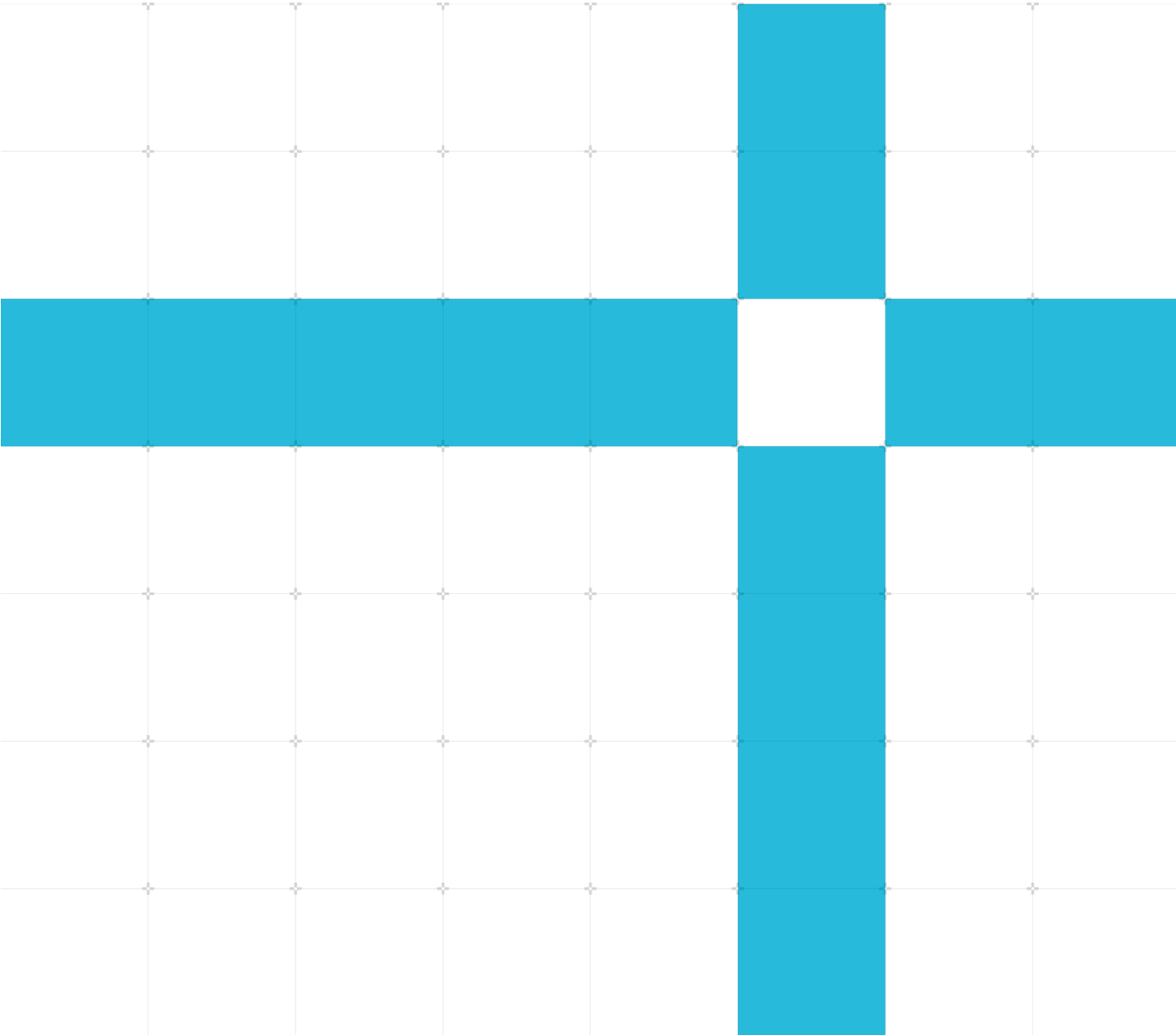
arm

Using Neon C# intrinsics with Unity Burst

Non-Confidential

Issue 1.0
102556



Using Neon C# intrinsics with Unity Burst

Document history

Issue	Date	Confidentiality	Change
1.0	16 June 2021	Non-Confidential	First release

Non-Confidential Proprietary Notice

Arm Limited. Company 02557590 registered in England.
110 Fulbourn Road, Cambridge, England CB1 9NJ.
(LES-PRE-20349)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Web Address

developer.arm.com

Progressive terminology commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used terms that can be offensive. Arm strives to lead the industry and create change.

Contents

1 Overview

This guide explains how you can use Arm Neon C# intrinsics with the Unity Burst compiler to improve performance of your Unity Android application.

By the end of this guide you will have learned:

- How Single Instruction Multiple Data (SIMD) instructions help improve performance by operating on data in parallel.
- Best practices for writing code that the compiler can automatically optimize to Neon instructions.
- How you can use Arm Neon intrinsics when the compiler misses Neon optimization opportunities.
- How to use Arm Neon intrinsics with the Unity Burst compiler to improve performance for Android applications in Unity.

1.1 Before you begin

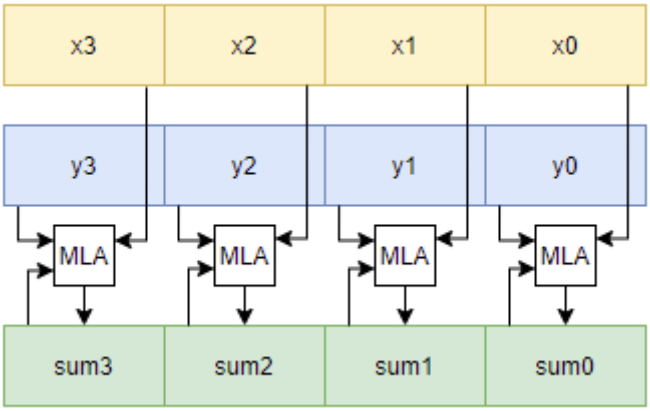
This guide assumes that you are familiar with Unity, C# programming, and Unity Burst.

To use Neon intrinsics in your code you must use Unity Burst compiler version 1.5 or above.

2 What are Neon intrinsics?

Neon intrinsics in the Unity Burst compiler let programmers write commands in their C# scripts that compile directly to specific Arm native instructions. The Neon set of instructions are Single Instruction Multiple Data (SIMD) instructions. SIMD instructions perform the same operation on multiple pieces of data in parallel.

The following diagram shows an example Neon instruction, Multiply-Accumulate (MLA). Each of the input registers x and y contain four separate elements of data. The MLA instruction multiplies individual elements in registers x and y together, then adds the result to any existing value in the corresponding element of the destination register:



In this example, the single MLA instruction performs four separate arithmetic operations:

- sum0 += x0 * y0
- sum1 += x1 * y1
- sum2 += x2 * y2
- sum3 += x3 * y3

The Neon instructions can operate on either 64-bit or 128-bit wide vectors, with individual data elements of 8, 16, 32 or 64-bits. It is therefore possible for a single Neon instruction to multiply four 32-bit float numbers by a constant in one operation, or to perform a logical operation on sixteen 8-bit booleans, for example.

To learn about all the operations that are available with Neon intrinsics, see intrinsics on Arm Developer.

There are many applications for Neon and SIMD calculations, including the following:

- Graphics
- Physics
- Machine learning
- Calculations that use matrices and vectors

The Neon instructions themselves are assembly instructions. Neon intrinsics let programmers use these assembly instructions in a high-level language without the overhead of writing assembly code by hand, leaving the compiler to take care of register allocation, for example. Neon intrinsics were originally developed for use in C and C++, but Unity has added the intrinsics into their Burst compiler for use in C# scripts.

Unity have implemented most of the Arm v8.0 Neon instructions. There are a few Arm v8.2 instructions, including dot product, which have been added to the Arm instruction set more recently and can currently only be used with an experimental define in Unity. The full list of available Intrinsics in Unity is available in the Unity API documentation.

3 Configuring Unity for Neon intrinsics

This section of the guide explains how to configure the Unity project and build settings to let you use Neon intrinsics in your Android application.

3.1 Build settings

To configure the build settings, follow these steps:

1. Click File > Build Settings on the unity main menu to display the Build Settings.

2. Click the Android category from the category list on the left. The Android build settings pane displays on the right.


Note

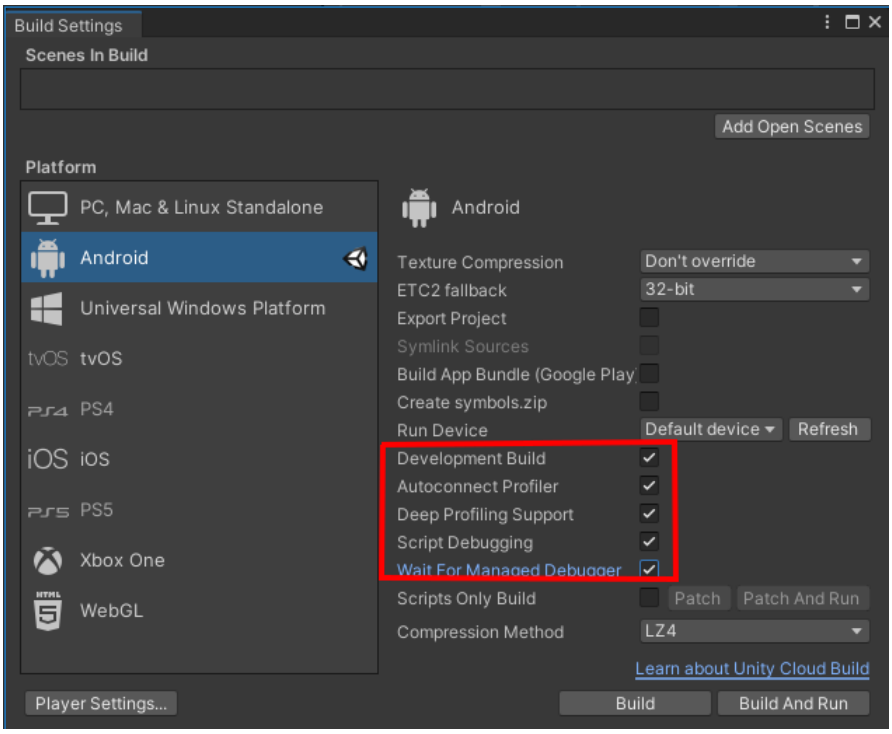
If your application is not already an Android application, you may need to click Switch Platform.

3. During application development, you might want to enable the following options to help with debugging:

- Development Build
- Autoconnect Profiler
- Script Debugging
- Wait For Managed Debugger

Once development of the scripts and functionality is complete, clear these build settings to ensure accuracy when profiling the performance.

The following image shows these settings:



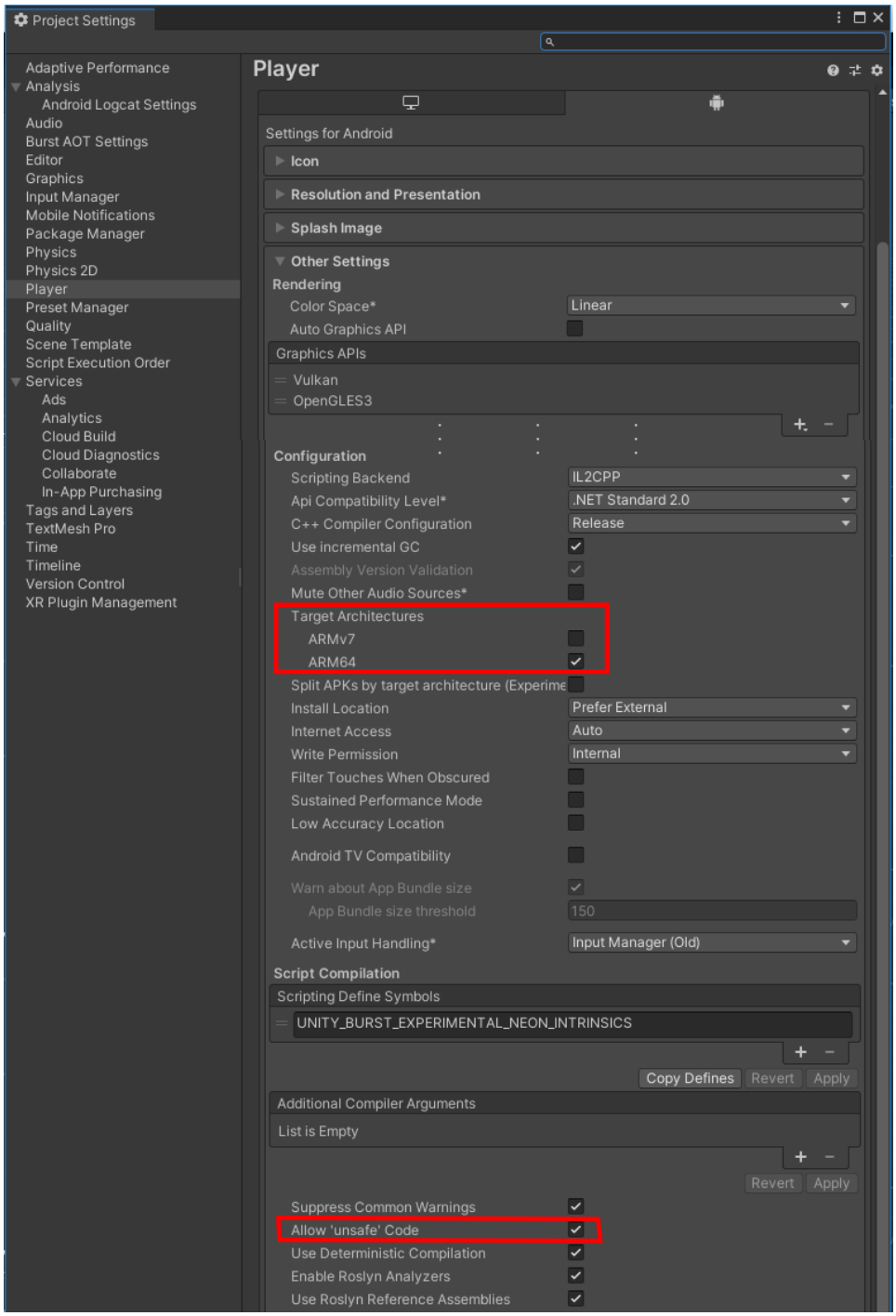
3.2 Project settings

To configure the project settings, follow these steps:

1. Click **Edit** > **Project Settings** on the Unity main menu to display the **Project Settings** dialog.
2. Click **Player** in the category list on the left. The details pane on the right shows the player settings.
3. Click the Android tab underneath the **Player** heading.
4. In the **Target Architectures** section:
   - Select the ARM64 checkbox to target the 64-bit version of Armv8.
   - Clear the ARMv7 checkbox.
5. In the **Script Compilation** section, select the **Allow 'unsafe' Code** checkbox.
6. For **Scripting Backend** select **IL2CPP**.

Targeting Arm64 hardware on Android disables the Mono scripting backend. You must use the IL2CPP scripting backend. This scripting backend converts your C# code into C++ which creates a much more efficient application than Mono.

The following image shows these settings:



4 Auto-vectorization best practice

The IL2CPP scripting backend uses the Clang compiler to convert your code to Arm instructions. The Clang compiler use of Neon is very mature, and in many cases can auto-vectorize your code to use Neon instructions without the need to use intrinsics.

However, you can improve the compiler's ability to auto-vectorize if you follow the best practice advice in this section.


Note

If your code uses the Burst compiler it does not go through the IL2CPP toolchain, but the following best practices still apply.

4.1 What to optimize

Think carefully about where you need performance improvement. Optimizing performance takes time and effort, and can result in code that is harder to read, so you should not try and optimize every single line of your code.

To optimize effectively, profile your application to discover where the bottlenecks in your code are. Then optimize the code at these bottlenecks. Finally, profile the code again to check what effect your changes made to the performance of your application.

Repeat the measure-optimize-check cycle for as many iterations as are required.

4.2 Code and data structure

For the compiler to efficiently auto-vectorize your code into Neon instructions, the structure of your code and the data it operates on needs to be straightforward. To give the compiler the best chance of auto-vectorizing your code, consider the following advice:

- Where the same operation is performed on multiple data items, keeping those data items together in memory will help auto-vectorization. For example, code that operates on data from a single sequential data array is usually more efficient than data that is scattered in more unpredictable patterns.
- Where the same operation is performed multiple times, try to have those operations performed sequentially in the code.

  For example, in the collisions example, for one optimization a structure is used with half the data having its sign changed so that all operations are greater-than rather than less-than. In another optimization, the x and y are processed separately so that the operations match.

4.3 Optimizing loops

Where there are repeated instructions that can be parallelized, there are loops. To improve the performance of your application consider applying the following optimizations to loops:

- Use multiple short simple loops instead of a single larger loops.
- Fixed-length `for` loops are easier to parallelize than `while` loops with arbitrary exit conditions.
- If possible, declare fixed length loops as multiple of 4 or 8 to remove the need for the compiler to generate tail-handling code, for example `i<(looplength & ~3)`. For more information on tail-handling code, see the Load and store-leftovers section in the Neon Programmer's Guide for Armv8-A: Coding for Neon.
- Avoid using break conditions in loops.
- Function calls made in the loop should be inlined if possible.
- Use simple indexing as any indirect addressing causes difficulties for the compiler to auto-vectorize.

4.4 Pointers

Pointers allow for memory addresses to be passed into functions. When passing pointers into `unsafe` functions, use the [NoAlias] attribute on the pointer parameters if there is no overlap in the data of the pointers passed. The attribute lets the compiler know it is safe to auto-vectorize.

4.5 Libraries

In addition to optimizing your code for SIMD, using libraries that are designed to be parallelized can improve the performance of your Android application.

The Unity.Mathematics library is a good example of a library with the structures and functions to use for improving the performance of your applications code.

5 Using Neon intrinsics

The benefit of using auto-vectorization is that the programmer can code their algorithm in a high-level language, and leave the compiler to convert it to Arm instructions. The disadvantage of auto-vectorization is that the compiler might fail to identify that a particular part of your code is vectorizable. In this situation, you can use intrinsics to ensure that compiler uses the Neon instructions you want.

Neon intrinsics are function calls that the compiler replaces with appropriate Neon instructions. Using Neon intrinsics gives you direct, low-level access to the exact Neon instructions that you want, all from C/C++ code.

The benefit of using intrinsics is that they provide almost as much control as writing assembly language, but leave details like register allocation to the compiler, so that developers can focus on the algorithms.

The disadvantage of programming with intrinsics is that it can be more complex than writing standard C/C++ code, and requires the programmer to learn about the available Neon intrinsics.

When using intrinsics, they must be added in a static Burst function, or Burst job, and wrapped in `if (IsNeonSupported)`. If added outside of a Burst function or job, or the application is not targeting the Arm64 architecture, `IsNeonSupported` always returns false. You can use an `else` statement to provide a fallback for running on non-Neon hardware or with Burst disabled.

The Unity Burst Inspector shows you what code is created at different stages of the pipeline. Where there are unexpected results, this tool can help show why.

After adding Neon intrinsics, and ensuring they are working, profile your code. If your code is slower, that usually indicates that the compiler was generating more highly optimized Neon code than you have written.

Unity provides a repository of available Neon intrinsics that can be used within your application, visit Unity Burst intrinsics Arm Neon for the repository. Arm also publishes a list of all the Neon intrinsics. There are a small number of Neon intrinsics that Unity does not support, which are documented in the Burst documentation.

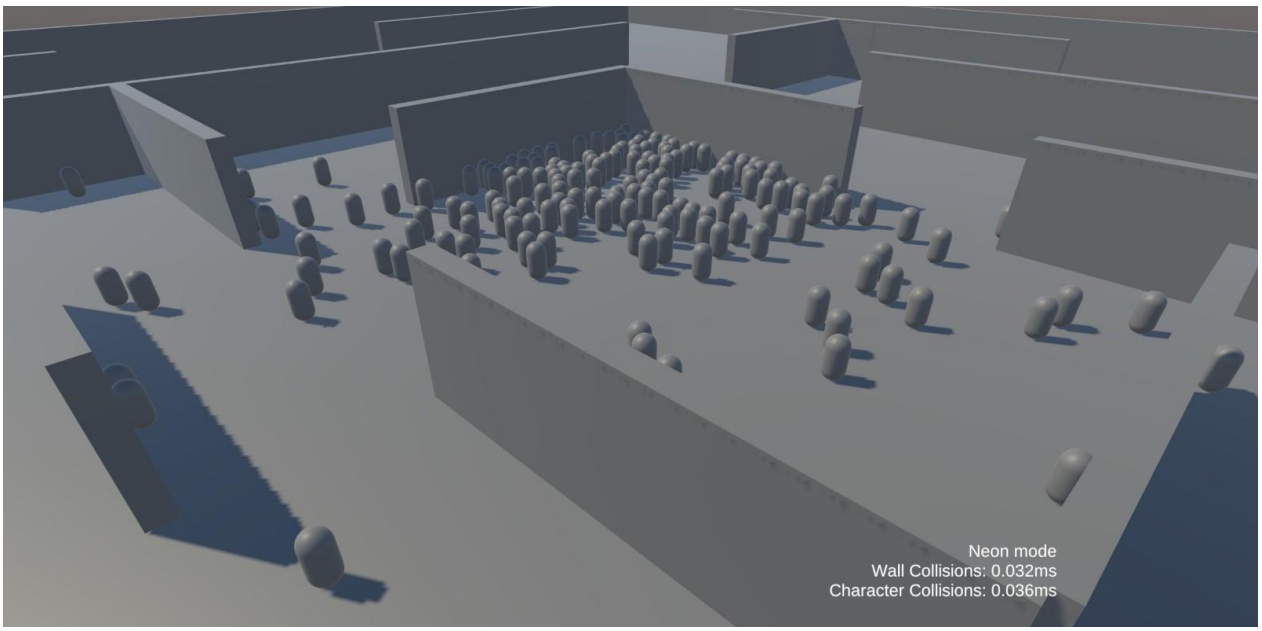
6 Example: Initial implementation

This section of the guide analyses three example implementations of collision detection code within Unity: a simple implementation, a Burst implementation, and a Neon implementation. Subsequent sections look at how we can optimize these implementations to improve performance.

Each code example implements two different types of collision detection algorithm:

- Axis Aligned Bounding Box (AABB) detection for collisions of characters with walls
- Radius-based detection for collisions between characters

The following image shows the application running, with characters and walls visible:



The code is available on the Unity Asset Store at: http://u3d.as/2yE3

The code described in the following section can be found in the script `CollisionCalculationScript.cs`.

6.1 Simple collision detection implementation

The simple collision detection implementation aims to achieve maximum performance using only standard C# code with no Burst attributes or Neon intrinsic instructions.

The following code shows a simple implementation of AABB collision detection:

```
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public bool Intersects(StaticCollisionObject other)
{
  return !(other.minPos.x > maxPos.x
        || other.maxPos.x < minPos.x
        || other.minPos.y > maxPos.y
        || other.maxPos.y < minPos.y);
```

```
}

...

    for (int c = 0; c < numChar; ++c)
    {
      for (int s = 0; s < staticObjects.Length; ++s)
      {
        staticCollisions[c * staticObjects.Length + s]
                = dynamicObjectsVstatic[c].Intersects(staticObjects[s]);
      }
    }
```

6.2 Burst Job collision detection implementation

The Burst Job collision detection example builds on the simple example but adds Burst constructs that provide additional information about the code to the compiler.

The following code shows the Burst Job implementation of AABB collision detection:

```
// Intersects function from previous Simple implementation
[BurstCompile]
public struct AABBObjCollisionDetectionJob : IJob
{
    [WriteOnly]
    public NativeArray<bool> collisions;
    [ReadOnly]
    public NativeArray<StaticCollisionObject> characters;
    [ReadOnly]
    public NativeArray<StaticCollisionObject> walls;

    public void Execute()
    {
        for (int c = 0; c < characters.Length; ++c)
        {
            for (int s = 0; s < walls.Length; ++s)
            {
                collisions[c * walls.Length + s] = characters[c].Intersects(walls[s]);
            }
        }
    }
}

...

      var staticJob = new AABBObjCollisionDetectionJob
      {
        characters = dynamicObjectsVstatic,
        walls = staticObjects,
        collisions = staticCollisions,
      };
      staticJob.Schedule().Complete();
```

6.3 Neon intrinsic collision detection implementation

The Neon intrinsic collision detection example builds on the previous Burst Job example by adding Neon intrinsics to the code, as well as the existing Burst constructs.

The following code shows the Neon intrinsic implementation of AABB collision detection:

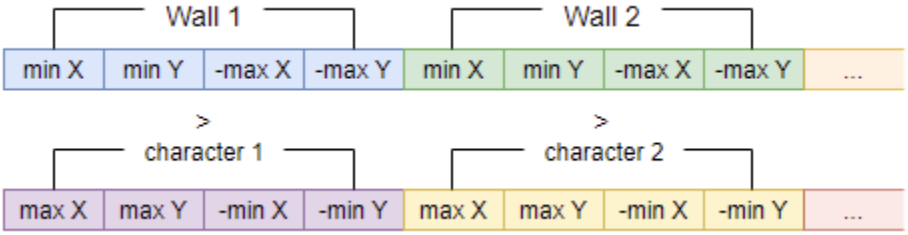
```
[BurstCompile]
public unsafe struct NeonAABBObjCollisionDetectionJob : IJob
{
    public NativeArray<bool> collisions;
    [ReadOnly]
    public int numCharacters;
    [ReadOnly]
    public int numWalls;
    [NativeDisableUnsafePtrRestriction]
    [ReadOnly]
    public float* walls;
    [NativeDisableUnsafePtrRestriction]
    [ReadOnly]
    public float* characters;

    public void Execute()
    {
        if (IsNeonSupported)
        {
            for (int c = 0; c < numCharacters; ++c)
            {
                var charLimits = vld1q_f32(characters + (4 * c));
                for (int w = 0; w < numWalls; ++w)
                {
                    var wallLimits = vld1q_f32(walls + (4 * w));
                    // we have data in 2 boxes:
                    // wall format [ min.x, min.y, -max.x, -max.y ] and character
                           format [max.x, max.y, -minx, -min.y]
                    collisions[c * numWalls + w] =
                           vmaxvq_u32(vcgeq_f32(wallLimits, charLimits)) == 0;
                }
            }
        }
    }
}

...

    fixed (float* wallFloats = staticFloats, charFloats = dynamicFloats)
    {
      var neonStaticJob = new NeonAABBObjCollisionDetectionJob
      {
        numCharacters = numChar,
        numWalls = staticObjects.Length,
        collisions = staticCollisions,
        walls = wallFloats,
        characters = charFloats
      };
      neonStaticJob.Schedule().Complete();
    }
```

7 Example: Initial analysis

The Neon implementation required careful planning of the initial data layout, to ensure efficient processing.

The two collision types are processed differently compared to the simple and Burst Job implementations. Radius-based collision x and y axis, and radius each are separated within an independent pipeline to be compared and processed. With AABB collisions, the order of the data is switched between walls and characters so that minimums are compared with maximums. Half of the AABB collision operation signs are also switched so that greater than comparisons are made.

After the order of the data and the signs have been switched, one Neon operation can process the four collisions in parallel to compare one wall against one character, as shown in the following diagram:



Timing is required to measure the optimizations of the collision implementations.

To time the optimizations, you can use the following:

- Unity Profiler
- `System.Diagnostics.Stopwatch`
- Profile Analyzer package

The Unity profiler shows timing for frames, which can vary significantly. The example used System.Diagnostics.Stopwatch for custom timing to display the ongoing frame average on screen. For normal profiling we recommend Profile Analyzer as this records averages over frames to get better values when measuring the optimization of the implementations.

Burst implementation results looked good initially on windows, while on Android the results were disappointing. The Android Neon and Burst implementations initially performed worse than the simple implementation. The differences in the results, show the importance of profiling on the device you are targeting.

The next section will analyze and improve the performance of the implementation to show you how to gain maximum performance with Neon.

8 Example: Initial optimization

For the Android version example, the IL2CPP backend, including clang, gave an automatic performance boost due to toolchain performance improvements including auto-vectorization producing Neon instructions. With the toolchain improvements and auto-vectorization, initial testing of the Android versions simple implementation provided favorable results compared to a faster Windows desktop.

However, the timing analysis done in Example: Initial analysis showed that both our Burst and Neon implementations are slower than the simple implementation. The following sections of this guide investigate why, and show how we can optimize performance.

The performance of the application is dominated by two major factors:

- The time taken to perform collision detection during each frame.
- The overhead incurred by using Burst Jobs to parallelize the workload.

The following sections examine each of these factors in turn.

8.1 Frame timing

The collision scenario for testing involves 59 walls with a single character spawning every 1.5 seconds until 121 characters are loaded. Every character model is tested against every wall and each other.

At the start of the scenario, there are fewer potential collisions due to the lower number of characters on screen. This lower initial number of collisions, means that we incur the overhead of vectorizing but do not get the performance benefits we would see with higher throughput. Burst also takes times to initialize the structures during the first few frames. To get timing data that is more representative of typical application usage, we can change the timing to reset after 60 frames or 2 minutes on Android so as to not count the early frames. During optimization, you often care about the worst-case scenario that makes your app stutter, not just the average frame time. Ignoring the early frames means we analyze timing during the peak workload period.

8.2 Burst Jobs overhead

Burst Jobs allow for asynchronous execution of code in parallel. The benefits of Burst Jobs mean you will likely want to use them when your app is complete. However, when trying to analyze timing performance, Burst Jobs add overhead that you must account for. Timing can be difficult as Burst jobs are not guaranteed to execute immediately, potentially causing incorrect timings.

Unity ProfilerMarker should be passed into Burst Jobs, enabling the code block to be profiled with Profile Analyzer. For more information on ProfilerMarker, see ProfilerMarker at Unity Documentation.

But Burst does not have to be used in Jobs. Instead static functions can be executed synchronously, allowing more accurate analysis of timing and performance improvements.

8.3 Using Burst without Jobs

To use Burst without Jobs, we must make several changes to the example Burst code implementation.

While Burst Jobs work well with the `NativeArray` datatype, static Burst functions are stricter in the rules for their parameters. `NativeArray` is a "handle" struct as defined in the Unity documentation, so passing it as a `ref` parameter would work. However, we converted the code to use `unsafe` pointers to arrays of `floats` instead because this is much closer to the Neon Burst implementation. The Neon Burst implementation also moved to returning a `bool` array.

The following code shows the example Burst without Jobs implementation using static Burst functions:

```
[BurstCompile]
[MethodImpl(MethodImplOptions.AggressiveInlining)]
static bool AABBIntersect(float wallMinX, float wallMinY, float wallMaxX,
       float wallMaxY, float charMinX, float charMinY, float charMaxX, float charMaxY)
{
  return !(charMinX > wallMaxX
        || charMaxX < wallMinX
        || charMinY > wallMaxY
        || charMaxY < wallMinY);
}


[BurstCompile]
static unsafe void AABBObjCollisionDetection(int numCharacters, int numWalls,
       [NoAlias] in float* walls, [NoAlias] in float* characters,
       [NoAlias] bool* collisions)
{
  for (int c = 0; c < numCharacters; ++c)
  {
    for (int w = 0; w < numWalls; ++w)
    {
      //we have 2boxes; wall format [ min.x, min.y, max.x, max.y ] and
             char format [max.x, max.y, minx, min.y]

      collisions[c * numWalls + w] = AABBIntersect(walls[4*w],
             walls[4*w+1], walls[4*w+2], walls[4 * w + 3],
             characters[4*c+2], characters[4 * c + 3], characters[4*c],
             characters[4*c+1]);
    }
  }
}

...
  fixed (bool* collisions = staticCollisions)
  {
    fixed (float* wallFloats = staticFloats, charFloats = dynamicFloats)
    {
      AABBObjCollisionDetection(numChar, staticObjects.Length, wallFloats,
             charFloats, collisions);
    }
  }
```

The `[NoAlias]` attribute is important to the Burst auto-vectorization. Without the `[NoAlias]` attribute, there is no improvement to the collision performance. Without this attribute, radius-based collisions show no improvement compared to the simple version and AABB collisions show a 4x performance reduction.

8.4 Timing results after initial optimization

Using Burst without Jobs, and ignoring the first 2 minutes of timing data shows a significant performance improvement.

The following table shows the Android timings relative to the simple collision detection execution time:

Collision type	Simple implementation	Burst without Jobs implementation	Neon implementation
AABB	1	0.17	0.38
Radius-based	1	0.36	0.68

This table shows that both Neon and Burst implementations are now much faster than the simple implementation. However, the new Burst without Jobs implementation is currently twice as fast as the Neon implementation. The next section looks at how we can make additional optimizations to the Neon implementation to increase performance even further.

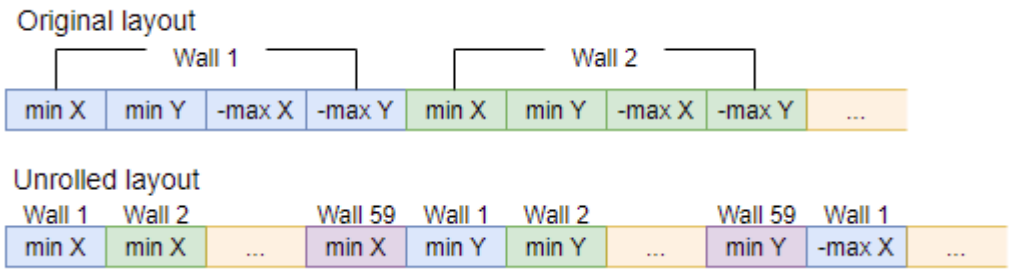
9 Example: Further Neon optimization

To further optimize performance, we can use loop unrolling. Unrolled loops allow more calculations to be done in parallel, and results can be extracted more efficiently.

For the AABB collisions, instead of putting the four collision conditions in one vector, we can split them and do comparisons with four walls at once, adding previous results together, and performing four collisions in each loop iteration. This loop unrolling provides a performance improvement of 16% for the AABB collisions.

Further loop unrolling for four characters against four walls, performing 16 collision detections in each loop iteration, produced no further performance improvement. Data fragmentation when storing the results cancels any improvement from the increased vectorization. This outcome emphasizes the importance of testing every optimization you make. Further loop unrolling might possibly have improved things in a much bigger scenario, or with guaranteed multiples of four walls and characters where tail handling code could be removed. As it is much bigger and harder to maintain, the more complex code is not worth keeping if it does not improve performance.

The following diagram compares the original layout of the position array data to the data used by the unrolled implementation:



For the Radius-based collisions, we are already performing four collision detection calculations simultaneously in each loop iteration in our simple implementation. In the optimized Neon implementation, we increase this to eight simultaneous calculations, and also optimize data extraction to be more efficient.

Burst 1.5 is missing the `vstr` set of Neon intrinsics for storing from the Neon vectors. Speed improvements are achieved by compressing eight 1-byte results into one vector and copying directly out into our boolean results array through a simple pointer cast:

```
*(Unity.Burst.Intrinsics.v64*)(collisions + (i * numChar + c-4)) = mask;
```

This gives a significant speed improvement. It shows how important it is to get data into and out of the Neon vectors efficiently, as well as keeping the calculations in vectors where possible.

Going back to the AABB collisions this optimization is added to compare against eight walls at once, gaining large speed improvements. Because the final simulation contains 236 walls, which is integer divisible by four, we can remove tail handling code for the AABB collisions as well.

9.1 Timing results after final optimizations

The following table shows the results from a final simulation that uses 2401 characters and 236 walls:

Collision type	Simple implementation	Burst without Jobs implementation	Neon implementation
AABB	1	0.17	0.10
Radius-based	1	0.36	0.29

After the optimizations, the collision detection has become a small part of the collision system, so any further optimization effort needed is best focused elsewhere. Burst has given a 2.75x speed increase for Radius-based and a 6x speed increase for AABB collisions. Utilizing Neon has given a further speed increase of 21% for Radius-based and a 40% AABB speed increase compared to Burst.

10 Example: Final implementation code

The following code shows the final Neon Burst implementation of AABB collision detection:

```
/// <summary>
    /// Final working Neon Burst implementation of AABB collision detection
    /// </summary>
    /// <param name="numCharacters">number of characters active in the scene</param>
    /// <param name="numWalls">number of walls in the scene</param>
    /// <param name="walls">array of wall positions - all walls min.x, then all min.y, then all -max.x, then all -max.y</param>
    /// <param name="characters">array of charcter positions - all characters max.x, then all max.y, then all -min.x, then all -min.y</param>
    /// <param name="collisions">returned bool array of which characters collide with which walls</param>
    [BurstCompile]
    static unsafe void NeonAABBObjCollisionDetectionUnrolled(int numCharacters, int numWalls, [NoAlias] in float* walls, [NoAlias] in float* characters, [NoAlias] bool* collisions)
    {
        if (IsNeonSupported)
        {
            var tblindex1 = new Unity.Burst.Intrinsics.v64((byte)0, 4, 8, 12, 255, 255, 255, 255);//255=> out of range index will give 0
            var tblindex2 = new Unity.Burst.Intrinsics.v64((byte)255, 255, 255, 255, 0, 4, 8, 12);


            int c = 0;
            for (; c < numCharacters; ++c)
            {
                var charMaxXs = vdupq_n_f32(*(characters + c));
                var charMaxYs = vdupq_n_f32(*(characters + numCharacters + c));
                var charMinusMinXs = vdupq_n_f32(*(characters + 2*numCharacters + c));
                var charMinusMinYs = vdupq_n_f32(*(characters + 3*numCharacters + c));


                int w = 0;
                for (; w < (numWalls&~7); w+=4)
                {
                   var wallMinXs = vld1q_f32(walls + w);
```

```
                    var wallMinYs = vld1q_f32(walls + numWalls + w);
                    var wallMinusMaxXs = vld1q_f32(walls + 2*numWalls + w);
                    var wallMinusMaxYs = vld1q_f32(walls + 3*numWalls + w);

                    var results = vqtbl1_u8(vorrq_u32(vorrq_u32(vcgeq_f32(wallMinXs,
charMaxXs), vcgeq_f32(wallMinYs, charMaxYs)),
                                                vorrq_u32(vcgeq_f32(wallMinusMaxXs,
charMinusMinXs), vcgeq_f32(wallMinusMaxYs, charMinusMinYs))), tblindex1);

                    w += 4;

                    wallMinXs = vld1q_f32(walls + w);
                    wallMinYs = vld1q_f32(walls + numWalls + w);
                    wallMinusMaxXs = vld1q_f32(walls + 2 * numWalls + w);
                    wallMinusMaxYs = vld1q_f32(walls + 3 * numWalls + w);

                    results = vmvn_u8(vqtbx1_u8(results,
vorrq_u32(vorrq_u32(vcgeq_f32(wallMinXs, charMaxXs), vcgeq_f32(wallMinYs, charMaxYs)),
                                                vorrq_u32(vcgeq_f32(wallMinusMaxXs,
charMinusMinXs), vcgeq_f32(wallMinusMaxYs, charMinusMinYs))), tblindex2));

                    *(Unity.Burst.Intrinsics.v64*)(collisions + (c * numWalls + w - 4))
= results; //straight into array
                }
                if (w+3<numWalls)
                {
                    var wallMinXs = vld1q_f32(walls + w);
                    var wallMinYs = vld1q_f32(walls + numWalls + w);
                    var wallMinusMaxXs = vld1q_f32(walls + 2 * numWalls + w);
                    var wallMinusMaxYs = vld1q_f32(walls + 3 * numWalls + w);

                    var results = vmovn_u32(vorrq_u32(vorrq_u32(vcgeq_f32(wallMinXs,
charMaxXs), vcgeq_f32(wallMinYs, charMaxYs)),
                                                vorrq_u32(vcgeq_f32(wallMinusMaxXs,
charMinusMinXs), vcgeq_f32(wallMinusMaxYs, charMinusMinYs))));//4x16bit answers after
narrow
                    results = vmvn_u8(vuzp1_u8(results, results));//4x8bit answers,
then the same answers again after unzipping; then negated

                    *(uint*)(collisions + c * numWalls + w) = vget_lane_u32(results,
0);//put all 4 in at once
                }
```

```
                }
            }
        }
```

For a full view of the code, access the Unity Asset Store at: http://u3d.as/2yE3

11 Related information

Here are some recommended guides, available on Arm Developer:

- Introducing Neon for Armv8-A
- Compiling for Neon with Auto-Vectorization
- Optimizing C code with Neon intrinsics
- Neon programmers guide for Armv8-A: Coding for Neon

Here are some resources related to the material in this guide:

- Intrinsics
- Unity intrinsics
- IL2CPP Unity Manual
- Unity optimization guidelines - Aliasing
- Unity unity.mathematics library - GitHub repository
- Unity Burst Inspector quick start guide
- Burst Documentation – DllImport and internal calls

12 Next steps

During the implementation of Neon intrinsics within your Unity Application, make sure:

- Data is loaded and extracted efficiently.
- Keep the data in Neon vectors.
- Use as much of the vector as possible.
- Use as much parallelism as possible.

To gain more advantages from implementing Neon, focus on the structure of the data and the code. Optimal data and code structures will usually bring a greater performance increase than using Neon intrinsics to code an inefficient algorithm or work with disorganized data layouts.