Intel® Network Builders

# Making vRAN Software Portable Across Xeon Generations with C++ SIMD class library

Nic Chautru
Network Edge Group, Wireless Access Network Division
September 26, 2022

intel®

# Outline

1. Intel FlexRAN Reference Architecture includes an optimized solution for building open, virtualized RAN It aims for highly performant implementation for optimized signal processing workload while keeping SW consistency across micro-architecture generations.

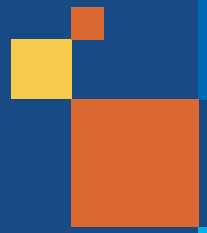
2. Available programming techniques: Intel C++ Class libraries and templates usage to allow developers to simplify code development and evolution, portable across multiple generations of Xeons, while still maximizing SIMD capabilities in each processor.

3. Examples of coding approach used in intensive workloads such as virtualized RAN (FlexRAN SDK) showing:

   - leveraging existing classes maintained by Intel and tightly coupled with oneAPI intel compiler
   - writing cleaner, smaller and more maintainable code, not relying on low level intrinsics
   - software portability and reuse across multiple generations: allowing code to coexist with previous generations and to exploit the latest features in an efficient way notably for the new ISA in SPR

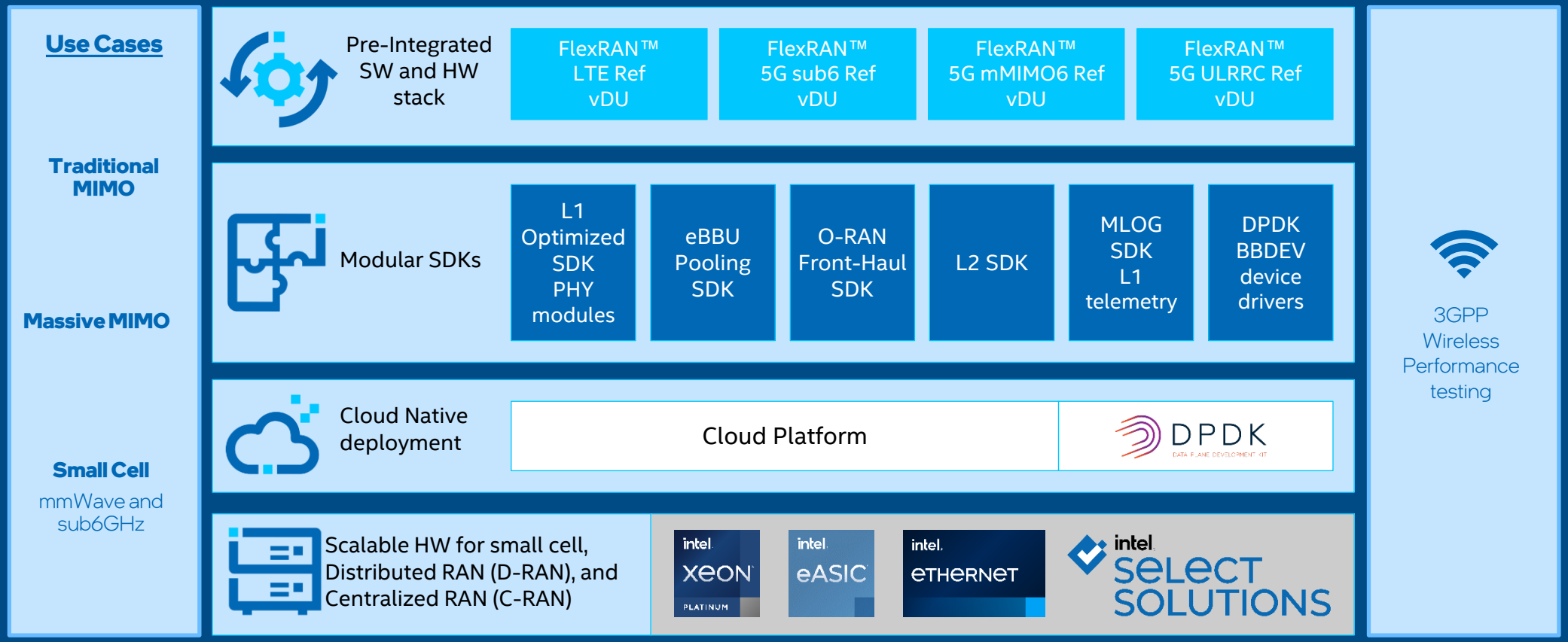
# Intel FlexRAN Reference Architecture

Optimized reference code for L1 signal processing workload while keeping SW consistency across micro-architecture generations and exploiting latest CPU features.



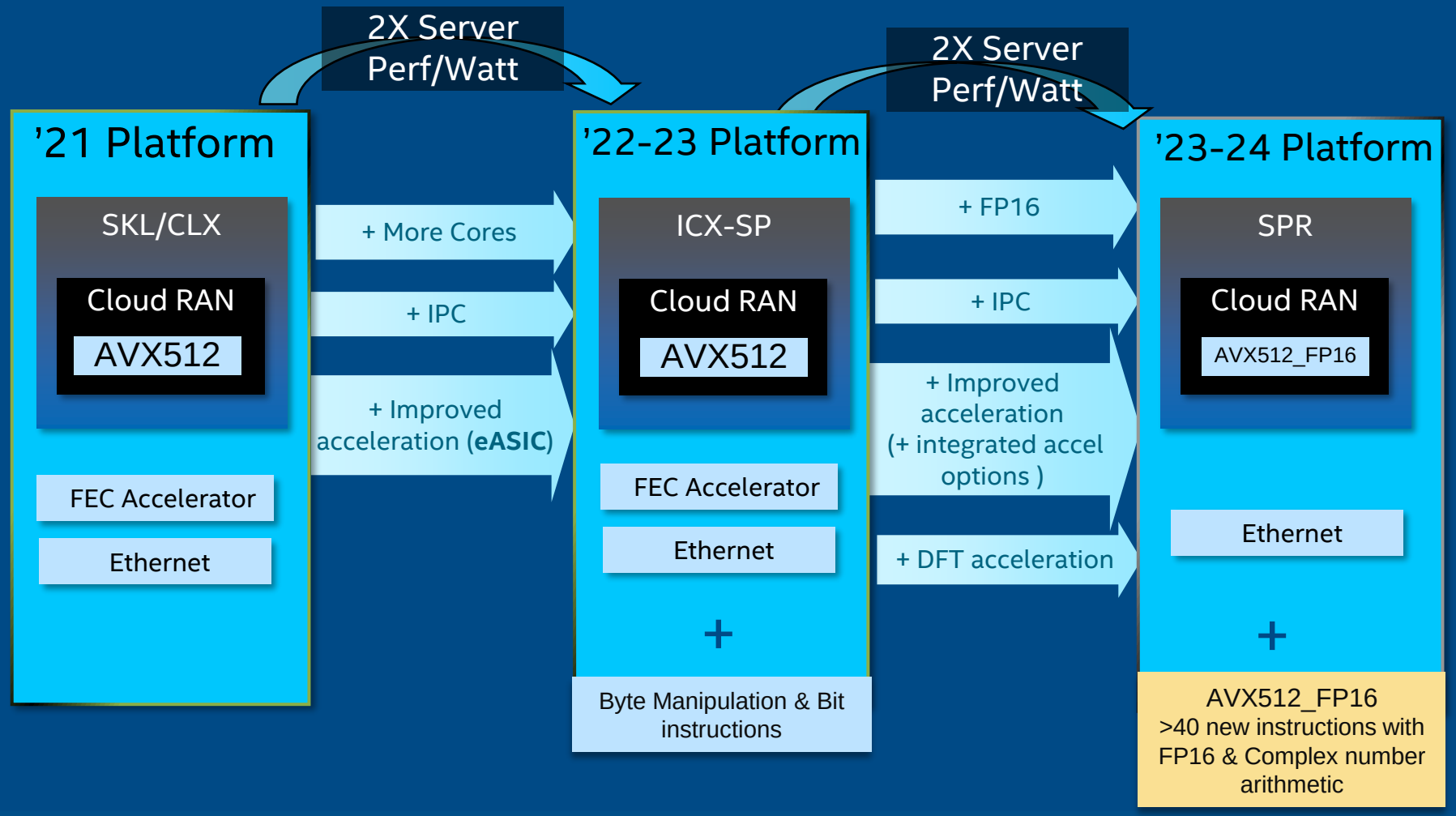
intel®

# FlexRAN™ Reference Architecture




FlexRAN is a SW and HW reference solution to enable customers build and deploy highly optimized, scalable 4G/5G Cloud Native, fully virtualized, Open RAN solutions on Intel architecture

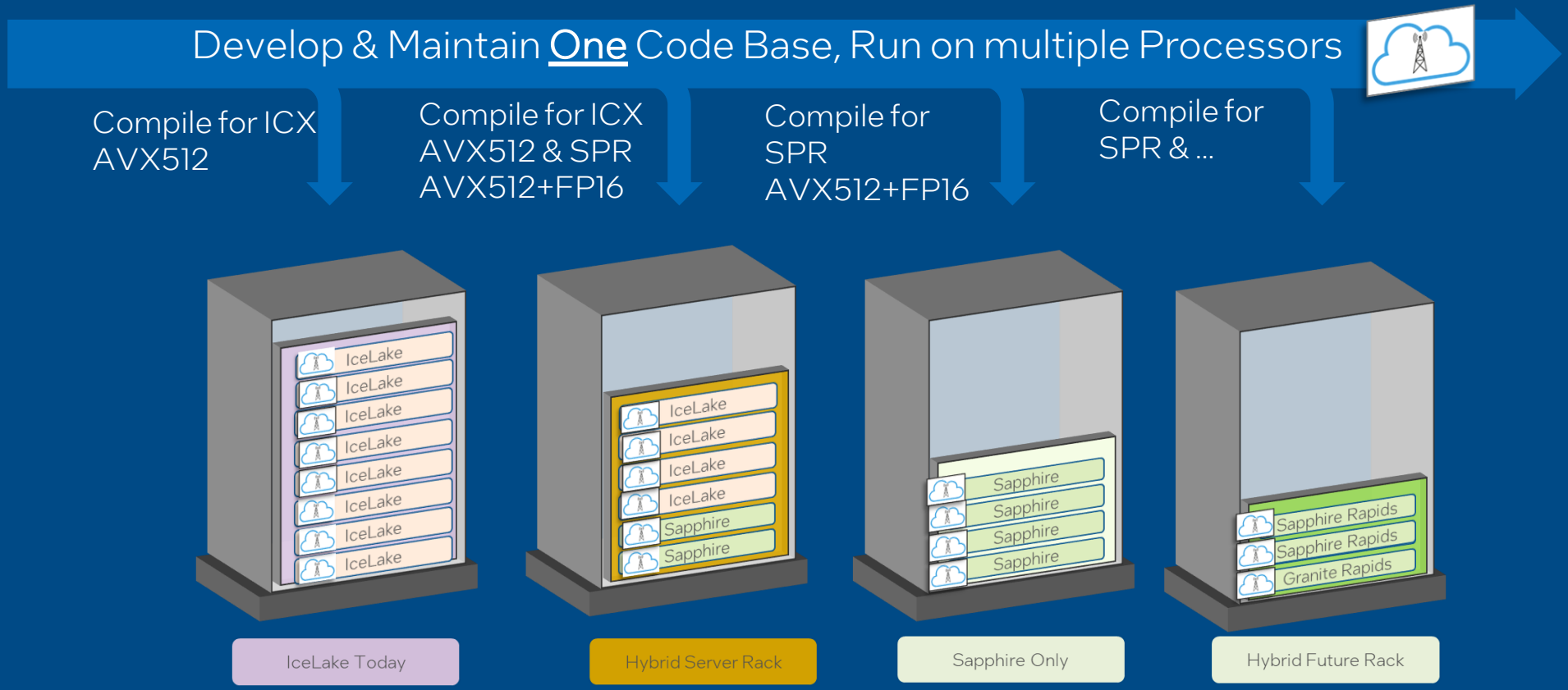
# Intel vRAN Roadmap – 2X Every Gen




Each Generation of Intel ® Xeon Delivering 2x baseband performance gain with ISO power.

Keeping SW consistent across generations while maintaining flexibility

# C++ Class & Standard Libraries for High Performant & Portable SW




- Intel C++ Class libraries allow developers to simplify code development and evolution , increase forward and backward portability and decrease maintenance costs
- Intel FlexRAN 22.03 (FlexRAN's first major Dvec release) delivered
- Measurable reduction in source code; on multiple uArchitectures delivering open RAN HW / SW Disaggregation

# AVX512_FP16– Sapphire Rapids

## New Built-in Instructions for 5G RAN workloads

### Benefits

**Half Precision (IEEE754) 16-bit floating point data types**

Compared to 32-bit single precision floating point data types it gives better data storage efficiency (cache, memory, registers) and higher throughput

**Complex Data Operations**

Plain and fused multiply operations, and conjugated variants of the same

**Reduced Development Complexity**

Floating point offers wide dynamic range, while implicitly offloading dynamic ranging/scaling to HW compared to 16-bit integer

### FlexRAN Layer 1

### New Instructions

- **42 New Instructions:** designed for entire algorithms to be written in FP16
- **Basic Operations:** add, sub, mul and div
- **Fused Operations:** fmadd, fmsub, fmaddsub… including negated equivalents (fnmadd)
- **Extended Operations:** rcp, rsqrt, min/max, compare, scale
- **Conversions:** to and from signed and unsigned integer and other floating-point formats
- **Complex Instructions:** plain and fused multiply operations and conjugated variants
- **General 16-bit support:** provided using existing 'epi16' AVX512 instructions

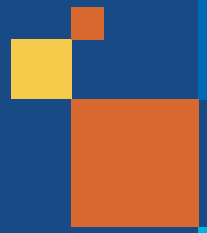
### Key Workloads, Libraries and Tools Support

**Workloads:** Layer Mapper, Resource Element Mapper, Precoding, Front Haul Formatting, Eigen Beamforming, SRS Channel Estimation, LLR De-mapping, PUSCH Channel Estimation, MMSE Equalization, DMRS Generation

**SW Libraries:** FlexRAN SDK, MKL

**Tools:** Compilers, Intel Debugger, Software Development Emulator, DVEC

Intel C++ Class libraries and templates allow developers to simplify code development and evolution, and increase forward and backward portability



intel®

# Accessing processor vector capabilities

- Signal processing algorithms implementation relying heavily on vectorized SIMD processing (AVX2, AVX512, AVX512_FP16)
- Broadly speaking, there are three ways to access vector features of the processor:

- Auto-vectorisation: Try to let the compiler do it for you.
- Intrinsics: Explicitly tell the compiler what you want
- Overloaded vector operations: syntactic sugar on top of intrinsics:
  - Use Intel's C++ SIMD class library (tight integration to OneAPI optimisation)
  - Convenient interface to access underlying instructions
  - Other libraries also exist

# Use of intrinsics to access ISA-specific features

The compiler can't always get the best performance out of the processor:

- Auto-vectorisation doesn't always do what the programmer wants
- Some instruction set features are not designed to be accessible by the compiler.

In this case, intrinsics can be used to directly access the underlying instruction set to allow the programmer to express themselves directly.

Using intrinsics allows direct access to the processor, and potentially very high performance. For example, a multiply-accumulate (fused-multiply-add) operation:

AVX2:

```
__m256 a, b, c;
const auto mac = _mm256_add_ps(_mm256_mul_ps(a, b), c);
```

AVX-512:

```
__m512 a, b, c;
const auto mac = _mm512_add_ps(_mm512_mul_ps(a, b), c);
```

BUT Intrinsics can be verbose, tedious to write, hard to read, and make it difficult to quickly get the sense of a program for non low-level experts.

Portability of code is reduced because the intrinsic names tend to be hard-coded to the platform

# C++ SIMD header allows operator-like programming style

The C++ SIMD library provides a full suite of operators to turn abstract vector types into built-in types:

AVX2:

```
F32vec8 a, b, c;
const auto mac = a * b + c;
```

SIMD data type – 8 x 32-bit floating-point in one vector

Clean, simple code fragment working on SIMD data types using normal operators.

AVX-512:

```
F32vec16 a, b, c;
const auto mac = a * b + c;
```

Only the data type has changed – the mac code fragment is identical.

Identical code will be generated compared to the original intrinsics.

C++ SIMD library code is just as fast, and it's readable too, including support for operators and functions/utility

Intel SIMD library supports many ISAs and data types

- Instruction sets: MMX, SSE, AVX, AVX2, AVX-512, AVX-512_FP16, future.
- Data Types:  Integer 8/16/32/64, float 16/32/64
- dvec, ivec, fvec, hvec, vec_complex all incrementally extend the C++ class templates reference for SIMD

# Reasons to use Intel's SIMD library

There are alternatives to the Intel C++ SIMD class library  (e.g., open source projects, Vector Class Library, custom made…).

There are several good reasons to use Intel's library, rather than an alternative or custom solution:

- It is maintained by Intel as part of the compiler. No need to invest time and effort maintaining custom header libraries.
- Tight optimisation integration with the Intel compiler (e.g., conjugate support in hvec is provided in such a way that oneAPI peephole optimisations work effectively).
- Intel's SIMD library exposes Intel-specific SIMD features and exploits those features in efficient ways.
- It provides a layer of abstraction which makes the code easier to understand than using intrinsics and in many cases is more portable
- Intel's C++ SIMD class library is supplied as part of the Intel compiler release, and works best in that environment

# Templates allow separation of types from algorithms

- Many algorithms will work on any data type. For example, sum a sequence of values.
- Using templates in association with dvec enables the use of a single generic function.

```
template<typename T>
T sum(T* values, int size) {
  T result = 0;
  for (int i=0; i<size; ++i)
    result = result + values[i];
  return result;
}
------
F32vec16 values[1024];
auto sum = sum(values, 1024);
```

- Allows for separation of type from algorithm
- Reduces function duplication as the same function can be called for any data type.
- Common algorithms can be written once, and then reused on other SIMD data types.

# Templates used for algorithm specialization

- The other use for templates in the FlexRAN code is for compile time specialization which brings other specific noteworthy advantages
- Typically used for dimensions (matrix, number of antennas, layers…)
- The compiler knows exactly the size of the problem and can generate more efficient specialized code (no loop nor branch, just straight-line parallel code, more likely to inline).
- Specialization through template allows optimized code while still using a very compact code base. The associated reduction in source code foot print enables lowered software development costs and higher software quality.
- Can increase code size but it allows highly efficient code with limited code base.

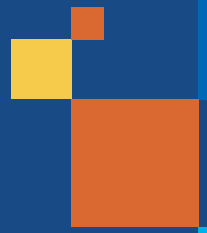
```
template<size_t N_RX, size_t N_L, uint8_t nLayerPerUe, typename T>
void mimo_mmse_llr_avx512_interp(bblib_pusch_irc_symbol_processing_request *request,
        bblib_pusch_irc_symbol_processing_response* response)
```

# Templates used for algorithm specialization (contd)

- Kernels need at least some form of run-time parameterization. Use branching to select the appropriate specialisation:
- Select at run-time:

```
    if (1 == nLayer) {
        if(1 == nLayerPerUE && 1 == nUeInGroup) {
            switch (nRxAnt) {
            case 1: mimo_mmse_llr_avx512_interp<1, 1, 1, T>(request, response);
            case 2: mimo_mmse_llr_avx512_interp<2, 1, 1, T>(request, response);
            case 4: mimo_mmse_llr_avx512_interp<4, 1, 1, T>(request, response);
            case 16: mimo_mmse_llr_avx512_interp<16, 1, 1, T>(request, response);
            default: throw std::runtime_error ("Invalid number of Ants");
            }
        }
    } else if(2 == nLayer) {
…
```

# FlexRAN SDK examples



intel®

# Example 1:

- Frequency domain correlation between 2 IQ sequences
- On SPR this can be run using <CF16vec16> for complex floating point data type
- On ICX this can be run using <Is16vec32> for fixed point data type (I+Q)

- The underlying function may use different specialized implementation
- On SPR the new native complex ISA-specific capabilities are being exploited as only SPR has support for complex-valued hardware instructions while keeping common code with previous generation.

```
template<typename T>
void fd_correlation(
            const bblib_fd_correlation_request *request,
            bblib_fd_correlation_response *response)
{
    constexpr unsigned complex_values_per_t = sizeof(T) / 4;
    const int16_t n_loop = request->len / complex_values_per_t;
    const int16_t n_rem  = request->len - (n_loop * complex_values_per_t);
    T *pIn0 = reinterpret_cast<T *>(request->in0);
    T *pIn1 = reinterpret_cast<T *>(request->in1);
    T *pOut = reinterpret_cast<T *>(response->out);

    #pragma unroll(8)
    for(int i=0; i<n_loop; i++)
    {
        auto in0 = loadu(pIn0 + i);
        auto in1 = loadu(pIn1 + i);
        auto out = fmulconj(in0, in1);
        storeu(pOut++, out);
    }
<…>
}
----------------
fd_correlation<CF16vec16>(request, response);

fd_correlation<Is16vec32>(request, response);
```

# Example 2:

- MMSE MIMO equalization
- Template usage based on N_TX, N_RX
- When type do not allow to reuse code, option to have parallel implementation using function overloading for independent specialized implementation and/or specialization
- Compile time selector to switch between the 2 options based on data type used (here complex split into 1 or 2 arrays).

```
template<typename T, size_t N_RX = 16, size_t N_TX = 16>
static void mimo_mmse_llr_avx512(bblib_pusch_symbol_processing_request *request,
bblib_pusch_symbol_processing_response* response){
{
        T *pRxIn[BBLIB_N_SYMB_PER_SF][N_RX], *pChIn[1][N_TX][N_RX];
        T ChIn[N_TX][N_RX];
…
                //1. A = H' * H + Sigma2
                if constexpr (fp16Int16 == FP16_E::INT16) {
                    HxH ( ftempARe, ftempBRe, ChIn, ftempAIm,
                                    ftempBIm, ChImNegRe, avxfSigma2);
                    // 2. invA = inv(H' * H + Sigma2*I)
                    matrix_inverse<N_TX>(ftempBRe, ftempBIm, finvARe,
                                     finvAIm);
                    // 3. gain calc
                    gainCalc(ftempGain, ftempPostSINR, avxGainShift,
                                    finvAIm, finvARe, ftempAIm, …);
                }
                else
                {
                    //1. A = H' * H + Sigma2
                    HxH<T, N_TX, N_RX> ( ftempARe, ftempBRe, ChIn,
                                    avxfSigma2);
                    // 2. invA = inv(H' * H + Sigma2*I)
                    matrix_inverse<T, N_TX>(ftempBRe);
                    // 3. gain calc
                    gainCalc(ftempGain, ftempPostSINR, ftempBRe, ftempARe…);

                }
…
    }
}
----------------
else if(3 == nLayer) {
            if (4 == nRxAnt) {
                mimo_mmse_llr_avx512_interp<T, 4, 3>(request, response);
```

# Example 3:

- There are still cases when dev/hvec/vec_complex may not always be sufficient
- We can define some custom classes in case missing which can be used within the code as an extension for default available vec_complex support for CF16vec16.
- The actual implementations would be using the available native instruction

```
// complex conjugates mul a * conj(b)
  friend CF16vec16 fmulconj (const CF16vec16 &a, const CF16vec16 &b) { return
_mm512_fcmul_pch(a.vec, b.vec); }

/* multiply complex: (A + iB)*(C+iD) = AC-BD + i(AD+BC) */
inline FORCE_INLINE Is16vec32 fmul(const Is16vec32& input0, const Is16vec32& input1)
{
    const __m512i m512_sw_r = _mm512_set_epi8(
        61, 60, 61, 60, 57, 56, 57, 56, 53, 52, 53, 52, 49, 48, 49, 48,
        45, 44, 45, 44, 41, 40, 41, 40, 37, 36, 37, 36, 33, 32, 33, 32,
        29, 28, 29, 28, 25, 24, 25, 24, 21, 20, 21, 20, 17, 16, 17, 16,
        13, 12, 13, 12, 9, 8, 9, 8, 5, 4, 5, 4, 1, 0, 1, 0);
    const __m512i m512_sw_i = _mm512_set_epi8(
        63, 62, 63, 62, 59, 58, 59, 58, 55, 54, 55, 54, 51, 50, 51, 50,
        47, 46, 47, 46, 43, 42, 43, 42, 39, 38, 39, 38, 35, 34, 35, 34,
        31, 30, 31, 30, 27, 26, 27, 26, 23, 22, 23, 22, 19, 18, 19, 18,
        15, 14, 15, 14, 11, 10, 11, 10, 7, 6, 7, 6, 3, 2, 3, 2);
    const Mask32  nMaskNegQ =0x55555555;

    // Select real or image part from a complex value
    __m512i ReRe = _mm512_shuffle_epi8(input0, m512_sw_r);
    __m512i ImIm = _mm512_shuffle_epi8(input0, m512_sw_i);

    // Swap real or image part and negative image part from a complex value
    // switch IQ
    __m512i tmp1 =    _mm512_rol_epi32(input1,16);/* t1,t0,t3,t2,t5,t4,t7,t6 */

    // Negative the Q part
    __m512i negImPosRe = _mm512_mask_sub_epi16(tmp1, nMaskNegQ, _mm512_setzero_si512(), tmp1); /* -
t1,t0,-t3,t2,-t5,t4,-t7,t6 */

    // Multiply complex
    tmp1 = _mm512_mulhrs_epi16(ReRe, input1);
    __m512i tmp2 = _mm512_mulhrs_epi16(ImIm, negImPosRe);
    return _mm512_adds_epi16(tmp1, tmp2);
}


-----------

xmmEstH[1] = fmulconj(S_CAST<Is16vec32>(xmmHTemp[1]),S_CAST<Is16vec32>(xmmTAComp));
```

# Example 4:

- Polar decoder implementation using a recursive template
- Organized different so that the compiler can understand the structure and optimize accordingly : Size explicitly known through specialization, not calling itself, flat raw code generated without branches, with lots of parallelism.
- Possible to mix this conventional function for higher levels to find right balance on flattening the code.
- Possible to specialize some specific sizes when developer knows better for some special cases
- Also applicable for FFT butterfly algorithm
- Maximized code reused and optimized

```
template<unsigned NUM_LLRS, typename ISA>
struct PolarListRecursiveInt16
{
static constexpr unsigned halfLlrs = NUM_LLRS / 2;
static ListStructure
Recurse (const __m128i* RESTRICT llrs_in, const SimdBitset<NUM_LLRS>& frozen,
            const SimdBitset<NUM_LLRS>& parity, BitList* RESTRICT message,
            BitList* RESTRICT codeword, Is16vec8& ref_metric,
            __m128i& ref_parity_reg, unsigned& ref_decision)
    {
        __align(64) __m128i llrBuffer[halfLlrs] = { };

        ISA::template LlrPriorMinProdInt16<halfLlrs * 8>(llrs_in, llrs_in +
halfLlrs, llrBuffer);
        SimdBitset<halfLlrs> frozenUpper = frozen.Lower();
        SimdBitset<halfLlrs> parityUpper = parity.Lower();

            ISA::template LlrPosteriorFrozenInt16<halfLlrs * 8>(llrs_in, llrs_in
+ halfLlrs, llrBuffer);
<…>
            list2 = PolarParityListRecursiveInt16<halfLlrs, ISA>::
      Recurse(llrBuffer, frozenLower, parityLower, message + list1.msg_len,
      codeword + halfLlrs, ref_metric, ref_parity_reg, ref_decision);
        }
<…>
    }
};
```

# Summary

1. Intel FlexRAN Reference Architecture is an optimized solution for building open, virtualized RAN
   It aims for highly performant implementation for optimized signal processing workload while keeping SW consistency across micro-architecture generations.

2. Available programming techniques: Intel C++ Class libraries and templates usage to allow developers to simplify code development and evolution, enable portability across multiple generations of Xeons, while still maximizing leverage of SIMD capabilities in each processor.

3. Examples of coding approach used in intensive workloads such as virtualized RAN (FlexRAN SDK) showing:
   - leveraging existing classes maintained by Intel and tightly coupled with oneAPI intel compiler
   - writing cleaner, smaller and more maintainable code, not relying on low level intrinsics
   - software portability and reuse across multiple generations: allowing code to coexist with previous generations and to exploit the latest features in an efficient way notably for the new ISA in SPR
   - reduce overall Total Cost of Ownership (TCO) and Faster Time To Market (TTM) without sacrificing performance

# Notices and Disclaimers

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See backup for configuration details. No product or component can be absolutely secure.

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Your costs and results may vary.

Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.