# Intro to SIMD in .NET

# What we'll cover

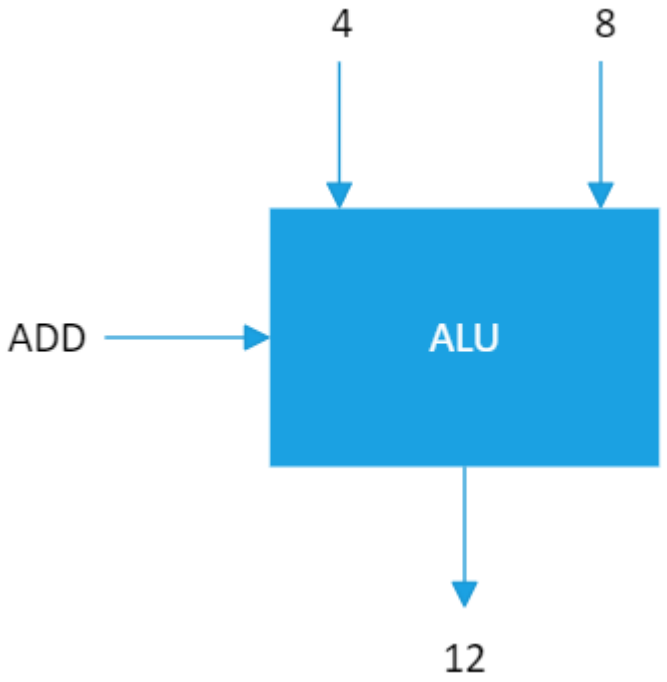
- What is SIMD
- Applications
- Overview of SIMD Implementations
- SIMD in .NET
- Examples and benchmarks
- Disadvantages
- Resources

# What is SIMD

- SIMD = Single Instruction Multiple Data
- Provides parallelism on a single CPU core on modern processors
- Allows a single instruction to operate on multiple input values simultaneously
- Can improve performance when instructions can be “vectorized” effectively

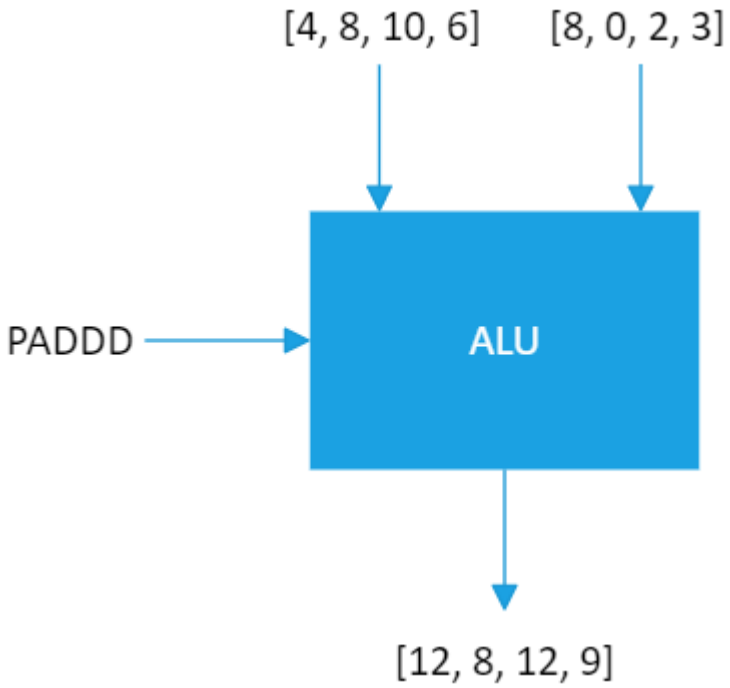
# What is SIMD (cont'd)

- Single Instruction Single Data:



# What is SIMD (cont'd)

- Single Instruction Multiple Data:



# Applications

- Numerical data processing (e.g. numpy)
- Computer Graphics, Image Processing, Gaming
- Scientific Computing
- Machine Learning (e.g. scikit-learn, ML.NET)
- JSON parsing? (e.g. simdjson)

# SIMD Implementation

- There are different SIMD implementations (e.g. SSE2, AVX2, etc.)
- Support may vary depending on processor
- SIMD support is usually in the form of extensions to existing instruction sets
- SIMD extensions give access to extra “vectorized” registers and instructions

# SIMD Implementation (cont'd)

- SSE2 extensions adds 16 128bit registers (xmm0-xmm15)
- AVX extensions add 16 256bit registers (ymm0-ymm15)
- 128bit register can hold 2 longs/doubles, 4 ints/floats,
- The instructions used determine how the data in the register is treated:
  - paddd (add packed double-word integers treats the 128bit register as 4 packed 32bit integers)

# SIMD Implementation (cont'd)

- C/C++ compilers can often generate SIMD instructions and auto-vectorize code in certain scenarios automatically (e.g. tight loops with basic arithmetic)
- C/C++ compilers provide access to hardware intrinsics functions that you can call in your code if hardware supports it
- The SIMD instructions must be supported on the target hardware. You should know this at compile time.
- Generated binary is hardware specific

# SIMD in .NET

- SIMD support was added to the standard library in recent versions of .NET:
  - System.Numerics
  - System.Runtime.Intrinsics
- Requires RyuJIT compiler, which is the default JIT compiler in recent versions of the .NET runtime
- RyuJIT does not auto-vectorize code, you have to manually call SIMD libraries

# SIMD in .NET – System.Numerics

- Provides fixed-size Vector types:
  - Vector2, Vector3, Vector4, Matrix3x2, etc.
- Provides generic Vector<T> type:
  - Vector<T>.Count = size of register/sizeof(T)
- Overloads basic operators for vectorized arithmetic: e.g. v1 + v2 for vectorized member-wise addition
- Provides methods for vectorized operations e.g. Vector.Max(v1, v2)
- Detects hardware support at runtime and falls back to software implementation if SIMD not supported.

# SIMD in .NET – System.Runtime.Instrinsics

- Provides access to hardware-specific instructions:
  - System.Runtime.Instrinsics.X86.Sse2.Add(Vector128<float>, Vector128<float>)
- Allows you to take advantage for more hardware capabilities.
- Does not have software fallback implementations
- Hardware support can be checked at runtime before use: Sse2.IsSupported

# Example – Member-wise addition

- Scalar implementation

```
void Add(int[] A, int[] B, int[] result)
{
    for (int i = 0; i < A.Length; i++)
    {
        result[i] = A[i] + B[i];
    }
}
```

# Example – Vector addition (cont'd)

- Scalar implementation, what is happening:

```
int i = 0;
if (i >= length) go to loop end
result[i] = A[i] + B[i];
i++;
if (i >= length) go to loop end
result[i] = A[i] + B[i];
i++
if (i >= length) go to loop end
result[i] = A[i] + B[i];
i++
if (i >= length) go to loop end
result[i] = A[i] + B[i];
i++
if (i >= length) go to loop end
result[i] = A[i] + B[i];
i++
if (i >= length) go to loop end
result[i] = A[i] + B[i];
i++
// etc.
```

# Example – Vector addition (cont'd)

- How it would look like vectorized (assuming 128-bit vectors):

```
int i = 0;
Vector<int> vA;
Vector<int> vB;
Vector<int> vRes;

if (i >= length) go to loop end
copy A[i..i+4] to vA
copy B[i..i+4] to vB
vRes = vA + vB;
copy vRes to result[i..i+4]
i += 4;
if (i >= length) go to loop end
copy A[i..i+4] to vA
copy B[i..i+4] to vB
vRes = vA + vB;
copy vRes to result[i..i+4]
i += 4;
// etc.
```

# Example – Vector addition (cont'd)

- Vectorized implementation

```
void Add(int[] A, int[] B, int[] result)
{
    for (int i = 0; i < A.Length; i += Vector<int>.Count)
    {
        Vector<int> vA = new Vector<int>(A, i);
        Vector<int> vB = new Vector<int>(B, i);
        Vector<int> vRes = vA + vB;
        vRes.CopyTo(result, i);
    }
}
```

# Example – Benchmarks

- Vector addition and array sum (total sum of array elements)

```
// * Summary *

BenchmarkDotNet=v0.13.1, OS=Windows 10.0.22000
Intel Core i7-8665U CPU 1.90GHz (Coffee Lake), 1 CPU, 8 logical and 4 physical cores
.NET SDK=6.0.201
  [Host]     : .NET 6.0.3 (6.0.322.12309), X64 RyuJIT
  DefaultJob : .NET 6.0.3 (6.0.322.12309), X64 RyuJIT


|              Method |    Categories | dataSize |         Mean |      Error |      StdDev |        Median | Ratio | RatioSD | Code Size |
|-------------------- |-------------- |--------- |-------------:|-----------:|------------:|--------------:|------:|--------:|----------:|
| MemberWiseSumScalar | MemberWiseSum |     4096 |     4.975 us |  0.1233 us |   0.3635 us |     4.9436 us |  1.00 |    0.00 |     184 B |
|   MemberWiseSumSIMD | MemberWiseSum |     4096 |     4.052 us |  0.3090 us |   0.9111 us |     3.9933 us |  0.82 |    0.21 |     134 B |
|                     |               |          |              |            |             |               |       |         |           |
|       ArraySumSalar |      ArraySum |     4096 |     3.771 us |  0.1445 us |   0.4262 us |     3.7618 us |  1.00 |    0.00 |      51 B |
|        ArraySumSIMD |      ArraySum |     4096 |     1.035 us |  0.1079 us |   0.3078 us |     0.9203 us |  0.28 |    0.08 |     136 B |
|                     |               |          |              |            |             |               |       |         |           |
| MemberWiseSumScalar | MemberWiseSum |    65536 |    67.470 us |  3.9889 us |  11.0533 us |    67.9839 us |  1.00 |    0.00 |     184 B |
|   MemberWiseSumSIMD | MemberWiseSum |    65536 |    36.167 us |  1.1960 us |   3.4123 us |    35.0679 us |  0.55 |    0.10 |     134 B |
|                     |               |          |              |            |             |               |       |         |           |
|       ArraySumSalar |      ArraySum |    65536 |    45.124 us |  1.4357 us |   4.1425 us |    44.3719 us |  1.00 |    0.00 |      51 B |
|        ArraySumSIMD |      ArraySum |    65536 |    14.721 us |  1.5395 us |   4.5391 us |    13.1433 us |  0.33 |    0.12 |     136 B |
|                     |               |          |              |            |             |               |       |         |           |
| MemberWiseSumScalar | MemberWiseSum |  1048576 | 1,582.610 us | 36.5532 us | 106.0473 us | 1,561.2326 us |  1.00 |    0.00 |     184 B |
|   MemberWiseSumSIMD | MemberWiseSum |  1048576 |   991.435 us | 19.1238 us |  18.7821 us |   982.2410 us |  0.58 |    0.02 |     134 B |
|                     |               |          |              |            |             |               |       |         |           |
|       ArraySumSalar |      ArraySum |  1048576 |   683.266 us | 13.0133 us |  11.5360 us |   684.4435 us |  1.00 |    0.00 |      51 B |
|        ArraySumSIMD |      ArraySum |  1048576 |   281.997 us | 23.0494 us |  65.7611 us |   289.8456 us |  0.26 |    0.02 |     136 B |
```

# Disadvantages

- Some algorithms are not easy to vectorize (e.g. when there are data dependencies, conditions or a lot of flow-control logic)
- Vectorized code may be more complex and harder to maintain
- Vectorization does not always lead to improved perf, (e.g. when there are memory bottlenecks), sometimes it may degrade perf
- Portability: SIMD is not supported on all hardware, or maybe supported differently on different hardware
- Dealing with low-level challenges: data alignment, different register sizes

# Resources

- https://docs.microsoft.com/en-us/dotnet/standard/simd
- https://devblogs.microsoft.com/dotnet/hardware-intrinsics-in-net-core/
- https://instil.co/blog/simd-performance-with-csharp-and-cpp/
- https://instil.co/blog/parallelism-on-a-single-core-simd-with-c/
- https://adamsitnik.com/Disassembly-Diagnoser/
- https://en.wikipedia.org/wiki/X86_instruction_listings#SSE2_instructions
- https://en.wikipedia.org/wiki/Advanced_Vector_Extensions#New_instructions
- https://en.wikipedia.org/wiki/Single_instruction,_multiple_data#Disadvantages
- https://devblogs.microsoft.com/dotnet/using-net-hardware-intrinsics-api-to-accelerate-machine-learning-scenarios/
- https://simdjson.org/
- https://docs.oracle.com/cd/E18752_01/html/817-5477/epmpv.html