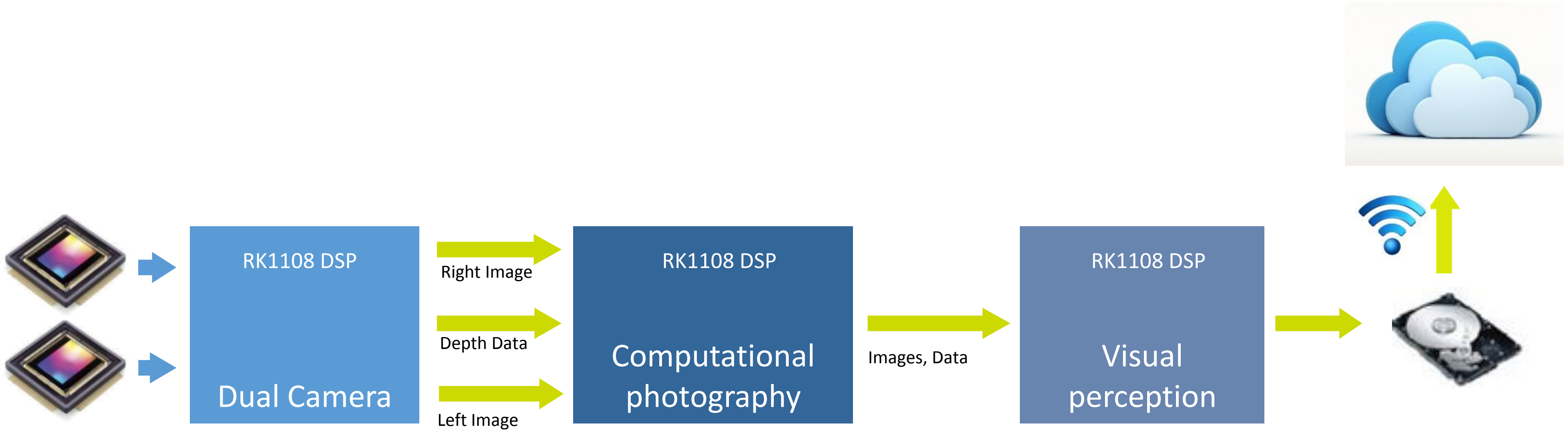
RK1108 DSP Platform

Algorithm Center

Outline

- **What's application you can use CEVA XM4 in RK1108**
- XM4 arch introduction
- XM4 SDT introduction
- XM4 program hints
- Develop your own algorithms on XM4

Intelligent Vision Processing



Dual Camera

- Optical Zoom
- Image Registration
- Depth map
- ISP

Computational photography

- Refocus image
- Video stabilization
- Low-light image enhance
- Zoom
- Super-resolution
- Background removal
- HDR

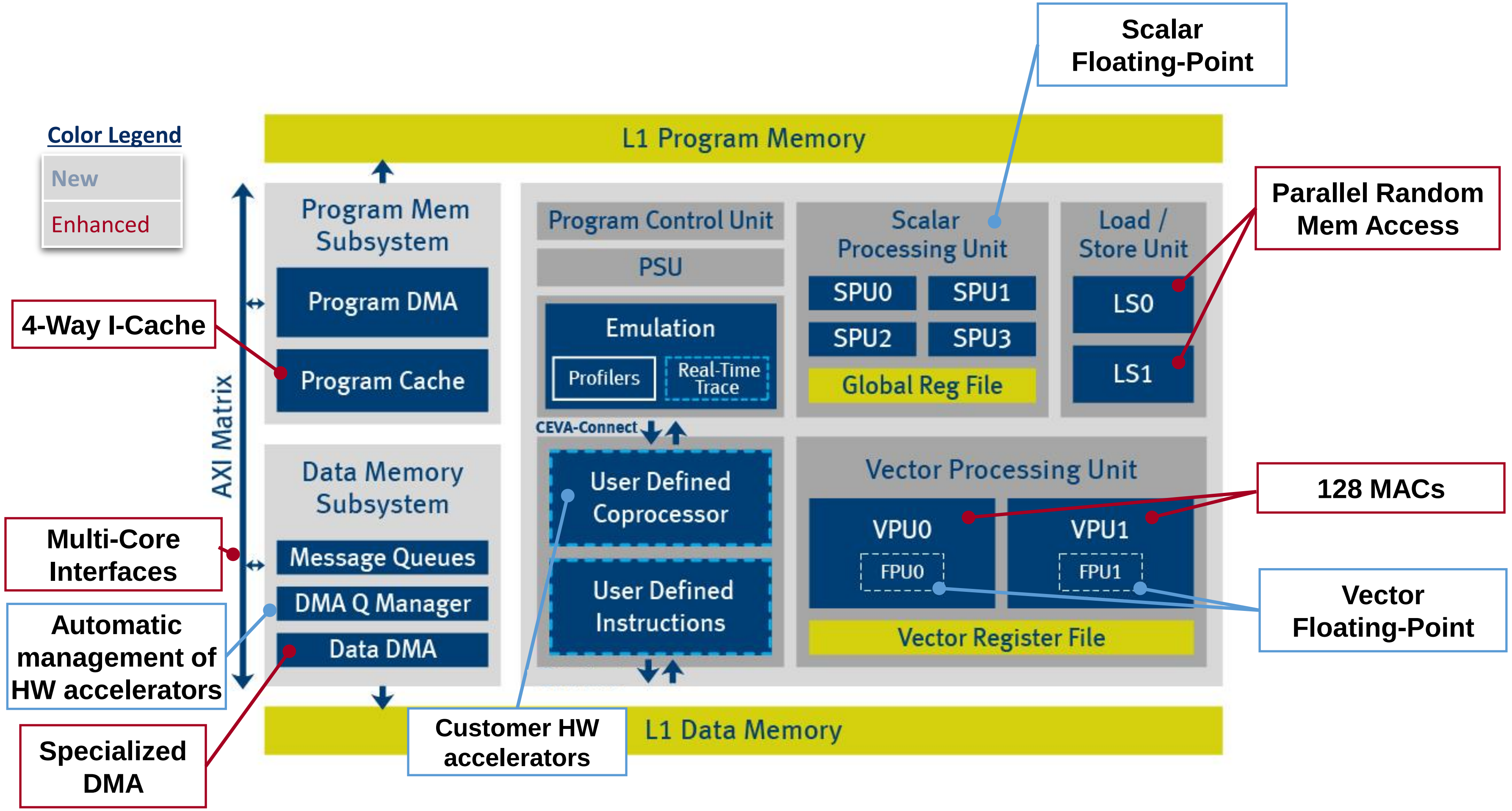
Visual perception

- Deep learning (CNN, DNN)
- Object detection, recognition & tracking
- Augmented reality (AR)
- Natural user interface (NUI)
- Context aware algorithms
- Biometric authentication

Outline

- What’s application you can use CEVA XM4 in RK1108
- **XM4 arch introduction**
- XM4 SDT introduction
- XM4 program hints
- Develop your own algorithms on XM4

XM4 Block Diagram



XM4 Architecture Highlights

DSP Core Features

- 8-way VLIW, dual VPU, 512-bit
- Flexible fixed-point 8/16/32-bit
- Vector and scalar floating-point support
  - Up to 20 FPUs per cycle (FPU32)
- 32 parallel random memory accesses
- Native non-linear functions support
  - 1/X, SQRT, 1/SQRT - Up to 32 fixed-point operations per cycle, also supported on vector FPUs
- 128 SAD operations (8-bit) per cycle
- Deep (14-stage) pipeline

MSS Connectivity

- 512-bit Internal data memory bandwidth
- 4-way set associative non-blocking i-cache
- Powerful DMAs for instruction and data Multi-core and system support

Most efficient vision processor (perf/mm^2, perf/mW)

XM4 Computing Capabilities

Extensive Fixed Point Capabilities

- 128 16x8 MACs per cycle
  - 64 16x16 MACs per cycle
  - 16 32x32 MACs per cycle
- Flexible operation types on vector
- Vector and scalar floating-point support
  - 8/16/32-bit native data types
    - 64/32/16 operations per cycle accordingly
  - Signed and unsigned full support
  - Inter-vector & intra-vector operations
- Native 32-bit operations in Scalar
  - 32x32 MAC/multipliers

Complementary Floating Point Support

- Scalar floating point
  - Single precision (double precision emulated)
  - Up to 4 floating point operations per cycle
- Vector floating point
  - Single precision (double precision emulated)
  - Up to 16 floating point operations per cycle

XM4 Computing Capabilities

Data Memory

- Dual load-store units - 2x256-bit load, 1x256 (2x64) store bandwidth
- 32 “random access” load/store operations per cycle
- Strong & flexible Data DMA capabilities
  - Parallel Load-Duplicate-Store
    - Single access copies into multiple locations
- DMA Queue managers
  - Automatic handling of DMA transfers through multiple transfer queues
  - Support automatic load balancing and prioritization

Program Memory

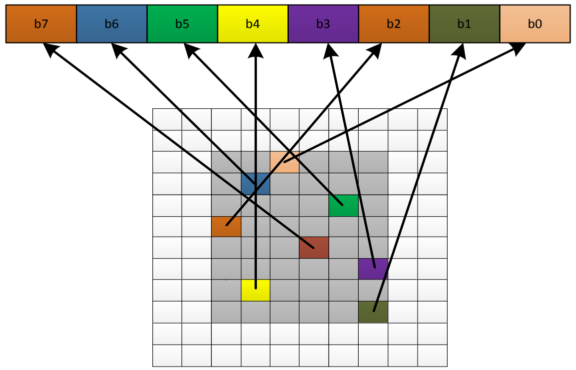
- 8-way VLIW
- 4-way non-blocking, set associative i-cache
- Program DMA
- Wide memory access (256-bit) to lower #accesses and increase power efficiency

Parallel Memory Access Mechanism

- **XM4 Scatter-gather capability enables load/store of vector elements into multiple memory locations in a single cycle**
  - Enables serial code vectorization
  - Significantly improves performance for various vision algorithms
- **Example: image histogram requires random accesses to memory per pixel**
  - Although the operation is very simple (adding 1 to a counter) this operation cannot be vectorized without appropriate memory access support

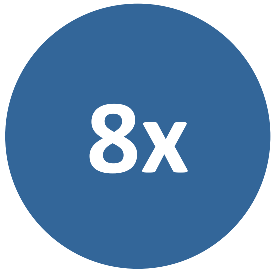
4x

Performance advantage using this mechanism (Histogram)

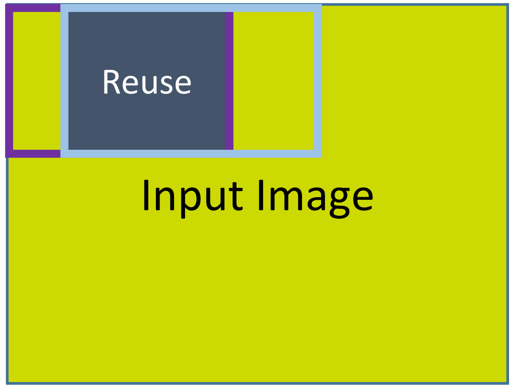


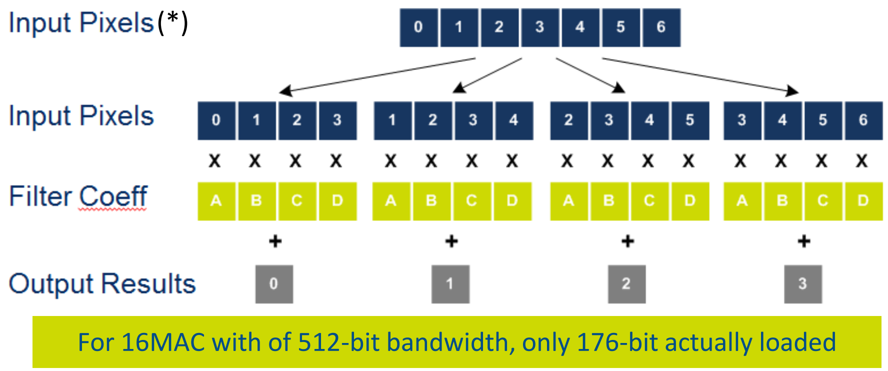
2-Dimension Data Processing Mechanism

- **XM4 processes 2D data efficiently**
- **Takes advantage of pixel overlap in image processing by reusing same data to produce multiple outputs**
  - Significantly increases processing capability
  - Saves memory bandwidth and frees system buses for other tasks
  - Reduces power consumption



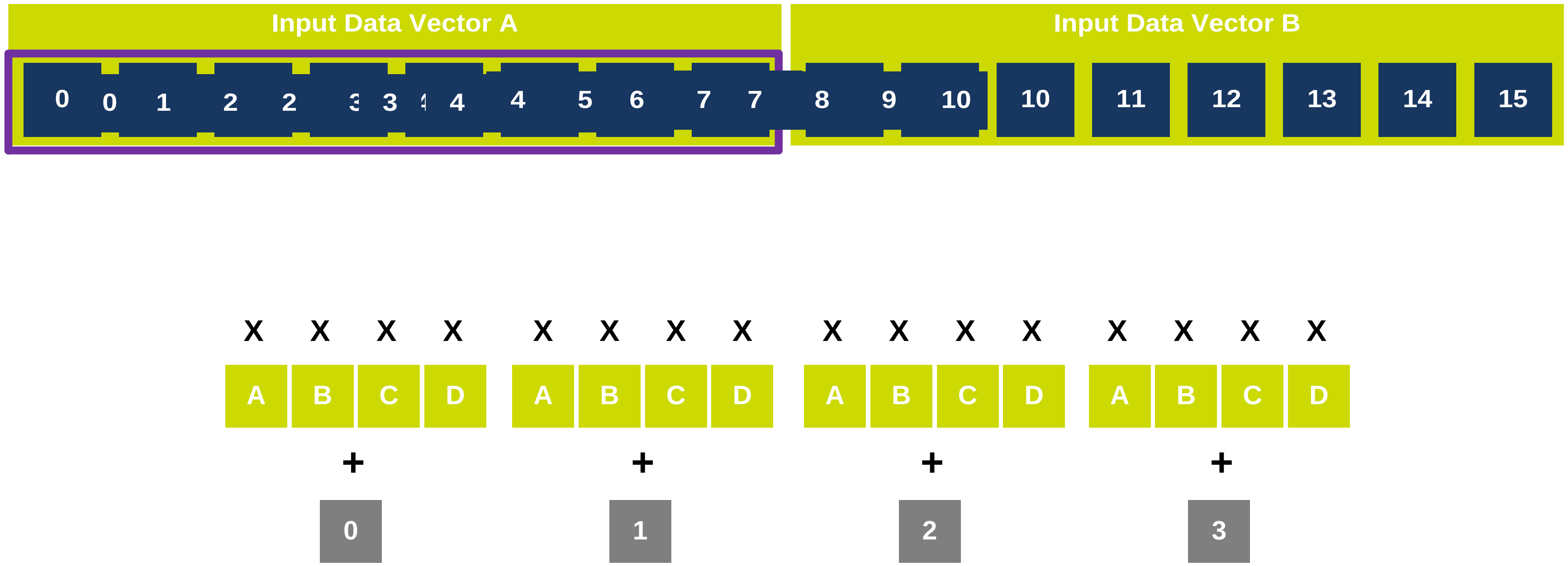
Lower memory bandwidth using 2D data processing





Input Data Reuse Example

- **The vector instruction performs the arithmetic operation multiple times in parallel (producing eight, 16 or 32 results). The next sliding-window operation is offset by a step from the previous operation.**

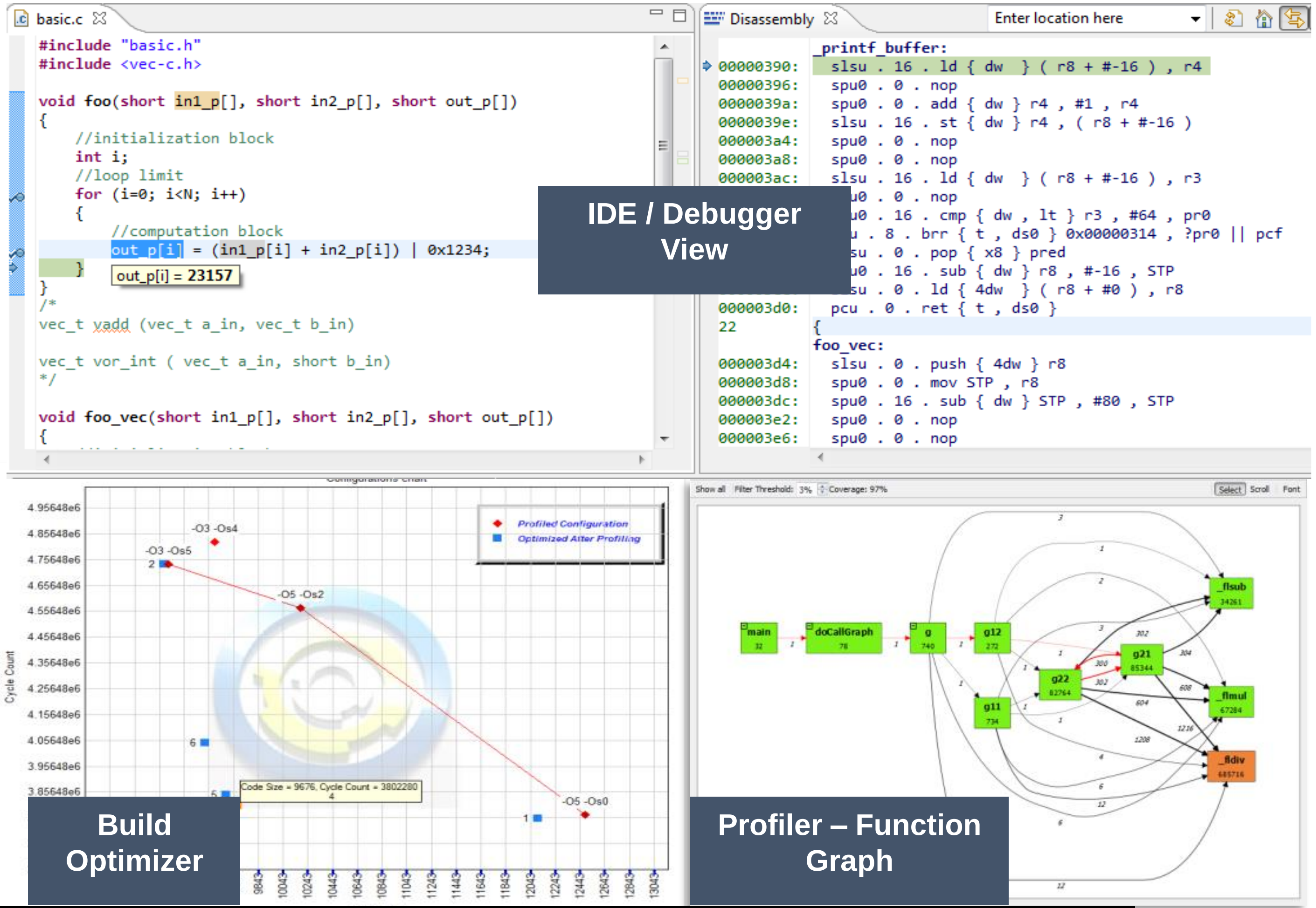


Outline

- What’s application you can use CEVA XM4 in RK1108
- XM4 arch introduction
- **XM4 SDT introduction**
- XM4 program hints
- Develop your own algorithms on CEVA XM4

IDE

- Advanced Eclipse-based IDE
- Optimizing C/C++ tailored compiler
  - Auto vectorization
  - Extensive Vec-C support
    - C language extensions in OpenCL-like syntax
    - Vector types for C/C++ - short8, ushort32,…
    - Vectorization from operators
- Linker and Utilities
- Built-in Debugger, Simulator & Profiler
  - Target Emulation dev. kit

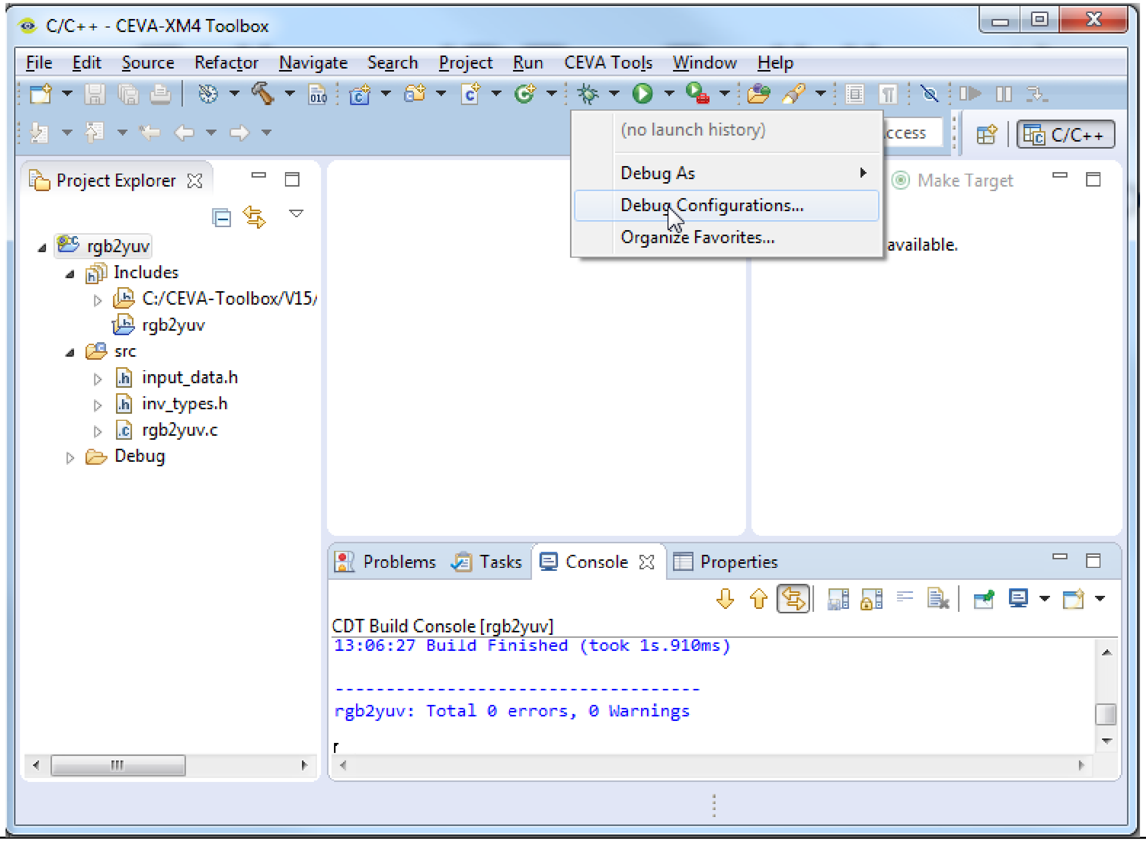


Complete Software Development tools.
Focused on ease of use and quick SW porting for performance optimization

IDE – Debugging

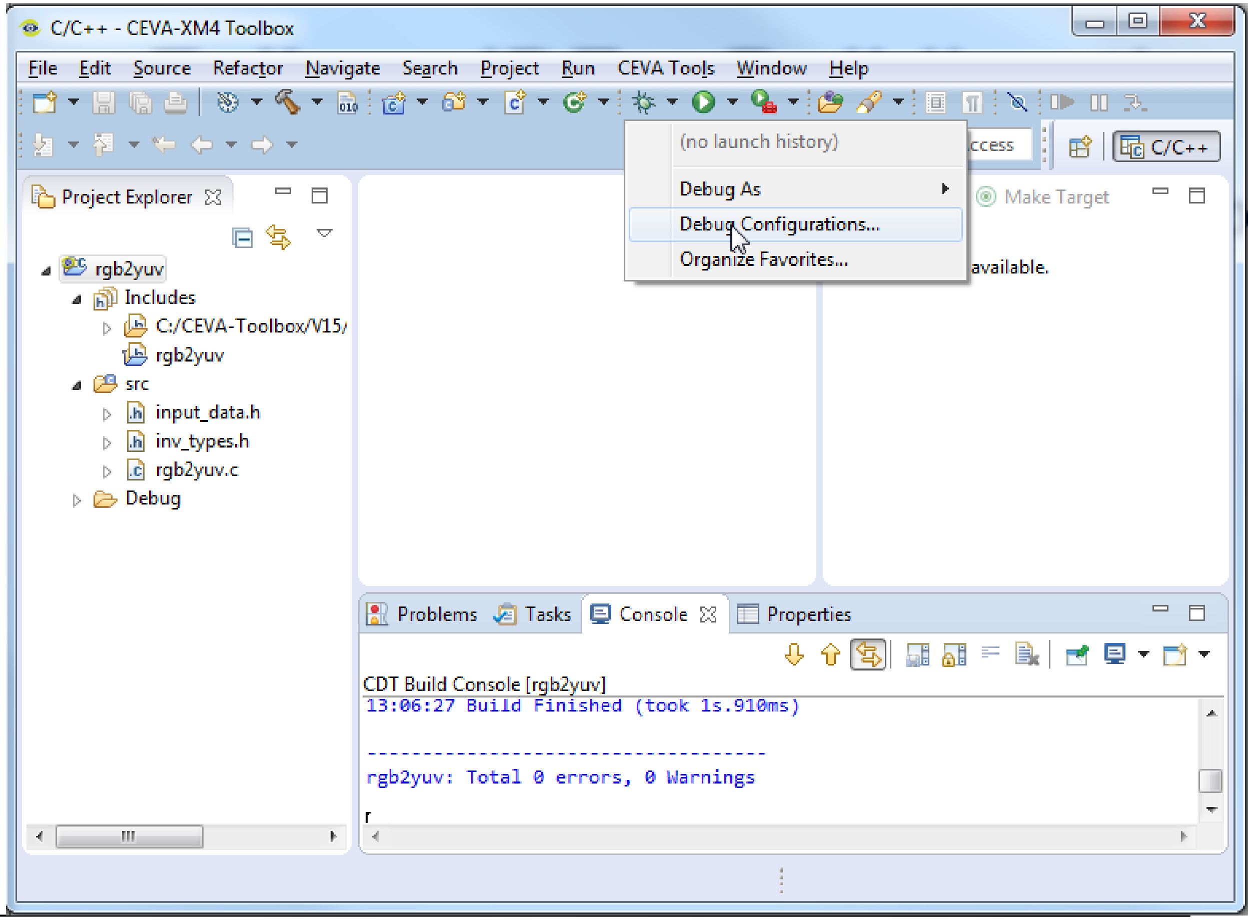
- To define a new debug configuration, press on the black triangle on the right side of the bug icon in the upper toolbar
- Select 'Debug Configurations'
  - And continue by following the instructions on the next slides
- An alternative quick method to enter debug:
  - Right click the project name in 'Project Explorer' 'Debug as' 'CEVA C/C++ Application Simulation'

  - It will create a new ISS debug configuration



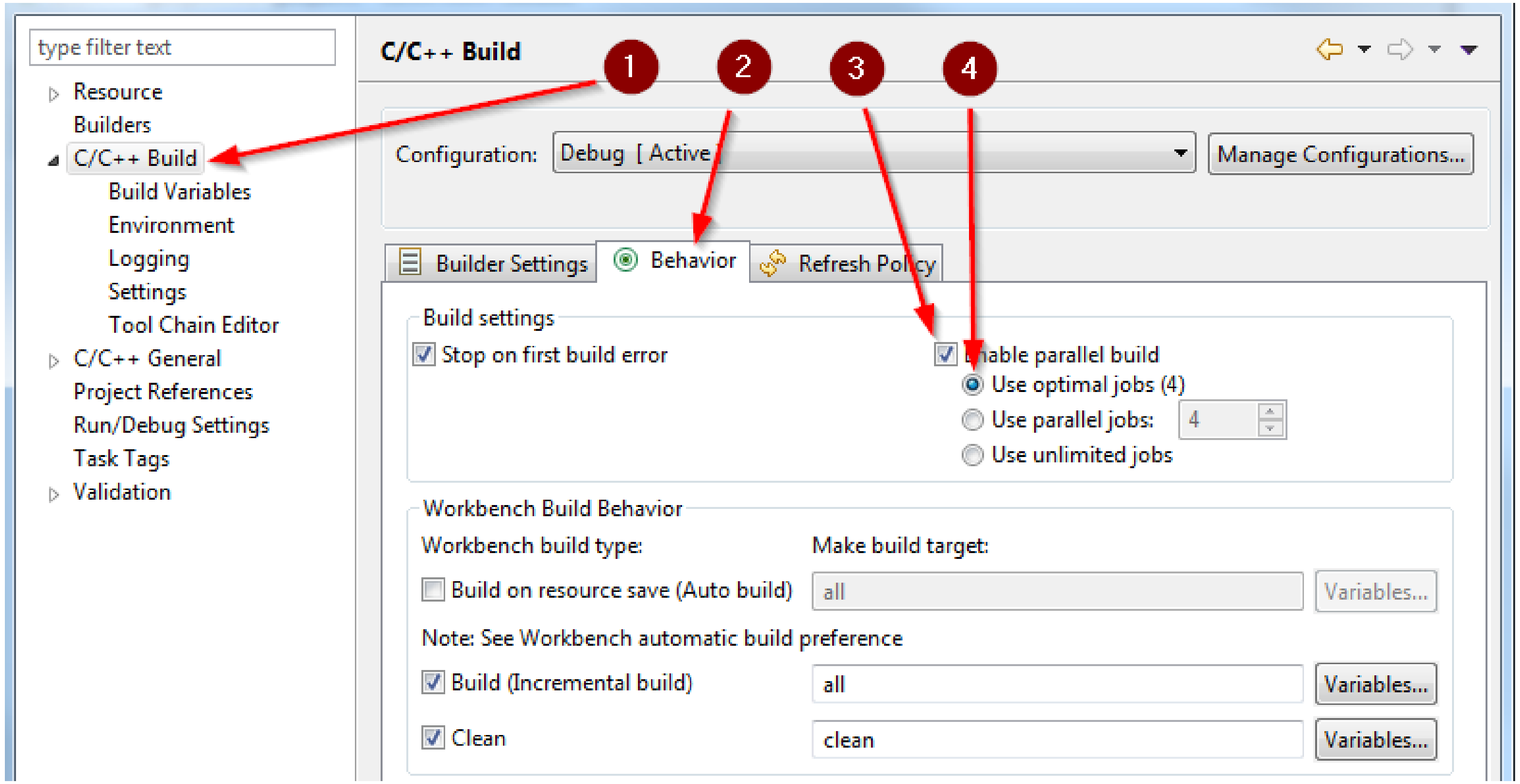
IDE – Profiling

- **We will define a new debug configuration to collect profiling information**
- **Press on the black triangle on the right side of the bug icon in the upper toolbar**
- **Select 'Debug Configurations'**



IDE – Parallel Build

- **CEVA Toolbox IDE parallel compilation by right clicking the project name in the 'Project Explorer' window, and choose 'Properties'**
- **Follow those steps:**



Outline

- What’s application you can use CEVA XM4 in RK1108
- XM4 arch introduction
- XM4 SDT introduction
- **XM4 program hints**
- Develop your own algorithms on CEVA XM4

Optimization levels comparison

	Advantage	Disadvantage
C level optimization	▪ Fast implementation ▪ Easier to reach bit exact results ▪ Portable across platforms	▪ Unexpected complier behavior ▪ Partial utilization of core features
Intrinsic usage	▪ Can be used in C level ▪ Complier is responsible for local frame, register allocation, parallelism	▪ Code portability is damaged ▪ Requires knowledge of instruction set and architecture ▪ Slower to implement
Assembly level optimization	▪ Potentially reaches optimal performance ▪ Full utilization of core features	▪ Code portability is damaged ▪ Requires deep knowledge of instruction set and architecture ▪ Very slow and tedious

XM4 Compiler Infrastructure

- The Compiler is based on the same platform as other robust and highly optimized CEVA Compilers

- Supports aggressive optimizations targeted at VLIW processors (e.g. SIMD, SWP, inter-procedural analysis, efficient predication, aggressive loop transformation, …)

- Tailored to support vector types and access all of the XM4 architecture capabilities

- Supporting C99, C++98

XM4 Compiler Optimization Levels

- 5 Cycle count optimization levels
  - -O0 to -O4
  - -O4 introduces multiple background compilations
- 5 Code size optimization levels
  - -Os0 to -Os4
  - -Os4 introduces multiple background compilations
- Cycle count and code size levels can be mixed for a better balance between speed and size
- Maximum control can be achieved by tuning optimization levels per function

Vector Operators - Example

- **C level RGB2YUV conversion code**

```c
S32 rgb2yuv_ref( U8 *R, U8 *G, U8 *B, U8 *Y, U8 *U, U8 *V, S32 len )
{
    S32 i;

    for ( i = 0; i < len; i++ )
    {
        Y[i] = ( R[i] * 66 + G[i] * 129 + B[i] * 25 + 128 + 16*256 ) >> 8; // 3 taps calculation
        U[i] = ( B[i] * 126 - Y[i] * 147 + 128 + 128 * 256 ) >> 8;         // 2 taps using Y
        V[i] = ( R[i] * 160 - Y[i] * 186 + 128 + 128 * 256 ) >> 8;         // 2 taps using Y
    }

    return 0;
}
```

Vector Operators – Example – cont.

- **Vector Operators RGB2YUV conversion code**

```
S32 rgb2yuv_vecC( U8 *R, U8 *G, U8 *B, U8 *Y, U8 *U, U8 *V, S32 len )
{
    S32 i;
    uchar32  vR, vG, vB, vY, vU, vV;
    short32 Yacc, Uacc, Vacc;
    uchar32 *pR = (uchar32 *)R, *pG = (uchar32 *)G, *pB = (uchar32 *)B;
    uchar32 *pY = (uchar32 *)Y, *pU = (uchar32 *)U, *pV = (uchar32 *)V;
    for ( i = 0; i < len; i += 32 )
    #pragma dsp_ceva_trip_count_min=8
    #pragma dsp_ceva_trip_count_factor=8
    {
        vR = *pR++;
        vG = *pG++;
        vB = *pB++;
        Yacc  = (short32)vR*66  + (short)(128+16*256);
        vY    = (uchar32)((Yacc + (short32)vB*25 + (short32)vG*129)>>8);
        Uacc  = (short32)vB*126 + (short)(128+128*256);
        vU    = (uchar32)((Uacc + (short32)vY*(-147 + 128) + (short32)vY*(-
128))>>8);
        Vacc  = (short32)vR*160 + (short)(128+128*256);
        vV   =  (uchar32)((Vacc + (short32)vY*(-186 + 128) + (short32)vY*(-
128))>>8);
        *pY++ = vY;
        *pU++ = vU;
        *pV++ = vV;
    }
    return 0;
}
```

```
VPU0.vmac3 v22.uc32, r9.c, v22.uc32, r9.c, v33.s16, v25.s16, #0x8, v9.c32
||  LS0.vld (r24.ui).i8 +#32, v12.i8
||  VPU1.vmpyadd v2.uc32, r31.uc, r11.s, v25.s16, v27.s16
LS0.vst v7.c32, (r27.ui).c32+#32
||  VPU0.vmac3 v14.uc32, r10.c, v14.uc32, r29.c, v30.s16, v31.s16, #0x8,
v19.c32
||  VPU1.vmpyadd v6.uc32, r31.uc, r11.s, v30.s16, v31.s16
VPU0.vmac3 v14.uc32, r9.c, v14.uc32, r9.c, v38.s16, v39.s16, #0x8, v14.c32
||  VPU1.vmpyadd v29.uc32, r31.uc, r11.s, v24.s16, v29.s16
||  LS0.vst v4.c32, (r27.ui).c32+#32
VPU1.vmac3 v11.uc32, r9.c, v11.uc32, r9.c, v24.s16, v29.s16, #0x8, v2.c32
||  LS0.vst v9.c32, (r7.ui).c32+#32
||  VPU0.vmpyadd v23.uc32, r12.c, r11.s, v38.s16, v39.s16
VPU0.vmac3 v11.uc32, r10.c, v11.uc32, r29.c, v36.s16, v37.s16, #0x8,
v3.c32
||  LS0.vst v14.c32, (r7.ui).c32+#32
||  VPU1.vmpyadd v5.uc32, r31.uc, r11.s, v36.s16, v37.s16
```

Vec-C Intrinsics

- **Vector-C Intrinsic are built in functions which are mapped to specific assembly instructions**
- **Fully optimized by the Compiler:**
  - Instruction scheduling
  - Register allocation
  - Loop optimization
  - Redundancy and dead code elimination
- **Enables near-assembly performance from within C/C++ code**

Outline

- What's application you can use CEVA XM4 in RK1108
- XM4 arch introduction
- XM4 SDT introduction
- XM4 program hints
- Develop your own algorithms on CEVA XM4

To develop your own algorithms, you should know

1. Type of algorithms

- Image/Video Processing
  - ISP Pre-processing
  - ISP Post-processing
  - Video encoder pre-processing
- Sensor Processing
  - Sensor fusion(Gyro, G-Sensor)
- Visual perception/Computational photography
  - Face detection/recognition
  - ADAS
  - License plate recognition
- Audio Processing
  - Voice control
  - Speech recognition

To develop your own algorithms, you should know

2. Complexity of algorithms

- Complexity of computing
  - Performance Comparison of different platform
    - If the algorithms were ported on ARM/MIPS/TI-DSP before, it is easier to do the same thing on CEVA DSP.
    - But it’s difficult to evaluate the performance gap if you have only PC-based algorithm implementation.
- Memory Bandwidth
  - Total 3.2GBytes bandwidth@800Mhz DDR.
  - 2GBytes bandwidth available.

To develop your own algorithms, you should know

3. Algorithms optimization

- Writing Vector-C or using optimized OpenCV in key modules
  - Image Registration
    - Surf, Fast-9, RANSAC…
  - Image Filtering
    - Gaussian, Bilateral, Mean, Average, Non-Local…
  - Motion Tracking
    - SAD, Kalman, KLT…
  - Deep Learning
    - CNN, DNN
- Pipeline the DSP calculation and DMA transfer
- Using Motion Vector generated by Video Encoder in RK1108

To develop your own algorithms, you should know

3. Example

- Video Noise Reduction
- HDR

Thank You!

Rockchip
瑞芯微电子