



Neon C# intrinsics in .NET 5 and 6

Non-Confidential

Copyright © 2021 Arm Limited (or its affiliates).
All rights reserved.

Issue 1.0

102753

Neon C# intrinsics in .NET 5 and 6

Copyright © 2021 Arm Limited (or its affiliates). All rights reserved.

Release information

Document history

Issue	Date	Confidentiality	Change
1.0	07 December 2021	Non-Confidential	

Non-Confidential Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

If any of the provisions contained in these terms conflict with any of the provisions of any click through or signed written agreement covering this document with Arm, then the click through or signed written agreement prevails over and supersedes the conflicting provisions of these terms. This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm's trademark usage guidelines at <http://www.arm.com/company/policies/trademarks>.

Copyright © 2021 Arm Limited (or its affiliates). All rights reserved.

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.

Product Status

The information in this document is Final, that is for a developed product.

Web Address

www.arm.com

Contents

1 Overview	5
2 What are Neon intrinsics?	6
3 How to use .NET intrinsics.....	8
4 Neon intrinsics examples.....	9
4.1 Weighted average	9
4.2 Matrix-Vector multiply.....	11
4.3 1D convolution.....	13
5 Performance gains advice	15
6 Related information	16
7 Next Steps.....	17
8 Appendix.....	18
8.1 Example 1: Weighted mean	18
8.2 Example 2: Matrix-Vector multiply.....	19
8.3 Example 3: 1D convolution	20

1 Overview

In this guide, you will learn how to use Neon C# intrinsics in .NET framework 5 and 6.

Microsoft introduced Arm64 hardware intrinsics in .NET 5, and this debut creates the opportunity to make your .NET C# code faster on Windows on Arm devices. The intrinsics also work in some other .NET languages, and with other .NET-supported operating systems with appropriate hardware.

Intrinsics were introduced to C/C++ code a while ago, but it is recently become a possibility in C# code. The instructions fit well into the language and give some good opportunities for performance gains, without the pain of going to assembly language. There is no need for complicated tasks like tracking registers, the intrinsics are functions that fit well in your normal code, and they are easier to maintain. Neon intrinsics are not quite as fast as assembly, but they are a lot easier to read and write. However, as compilers like clang have become very good at auto-vectorization, for C/C++ you now must profile carefully and make sure your Neon intrinsics code is faster than what the compiler does automatically. Significant speed gains are expected because you know your code better than the compiler, but only with careful programming.

[Unity has added Neon intrinsics](#) to their C# when targeting Android with their burst compiler. Burst knows a lot about the code and pre-compiles for a specific platform. Also, they can auto-vectorize a lot, and again, careful programming is needed on your performance-critical code sections to get significant performance improvements over the compiler.

Net C# is bytecode and aimed at running on different processors. This scenario needs a [Just-in-Time compiling \(JIT\)](#) compiler. The JIT (compiling just before running) cannot do complicated things. Auto-vectorization of Single Instruction, Multiple Data (SIMD) (Neon being our SIMD example) is complicated, so there is no auto-vectorization. This lack of auto-vectorization means it becomes worthwhile to implement some Neon when your code is likely to run on Arm devices.

2 What are Neon intrinsics?

Neon intrinsics in .NET let you write commands in their C# code that map directly to specific Arm native instructions. The Neon intrinsics are a way to write assembly instructions, without the detail and difficulty of coding in assembly. The Neon set of instructions are 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. The instruction then adds the result to any existing value in the corresponding element of the destination register:

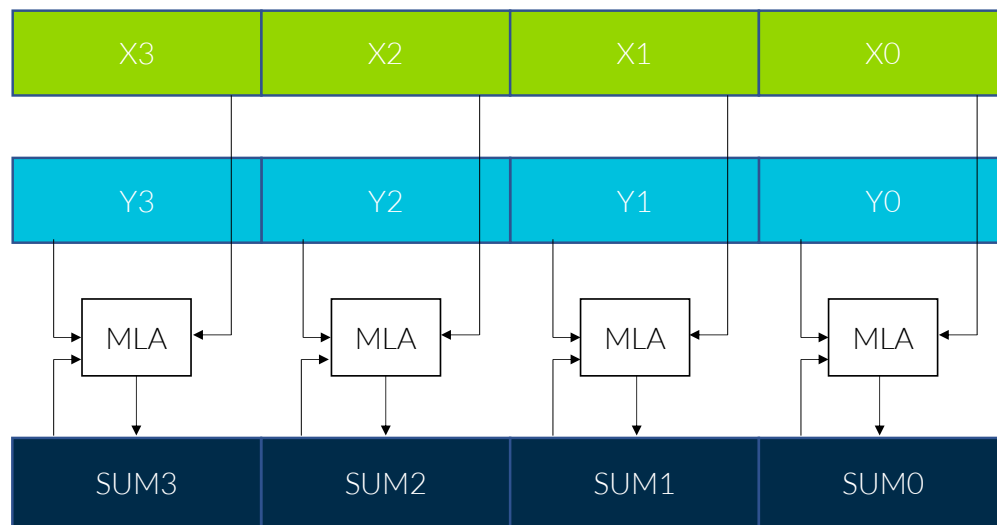


Figure 1: MLA arithmetic operations

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

- $\text{sum0} += \text{x0} * \text{y0}$
- $\text{sum1} += \text{x1} * \text{y1}$
- $\text{sum2} += \text{x2} * \text{y2}$
- $\text{sum3} += \text{x3} * \text{y3}$

The Neon instructions can operate on either 64-bit or 128-bit wide vectors, with individual data elements of 8, 16, 32-bits or 64-bits. It is 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 16 Bytes.

To learn about all the operations that are available with Neon intrinsics, see [Intrinsics](#).

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 you use these assembly instructions in a high-level language without the overhead of writing assembly code by hand. The compiler takes care of register allocation and other details. Neon intrinsics were first used in C and C++, but Microsoft has now added the intrinsics into .NET for use in C# code.

Microsoft has implemented most of the Arm v8.0 Neon instructions. There are a couple of later Arm v8.x extensions, including dot product, which have their own separate classes. Intrinsics that require multiple vectors to be sequential in memory, and intrinsics that can be composed from other intrinsics have not been implemented. In the examples below we will look at ways around these limitations. The `bfloat` type that is becoming popular in Machine Learning (ML) is also unsupported. The full list of supported intrinsics in .NET 5 and 6 is available in the [Microsoft .NET documentation](#).

3 How to use .NET intrinsics

When using the Arm Neon intrinsics in .NET, the first thing is to find them with the appropriate namespace, they are all under `System.Runtime.Intrinsics.Arm`.

Microsoft has separated sets of intrinsics by class, with extensions separate from the main body of instructions. Those instructions only available in Arm64 are separated from intrinsics available in both 32-bit and 64-bit Arm. Note that with Windows 10 or 11 on Arm all devices are 64-bit, and both sets of instructions are available. As examples, the main body of instructions is in the classes are:

- `System.Runtime.Intrinsics.Arm.AdvSimd`
(For instructions available in both 32- and 64-bit)
- `System.Runtime.Intrinsics.Arm.AdvSimd.Arm64`
(For instructions only on 64-bit Arm)

The dot product extension is in the class of `System.Runtime.Intrinsics.Arm.Dp`.

The names of the intrinsics functions are not a straight map to the Arm C-Language Extension (ACLE) intrinsics. There is a consistent formula to map them to more plain language names, and the Microsoft documentation says which ACLE intrinsic it is implementing: For example, `AdvSimd.Arm64.AddPairwise` (`Vector128<UInt16>`, `Vector128<UInt16>`) comes from the ACLE intrinsic `vpaddq_u16` and the assembly instruction `ADDP`.

When using .NET intrinsics, the standard types passed for 16- and 8-byte vectors respectively are:

- `System.Runtime.Intrinsics.Vector128<>`
- `System.Runtime.Intrinsics.Vector64<>`

Most methods in these classes are designed to keep data in the intrinsics, but there are also methods like `ToScalar()` bring the data back into standard .NET types. Over time these classes will have methods like `Sum()` added to do some intrinsics work automatically, without needing separate code per architecture.

When adding intrinsics, you must check their support, with each class having an `IsSupported` property. For example, the standard Neon intrinsics are checked with:

- `System.Runtime.Intrinsics.Arm.AdvSimd.IsSupported`
- `System.Runtime.Intrinsics.Arm.AdvSimd.Arm64.IsSupported`

If using dot product, you also must check `System.Runtime.Intrinsics.Arm.Dp.IsSupported`. fallback code can then be written for non-Arm devices, or those devices that do not support specific extensions.

After checking for intrinsics support, what code is needed depends on the task that needs performance improvement. Intrinsics work involves significant profiling and more maintenance overhead than traditional code, so focus on the areas where it provides most benefit.

4 Neon intrinsics examples

Let us look at three examples from the domain of ML. The final code for each of the three examples is in the [Appendix](#).

4.1 Example 1: Weighted average

The basics of a neural network involve taking a weighted average of the activation of neurons in the layer before. This weighted average is something that can be parallelized easily, and therefore is ideal for SIMD intrinsic commands. For a C/C++ compiler, this process would be easily auto-vectorized, but with .NET we must do the vectorization into intrinsics ourselves.

A following fragment code shows weighted mean code before vectorization:

```
float sum = 0;
for (uint i=0; i<count; ++i)
{
    sum += inputs[i] * weights[i];
}
return sum/count;
```

The previous code is very simple. With Neon SIMD commands we can sum four weighted values at once, instead of one, which gives us a nearly 4x speed improvement. There is a small amount of overhead, but if there are four inputs to average, we have a significant improvement. We converge to 4x, as the number of values gets larger. From experiments, 1023 values give a 3.98x improvement and 2047 values give a 3.99x improvement.

Let us go through how to do it. The full final code is in the [Appendix](#).

First, we need an `unsafe` function to pass pointer values into, so they can be loaded into the `Vector<>` classes.

```
public static unsafe float WeightedMean(float* inputs, float* weights, uint count)
```

This presumes `inputs` and `weights` are arrays of `floats`. This function is called with the `fixed` keyword, which is used to extract the needed pointers from the .NET objects (and stop the garbage collector from moving them):

```
fixed (float* inptr = inputs, weighptr = weights)
{ ArmMLImplementations.WeightedMean(inptr, weighptr, count); }
```

This example raises the first point about good use of intrinsics, how we store our data. In this case, it is simple, an array of floats. There is thought needed when using more complicated structures about the easiest way to get that data into and out of vectors. Often, we move from an array of structures to a structure of arrays, so that each of those arrays can be loaded and stored easily.

Once we have the data in the function in pointers, we can load it and do the multiplication.

The following fragment code illustrates the core weighted mean loop with Neon intrinsics:

```
var total = Vector128<float>.Zero;
uint i = 0;
for (; i < vec4length; i+=4)
{
    var inpVec = AdvSimd.LoadVector128(inputs + i);
    var weighVec = AdvSimd.LoadVector128(weights + i);
    total = AdvSimd.FusedMultiplyAdd(total, inpVec, weighVec);
}
```

In this code example, we go through the array four at a time. Loading the inputs and the weights, and then multiplying them together and adding them to a vectorized total. As we go through, we end up with four values in a vector that still need summing up, and we still must average. So how do we get the data back out of the `Vector128<>`?

The most efficient result returned is to have four (or eight) outputs from the vector return into a pointer. Our three examples all end up with a single number, so we must sum our vector. As `Sum()` has not been added to .NET yet, we need an intrinsic that does the sum for us. For integer types `Vector128<>` has an `AddAcross()` method, but the `vaddvq_f32` ACLE intrinsic does not have a corresponding float `AddAcross()`. If we look closer at the `vaddvq_f32` intrinsic we discover that it is composed of two pairwise add operations. Indeed, we achieve the result we want by making these two calls.

The following fragment code shows how to get a single value out of a vector:

```
var total64 = AdvSimd.Arm64.AddPairwise(total, total).GetLower();
var sum = AdvSimd.Arm64.AddPairwiseScalar(total64).ToScalar();
```

The first `AddPairwise()` adds neighboring floats, and two `Vector128<>` values are then reduced to one. We then call `GetLower()` to get a `Vector64<>` of the bottom two values in the vector (meaning `AddPairwise()`'s first `Vector128<>` parameter is effectively ignored). The second `AddPairwiseScalar()` adds the two float values within the resultant `Vector64<>` to get our result. But it is only with the `ToScalar()` method call, that we move the result out of SIMD registers and back into a normal .NET C# float.

This method results in a single value, but it also only adds groups of four numbers, what happens if I have two or three numbers left over? For this case we need so-called tail code, where we add on the remaining values that did not fit our vectorization.

First, we must work out only going through groups of four to begin with. For the previous case, we work out `vec4length = count & ~3`; this calculation rounds us down to the nearest multiple of four. Then we add the tail code around the `sum` statement in the next code sample.

The following fragment code shows the tail code for the weighted mean method:

```
uint tail = count - vec4length; // how many left over after main loop
if (tail >= 2)
{
    var inpVec = AdvSimd.LoadVector64(inputs + i);
    var weighVec = AdvSimd.LoadVector64(weights + i);
    total64 = AdvSimd.FusedMultiplyAdd(total64, inpVec, weighVec);
    i += 2;
}
var sum = AdvSimd.Arm64.AddPairwiseScalar(total64).ToScalar();
if ((tail & 1) == (uint)1)
{
    sum += inputs[i] * weights[i];
}
return sum/count;
```

We can still use intrinsics on a `Vector64<>` for two left over, but the last one is done the old scalar way.

This method is a simple weighted mean, where we know our numbers are not going to get too large. If we are concerned we would have an overflow, there are intrinsics to do the multiply-add from 32-bit float inputs to a 64-bit float output. Widening and narrowing of float and integer precisions between 8-bit to 64-bit is part of many operations. There are more advanced optimizations that can go along with them.

One more advanced optimization that can work well alongside both widening or narrowing and in our simple case is loop unrolling. If we go through eight values at a time, instead of four, we can get a further speedup as two SIMD registers can get utilized efficiently in sequence. We need more tail code to cope with the case of up to seven left over after the main loop. In this case, we get more than 30% speedup compared to looping through four at a time. This speedup means we compute our weighted mean 5.75x faster than traditional scalar code that is, in nearly 1/6 the time. Further improvements in this area are possible but profiling and measuring outcomes are important. If you have assembly knowledge, you can look at the [Arm Cortex-X1 core software optimization guide](#) to see the benefits of sequences of commands can work well together.

4.2 Example 2: Matrix-Vector multiply

More complicated problems often offer bigger performance improvements. Bear in mind that often with more complicated setups there is some more overhead, so more data is needed to get the benefits.

Our second example is a matrix-vector multiply. With an initial implementation before unrolling we get a 3.5x speedup for an 8x8 matrix, 6.7x speedup for a 24x24 matrix and a 7.6x speedup for a 36x36 matrix. Unrolling added 15% more performance, meaning the speed-up was from 4.1x faster for 8x8 matrices and 8.9x faster for 36x36. Larger matrices would no doubt have even better performance gain.

The following fragment code shows the main loop of the matrix-vector multiply after unrolling:

```
var zeroVec = Vector128<float>.Zero;
for (uint i = 0; i < numRows; ++i)
{
    var tempVec = zeroVec;
    uint j = 0;
    for (; j < vec8length; j+=8)
    {
        var matVec = AdvSimd.LoadVector128(matrix + i*numCol + j);
        var matVec1 = AdvSimd.LoadVector128(matrix + i * numCol + j+4);
        var vecVec = AdvSimd.LoadVector128(vector + j);
        var vecVec1 = AdvSimd.LoadVector128(vector + j+4);
        tempVec = AdvSimd.FusedMultiplyAdd(tempVec, matVec, vecVec);
        tempVec = AdvSimd.FusedMultiplyAdd(tempVec, matVec1, vecVec1);
    }
    // tail code
}
```

The matrix-vector multiply involves a nested loop, and we must consider the best way to access the data sequentially. There is now tail code at the end of each vector.

In this example data setup is not too complicated. We want to access the matrix data sequentially along a row, and the vector is a single array. The matrix data can be stored in a single array with each row after the other and accessed efficiently. For a matrix-matrix multiply, the second matrix would want to be accessed sequentially along columns. Therefore, consider how the data is stored, transposed, or loaded in a different order. Then profile to see what works best.

Profiling is needed to measure any optimization. In this example creating a zero vector outside the loop ended up creating a 5% time improvement on larger matrices. 8x8 matrices were unaffected. Moving other initializations out of their loops resulted in a very small speed loss, as the loss of flexibility around register uses hampered performance.

Another optimization to apply is to encourage the JIT compiler to do inlining for your functions. `[MethodImpl(MethodImplOptions.AggressiveInlining)]` can give significant improvement to both normal and Neon code. For our matrix case, it improves the Neon more, so 8x8 matrices are now 5.8x faster than non-Neon code. The 36x36 matrices are 9.6x faster than inline non-Neon code, and just under 10x faster than the code we started with.

4.3 Example 3: 1D convolution

The third example is a 1D convolution. The speed-ups were harder here, but still 2.5x or more.

There is more data preparation required as we wanted to reverse the kernel used in the convolution so we can multiply in reverse order.

The following fragment code shows the basic 1D convolution logic:

```
float total = 0;
for (uint j = 0; j < sizeKernel; ++j)
{
    total += signal[i + j] * kernel[sizeKernel - 1 - j];
}
result[i] = total;
```

If you are unfamiliar with convolution mathematics, we multiply an incoming signal at each point by a kernel that it slides past. The section of kernel and signal that overlaps are added for a new outgoing signal for each point, producing a new sequence that is the combination of the two.

The values of the kernel end up used in reverse, so for data preparation we first reverse the kernel vector.

The following fragment code shows the reversing of the kernel, loading the start before the main loop, and multiplying the start with it before any other calculation:

```
var kernelReverse = new float[sizeKernel>4?sizeKernel:4]; //initialized to 0
for (int k = 0; k < sizeKernel; ++k)
{
    kernelReverse[k] = kernel[sizeKernel - k - 1];
}

fixed (float* kernelPtr = kernelReverse)
{
    Vector128<float> kernelReversel = AdvSimd.LoadVector128(kernelPtr);
    for (uint i = kernelPad; i < sizeSignal - kernelPad; ++i)
    {
        var signal1 = AdvSimd.LoadVector128(signal + i - kernelPad);
        var tempVec = AdvSimd.Multiply(signal1, kernelReversel);
```

Many optimizations require you to know your problem. In this instance, we know kernels are usually three, five, or seven long. With this knowledge, we can specialize to make the first four values of the reversed kernel more efficient. We store them in a separate Neon vector, permanently loaded, to do our first multiplication, before we work out the rest as needed. With a kernel smaller than four (that is

three as the only sensible small size) the fourth value is set to zero and does not affect the calculation. For a kernel of three this is all the calculation, and we just store the result.

We also make the function suit our needs. When using convolutions in ML, it is usually for image processing and a line of pixels is used. This line is either padded by the kernel size to return the same length line, or the returned line is smaller. Meaning, we do not have to worry about the edge case where the kernel and signal do not quite line up. We start from the point where they do. Not having these edge cases makes the algorithm simpler and easier to vectorize.

Once again, we see the bigger the volume, the better the savings on these slightly more complicated cases, as we must account for some setup cost. Here we see for both five and seven value kernels we get a 2.5x speed-up for a 256 pixel line, and a 2.7x speed-up for a 1024 pixel line. The three value kernel, which does not go through the tail code, gets a 2.7x speedup with 256 pixels and 3x speedup on 1024 pixels. Therefore, it executes in 1/3 the time.

5 Performance gains advice

These three examples are simple cases, and more complicated ones (like the physics collisions examined for [Unity C# Neon intrinsics](#)) have the potential for greater benefits. There is a cost to getting data into and out of Neon registers. The more you can do while the data is in those registers, and the more parallel work you can do, the greater the gains you can reap.

Given that .NET JIT has no auto-vectorization, any parallel work you do must result in some gains. But following the next advice can help you maximize the performance improvement.

The main SIMD principles for best gain from Neon intrinsics are:

- Groom the data into arrays that make for efficient loading for the operations you want to perform. Often structures of arrays instead of arrays of structures.
- Do as much as possible, and as parallel as possible, once you have loaded into Neon vectors. Do not move back and forth out of Neon (therefore only use `ToScalar()` or similar when needed).
- Try loop unrolling for often an easy extra gain.
- Extract the data out efficiently – if you can have more than one value calculated to store at once, that is better.
- Profile. It is only by measuring you can be sure that performance is improved.

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

6 Related information

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

- [Arm intrinsics](#)
- [Introducing Neon for Armv8-A](#)
- [Microsoft .NET API documentation: System.Runtime.Intrinsics.Arm](#)
- [Neon programmers guide for Armv8-A: Coding for Neon](#)
- [Optimizing C code with Neon intrinsics](#)
- [Using Neon C# intrinsics with Unity burst](#)

7 Next Steps

This guide has introduced you the Neon intrinsics and how to use Arm Neon intrinsics in .NET. You learned about ML of the Neon intrinsics through three examples.

To learn more about .Net intrinsics implementation, follow the links to the Microsoft documentation site:

- [ARM64 Performance in .NET 5](#)
- [System.Runtime.Intrinsics.Arm Namespace](#)

8 Appendix

The final version of .NET Neon C# code for the 3 examples are the following:

- [Example 1: Weighted mean](#)
- [Example 2: Matrix-Vector multiply](#)
- [Example 3: 1D convolution](#)

8.1 Example 1: Weighted mean

```
/// <summary>
/// Weighted mean of inputs
/// </summary>
/// <param name="inputs">inputs to be averaged</param>
/// <param name="weights">weights for each input</param>
/// <param name="count">number of inputs</param>
/// <returns>weighted mean</returns>
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static unsafe float WeightedMean(float* inputs, float* weights, uint count)
{
    uint vec8length = count & ~(uint)7;
    uint tail = count - vec8length;

    var total = Vector128<float>.Zero;
    uint i = 0;
    for (; i < vec8length; i+=8)
    {
        var inpVec = AdvSimd.LoadVector128(inputs + i);
        var inpVec1 = AdvSimd.LoadVector128(inputs + i+4);
        var weighVec = AdvSimd.LoadVector128(weights + i);
        var weighVec1 = AdvSimd.LoadVector128(weights + i+4);
        total = AdvSimd.FusedMultiplyAdd(total, inpVec, weighVec);
        total = AdvSimd.FusedMultiplyAdd(total, inpVec1, weighVec1);
    }
    // tail code
    if (tail >= 4)
    {
        var inpVec = AdvSimd.LoadVector128(inputs + i);
        var weighVec = AdvSimd.LoadVector128(weights + i);
        total = AdvSimd.FusedMultiplyAdd(total, inpVec, weighVec);
    }
}
```

```

        i += 4; tail -= 4;
    }
    var total64 = AdvSimd.Arm64.AddPairwise(total, total).GetLower();
    if (tail >= 2)
    {
        var inpVec = AdvSimd.LoadVector64(inputs + i);
        var weighVec = AdvSimd.LoadVector64(weights + i);
        total64 = AdvSimd.FusedMultiplyAdd(total64, inpVec, weighVec);
        i += 2;
    }
    var sum = AdvSimd.Arm64.AddPairwiseScalar(total64).ToScalar();
    if ((tail & 1) == (uint)1)
    {
        sum += inputs[i] * weights[i];
    }
    return sum/count;
}

```

8.2 Example 2: Matrix-Vector multiply

```

/// <summary>
/// Matrix Multiply function using Neon
/// </summary>
/// <param name="matrix">matrix to multiply</param>
/// <param name="vector">vector to multiply</param>
/// <param name="numRow">number of rows in matrix</param>
/// <param name="numCol">number of columns in matrix, and length of vector</param>
/// <param name="result">matrix times vector (assumes correct size allocated)</param>
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static unsafe void MatMul(float* matrix, float* vector, uint numRows, uint
numCol, float* result)
{
    uint vec8length = numCol & ~(uint)7;
    uint tail = numCol - vec8length;

    var zeroVec = Vector128<float>.Zero;
    for (uint i = 0; i < numRows; ++i)
    {
        var tempVec = zeroVec;
        uint j = 0;
        for (; j < vec8length; j+=8)

```

```

{
    var matVec = AdvSimd.LoadVector128(matrix + i*numCol + j);
    var matVec1 = AdvSimd.LoadVector128(matrix + i * numCol + j+4);
    var vecVec = AdvSimd.LoadVector128(vector + j);
    var vecVec1 = AdvSimd.LoadVector128(vector + j+4);
    tempVec = AdvSimd.FusedMultiplyAdd(tempVec, matVec, vecVec);
    tempVec = AdvSimd.FusedMultiplyAdd(tempVec, matVec1, vecVec1);
}
if (tail >= 4)
{
    var matVec = AdvSimd.LoadVector128(matrix + i * numCol + j);
    var vecVec = AdvSimd.LoadVector128(vector + j);
    tempVec = AdvSimd.FusedMultiplyAdd(tempVec, matVec, vecVec);
    j += 4;
}
var firstStep = AdvSimd.Arm64.AddPairwise(tempVec, tempVec).GetLower();
// tail code
if ((tail&3)>=2)
{
    var matVec = AdvSimd.LoadVector64(matrix + i * numCol + j);
    var vecVec = AdvSimd.LoadVector64(vector + j);
    j += 2;
    firstStep = AdvSimd.Arm64.AddPairwiseScalar(AdvSimd.FusedMultiplyAdd(firstStep,
matVec, vecVec));
}
result[i] = AdvSimd.Arm64.AddPairwiseScalar(firstStep).ToScalar();
if ((tail&1)==(uint)1)
{
    result[i] += vector[j] * matrix[i * numCol + j];
}
}
}

```

8.3 Example 3: 1D convolution

```

/// <summary>
/// 1D convolution using Neon on a pixel line
/// </summary>
/// <param name="signal">pixel line (including padding)</param>
/// <param name="kernel">kernel to convolve with</param>

```

```

/// <param name="sizeSignal">length of pixel line (signal array) - presumed to be zero
padded for returning if want same length</param>
/// <param name="sizeKernel">length of kernel array</param>
/// <param name="result">output of convolution on pixel line - assumes correct amount
of space is allocated</param>
/// <returns>size of data inserted into result</returns>
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static unsafe uint Convolution1D(float* signal, float* kernel, uint sizeSignal,
uint sizeKernel, float* result)
{
    uint kernelPad = sizeKernel / 2;
    uint returnSize = sizeSignal - kernelPad * 2;

    uint kernel4length = sizeKernel & ~(uint)3;
    uint kerneltail = sizeKernel > 4 ? sizeKernel - kernel4length : 0;

    var kernelReverse = new float[sizeKernel > 4 ? sizeKernel : 4]; //initialized to 0
    for (int k = 0; k < sizeKernel; ++k)
    {
        kernelReverse[k] = kernel[sizeKernel - k - 1];
    }

    fixed (float* kernelPtr = kernelReverse)
    {
        Vector128<float> kernelReversel = AdvSimd.LoadVector128(kernelPtr);
        for (uint i = kernelPad; i < sizeSignal - kernelPad; ++i)
        {
            var signal1 = AdvSimd.LoadVector128(signal + i - kernelPad);
            var tempVec = AdvSimd.Multiply(signal1, kernelReversel);

            uint j = 4;
            for (; j < kernel4length; j += 4)
            {
                var signalVec = AdvSimd.LoadVector128(signal + i + j - kernelPad);
                var kernelVec = AdvSimd.LoadVector128(kernelPtr + j);
                tempVec = AdvSimd.FusedMultiplyAdd(tempVec, signalVec, kernelVec);
            }
            var firstStep = AdvSimd.Arm64.AddPairwise(tempVec, tempVec).GetLower();
            // tail code
            if (kerneltail >= 2)
            {
                var signalVec = AdvSimd.LoadVector64(signal + i + j - kernelPad);

```

Copyright © 2021 Arm Limited (or its affiliates). All rights reserved.

Non-Confidential

```
        var kernelVec = AdvSimd.LoadVector64(kernelPtr + j);
        j += 2;
        firstStep = AdvSimd.Arm64.AddPairwiseScalar(AdvSimd.FusedMultiplyAdd(firstStep,
signalVec, kernelVec));
    }
    result[i - kernelPad] = AdvSimd.Arm64.AddPairwiseScalar(firstStep).ToScalar();
    if ((kerneltail & 1) == (uint)1)
    {
        result[i - kernelPad] += signal[i + j - kernelPad] * kernelPtr[j];
    }

    }
}
return returnSize
```

