unit was the median frame time (in milliseconds) of 100 frames after 5 seconds and the median frame time (in milliseconds) of 100 frames after 180 seconds.

Test Type	Mono	DOTS	DOTS with recommendations
Frame time after 5 seconds	32,71	1,06	0,98
Frame time after 180 seconds	43,14	1,77	1,33

Table 5.16: Boids Simulation Results.

The results of the boids simulation have shown that the implementation using Unity DOTS dominates the naive implementation using MonoBehaviors. This is because the computation of the forces can be efficiently run in parallel on multiple threads and the instructions can be vectorized by Burst.

When we consider and apply the recommendations, we can see that some extra frame time can be saved. Although the saved time does not seem to be significant, we have to take into account that the implementation without recommendations is already heavily optimized by the DOTS system itself.

## 6. Conclusion

In this thesis, we tried to evaluate the performance of various features of the Unity DOTS system and generate a list of generally usable recommendations, which should be taken into account when implementing high-performance solutions within Unity DOTS.

Firstly, we implemented the performance testing suite for benchmarking custom user tests in general. Moreover, the set of tests stressing the respective parts of the system was created and evaluated. After the evaluation of the test results, we created a list of recommendations to be considered when writing code in Unity DOTS. The majority of the executed tests produced expected results. Nevertheless, some tests produced surprising results, e.g. the tests comparing old and new implementation of mathematical functions.

In order to evaluate the recommendations based on the results of the tests, we implemented the boids simulation, which is a simulation applicable to real-time modern video games. The boids simulation aimed to see if using these recommendation can yield additional performance and it has shown that cautious selection of some of those recommendations can produce better performance results.

### 6.1 Future Work

One of the possibilities how to extend the performance testing suite is to write and benchmark custom user tests, as the suite was written with extensibility in mind. The custom tests can be written not only for Unity DOTS system, but also for any piece of code in general.

Another way how to improve the current state of the performance testing suite would be the creation of visualization of the test results directly in Unity Editor or inside standalone application. This would be beneficial for the possible users/testers as it would decrease the time required to process and visualize the data.

Furthermore, it would be useful to evaluate the recommendations in other high-performance real-time scenarios, which are typically used in more complex simulations or modern video games.

# Bibliography

- Arm Limited. Optimizing C Code with Neon Intrinsic, 2019. URL <https://developer.arm.com/architectures/instruction-sets/simd-isas/neon/neon-programmers-guide-for-armv8-a/optimizing-c-code-with-neon-intrinsics/what-is-neon>. Accessed 15 November 2019.
- C. Ferreira and M. Geig. Get Started with the Unity Entity Component System (ECS), C# Job System, and Burst Compiler, 2018. URL <https://software.intel.com/en-us/articles/get-started-with-the-unity-entity-component-system-ecs-c-sharp-job-system-and-burst-compiler>. Accessed 5 October 2019.
- Google. Neon Support, 2019. URL <https://developer.android.com/ndk/guides/cpu-arm-neon>. Accessed 9 November 2019.
- J. Gregory. *Game Engine Architecture*. Third Edition. A K Peters/CRC Press, 2018. ISBN 1138035459.
- Microsoft. ref - C# Reference, 2019a. URL <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/ref>. Accessed 9 November 2019.
- Microsoft. What's new in C# 7.0, 2019b. URL <https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-7#ref-locals-and-returns>. Accessed 20 December 2019.
- Microsoft. Stopwatch Class, 2019c. URL <https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.stopwatch?view=netframework-4.0>. Accessed 9 November 2019.
- Pitaksarit, S. Analyzing Burst generated assemblies, 2019. URL <https://gametorrahod.com/analyzing-burst-generated-assemblies/>. Accessed 18 November 2019.
- Unity Technologies. Creating jobs, 2018a. URL <https://docs.unity3d.com/2019.1/Documentation/Manual/JobSystemCreatingJobs.html>. Accessed 16 November 2019.
- Unity Technologies. DOTS - Unity's new multithreaded Data-Oriented Technology Stack, 2018b. URL <https://unity.com/dots>. Accessed 5 October 2019.
- Unity Technologies. NativeContainer, 2018c. URL <https://docs.unity3d.com/2019.1/Documentation/Manual/JobSystemNativeContainer.html>. Accessed 16 November 2019.
- Unity Technologies. The safety system in the C# Job System, 2018d. URL <https://docs.unity3d.com/2019.1/Documentation/Manual/JobSystemSafetySystem.html>. Accessed 16 November 2019.

- Unity Technologies. Scheduling jobs, 2018e. URL <https://docs.unity3d.com/2019.1/Documentation/Manual/JobSystemSchedulingJobs.html>. Accessed 16 November 2019.
- Unity Technologies. Performance Testing Extension for Unity Test Runner, 2019a. URL <https://docs.unity3d.com/Packages/com.unity.test-framework.performance@1.3/manual/>. Accessed 9 November 2019.
- Unity Technologies. Unity Test Runner, 2019b. URL <https://docs.unity3d.com/2019.1/Documentation/Manual/testing-editorortestsrunner.html>. Accessed 9 November 2019.
- Unity Technologies. Burst User Guide, 2019c. URL <https://docs.unity3d.com/Packages/com.unity.burst@1.1/manual/index.html>. Accessed 5 October 2019.
- Unity Technologies. Profiler overview, 2019d. URL <https://docs.unity3d.com/Manual/Profiler.html>. Accessed 5 October 2019.
- B. Watson. *Writing High-Performance .NET Code*. Watson, B., 2014. ISBN 0-99058-342-2.

# List of Figures

2.1	Burst Inspector. . . . .	18
2.2	Unity Profiler Window. . . . .	20
2.3	Unity Test Runner. . . . .	22
4.1	Unity Editor Menu. . . . .	27
4.2	Unity Editor Test Set Runner. . . . .	27
4.3	Standalone Test Set Runner. . . . .	28
5.1	Wrapper Test Set. . . . .	33
5.2	Accessor Test Set. . . . .	35
5.3	Allocator Test Set. . . . .	37
5.4	Batch Size Test Set - Assign Float - Parallel. . . . .	39
5.5	Batch Size Test Set - Assign Float - Burst Parallel. . . . .	40
5.6	Batch Size Test Set - Matrix Multiplication - Parallel. . . . .	41
5.7	Batch Size Test Set - Matrix Multiplication - Burst Parallel. . . . .	42
5.8	Job Test Set - Assign Float. . . . .	44
5.9	Job Test Set - Matrix Multiplication. . . . .	46
5.10	Float Mode Test Set - Matrix Multiplication. . . . .	48
5.11	Float Mode Test Set - Sine. . . . .	49
5.12	Float Precision Test Set - Matrix Multiplication. . . . .	50
5.13	Float Precision Test Set - Sine. . . . .	51
5.14	Math Test Set - Dot Product. . . . .	53
5.15	Math Test Set - Matrix Multiplication. . . . .	54

# List of Tables

3.1	Unity DOTS Benchmark Results . . . . .	23
5.1	Wrapper Test Set . . . . .	33
5.2	Accessor Test Set . . . . .	35
5.3	Allocator Test Set. . . . .	36
5.4	Batch Size Test Set - Assign Float - Parallel . . . . .	39
5.5	Batch Size Test Set - Assign Float - Burst Parallel . . . . .	40
5.6	Batch Size Test Set - Matrix Multiplication - Parallel . . . . .	42
5.7	Batch Size Test Set - Matrix Multiplication - Burst Parallel . . . . .	43
5.8	Job Test Set - Assign Float . . . . .	45
5.9	Job Test Set - Matrix Multiplication . . . . .	45
5.10	Float Mode Test Set - Matrix Multiplication. . . . .	48
5.11	Float Mode Test Set - Sine. . . . .	49
5.12	Float Precision Test Set - Matrix Multiplication. . . . .	50
5.13	Float Precision Test Set - Sine. . . . .	51
5.14	Math Test Set - Dot Product. . . . .	53
5.15	Math Test Set - Matrix Multiplication. . . . .	54
5.16	Boids Simulation Results. . . . .	57

# List of Abbreviations

CPU - Central processing unity

CSV - Comma-separated values

DOTS - Data oriented tech stack

ECS - Entity component system

GPU - Graphics processing unit

IL - Intermediate language

IR - Intermediate representation

RAM - Random access memory

SIMD - Single instruction multiple data

UI - User interface

# A. User Documentation

The Performance Testing Suite for Unity DOTS is shipped with a zip package, which contains following folders:

- **PTS Editor** - contains package for Unity Editor in order to be able to run the PTS within the editor, run Boid simulation within Editor and to be able to build the PTS standalone application and Boid simulation
- **PTS Windows** - consists of the executable for running the PTS in standalone build on Windows
- **PTS Android** - contains installation file for running the PTS in standalone build on Android
- **Boid Mono** - executable for running the Boid simulation with MonoBehaviour on Windows
- **Boid DOTS** - executable for running the Boid simulation with Unity DOTS on Windows
- **Boid DOTS Recommended** - executable for running the Boid simulation with Unity DOTS on Windows, where the recommendations from section 5.8 are taken into account

The following sections describe the system requirements for running the Performance Testing Suite for Unity DOTS and Boid Simulation inside Unity Editor on Windows and in standalone builds on Windows and Android. The information about the installation process is also contained.

## A.1 Unity Editor

### A.1.1 System Requirements

It is required to use Unity 2019.2.9 or newer<sup>1</sup>. The system requirements for Unity 2019.2.9 are following:

- 64-bit versions of Windows 7 SP1+, Windows 8 or Windows 10.
- CPU with SSE2 instruction set support.
- Graphics card with DX10 (shader model 4.0) capabilities.

For the Unity project, approximately 400 MB on disc.

---

<sup>1</sup><https://unity3d.com/unity/whats-new/2019.2.9> (Accessed 23th November 2019)

## A.1.2 Installation Instructions

Firstly, Unity3D 2019.2.9 has to be downloaded<sup>2</sup>.

In order to be able to use Unity DOTS features, the appropriate packages need to be installed:

- In Unity, in Menu bar, open the tab Window/Package Manager.
- In the upper part, click on Advanced button and Show preview packages
- Install Burst, Collections, Entities, Jobs, Hybrid Renderer and Mathematics packages by hitting the Install button.

The file PTS.unitypackage, which contains the PTS package can be found at PTS Editor folder.

The package can be installed as follows:

- In the Menu bar inside Unity, open the tab Assets/Import Package/Custom Package...
- Import the PTS.unitypackage
- Unity forces automatic reload of all assets and the plugin should be loaded
- PTS tab in the Menu Bar should be visible and the Standalone Build Scene, Boid Mono Scene, Boid DOTS Scene and Boid DOTS Recommended Scene should be available in the Assets/Scenes

## A.2 PTS Windows Standalone Build

### A.2.1 System Requirements

The system requirements for Windows standalone build match the requirements for the Unity Editor, the target CPU architecture of the PTS is 64-bits. The build takes approximately 55 MB on disc.

### A.2.2 Installation Instructions

In order to launch the Windows standalone build of the PTS, it suffices to launch the PTS.exe application, which is located in the folder named PTS Windows.

---

<sup>2</sup><https://unity3d.com/get-unity/download/archive> (Accessed 23th November 2019)

## **A.3 PTS Android Standalone Build**

### **A.3.1 System Requirements**

- OS 4.1 or later
- ARMv7 CPU with NEON support or Atom CPU
- OpenGL ES 2.0 or later

The installation package takes approximately 17 MB on disc.

### **A.3.2 Installation Instructions**

In order to launch the Android standalone build of the PTS, the PTS.apk installation executable inside the PTS Android folder should be imported to the Android device. Before the installation, it is necessary to allow the installations from unknown sources. After the installation, user should be able to launch the PTS application.

## **A.4 Boid Windows Standalone Build**

### **A.4.1 System Requirements**

Boid Mono, Boid DOTS and Boid DOTS Recommended standalone builds for Windows match the requirements for the Unity Editor, the target CPU architecture is 64-bits. The each build takes approximately 55 MB on disc.

### **A.4.2 Installation Instructions**

In order to launch the Boid Mono, it suffices to launch the Boid\_Mono.exe application, which is located in the folder named Boid Mono. Analogously, Boid DOTS can be launched via Boid\_DOTS.exe application, which is located in the Boid DOTS folder and Boid DOTS Recommended can be found at location Boid DOTS Recommended/Boid\_DOTS\_Recommended.exe.

## B. All Test Results

The results of all test sets of the Performance Testing Suite for Unity DOTS are shipped with a zip package and stored inside the folder named Test Results. The tests were executed within Standalone build on 2 desktop platforms - Desktop A, Desktop B and on 1 mobile device - Android, as described in chapter 5. The results for each platform are stored in its own folder at the root of the Test Results folder.

The results of the Boid simulation are contained at the root of the folder named Test Results, the files are named BoidMono.csv, BoidDOTS.csv and BoidDOTSRecommended.csv.