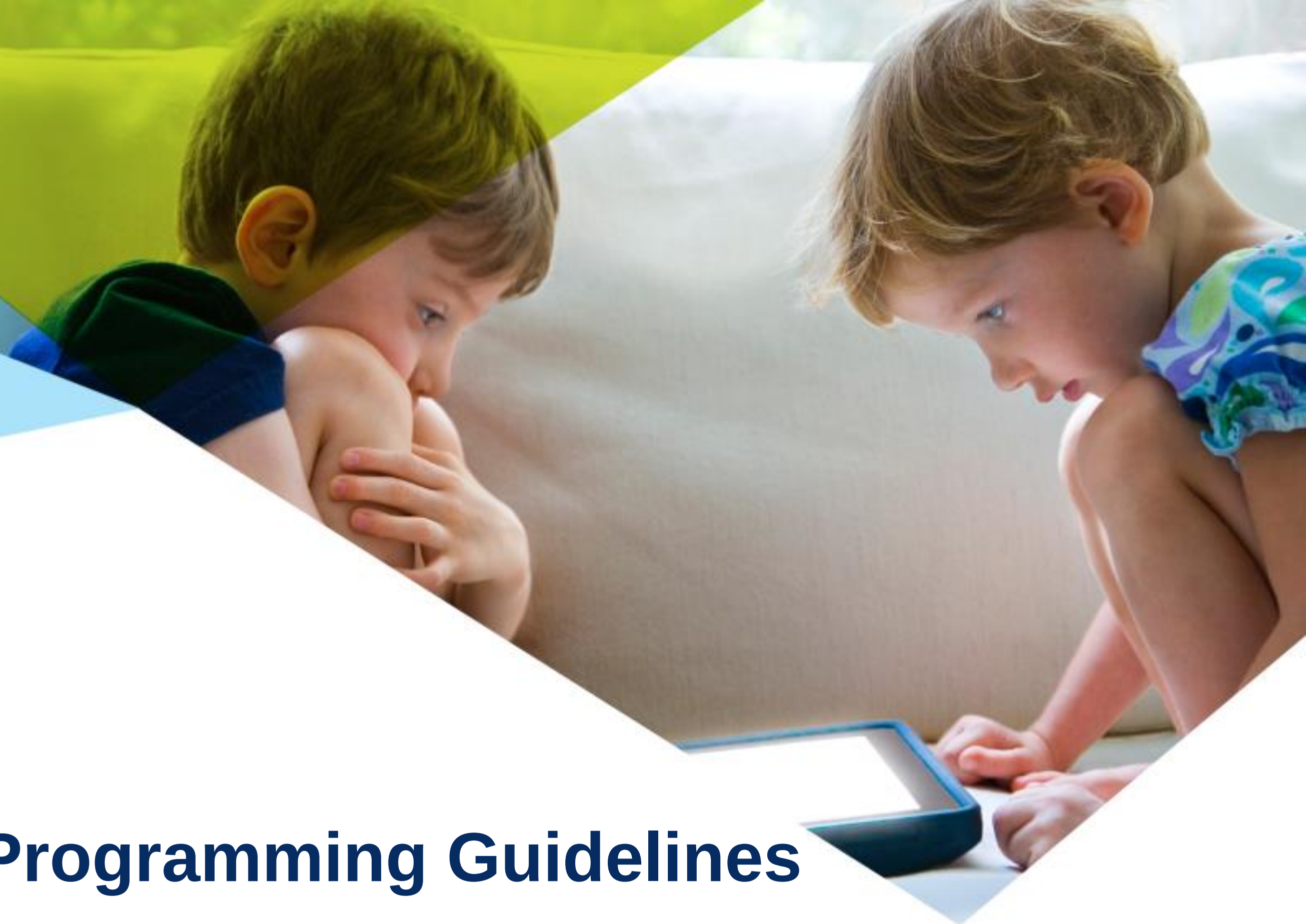

CEVA®
The DSP Powerhouse

# CEVA-XM4 C Programming Guidelines

2015

www.ceva-dsp.com

# Objectives

- Create optimized, high performance applications
  - Minimization of cycle count
  - Concentrate on loops and cycle intensive functions
- Create small, compact code
  - Minimization of code memory footprint
  - Concentrate on large functions that do not have a significant effect on cycle count

# Why Applying Programming Guidelines?

- Compiler has many compilation options that yield different results for different pieces of code
- Optimization hints supplied by the programmer in the code may “guide” the complier in the optimization process
- Many algorithmic implementations perform much better using instruction set special features that cannot be represented by ANSI C
- SIMD/Vector utilization in ANSI C code is not sufficient

→ **Code enhancement can be dramatic (better in factors)**

# Optimization Stages

- Profiling the C level code
  - In order to proceed to next optimization stage perform a profiling stage and check if further optimization is needed
- Tune the Compiler switches
  - Plain C level optimization
  - Modify the code to help the complier make the right decisions
- Hint the compiler on possible optimizations (pragmas, attributes,…)
- Vectorize C code + VEC-C/Intrinsics usage
- Assembly coding

# Optimization Stages - Introduction

| | Advantage | Disadvantage |
|---|---|---|
| C level optimization | ▪ Simple to perform<br>▪ Very fast to implement<br>▪ Easier to reach bit exact results<br>▪ Most optimization products are portable cross platform | ▪ Sometimes it is hard to know how the complier will behave – trial and error<br>▪ Control code is harder to implement<br>▪ Can hardly use SIMD operations |
| VEC-C<br>+ Intrinsics usage | ▪ Can use a C level function to implement<br>▪ Intrinsics use local variable as an input (not registers)<br>▪ Complier is responsible for local frame (local variables), register allocation, parallelism<br>▪ Can use defines and macro defined in C level | ▪ Code portability damaged<br>▪ Need good knowledge of instruction set and architecture<br>▪ Slower to implement |
| Assembly level optimization | ▪ Yields the best performance improvement<br>▪ Features that can be used only in ASM level – variadic sized cyclic buffer, predicated ret instructions, multiple ret instructions | ▪ Code portability damaged<br>▪ Writing the code from scratch<br>▪ Need very good knowledge of instruction set and architecture<br>▪ Very slow and tedious<br>▪ Can't use defines and macro defined in C level |

# Analyzing the Compiler Generated Assembly

- C level optimization is often more like a trial & error process
- User should review the profiling report on each trial in order to inspect the results
- User should also learn how to read the Compiler generated assembly file in order to:
  - Identify the inefficiency spots in the code
  - Pick the right C level optimization for the case inspected
  - Review the resulting code even before profiling
- Compiler assembly generated (.s) file is kept when using *-save-temps* compilation option

# Compiler Generated Assembly Structure

## Function structure:

```
.GLOBAL  _foo
_foo:
              ; gra_spill_temp_0 = -44
              ; …
              ; lgra_spill_temp_2 = -52
              ; …
              ; lra_spill_temp_5 = -64
              ; …
.func_start   3 _foo
… FUNCTION CODE …
; PU cycle count:     2165.722900
; PU code size:       0x1fc
.func_end   3 _foo
```

**Function declaration**

**Function register spills**

Too Many register spills are usually an indication that Compiler had trouble allocating efficiently registers for all variables in the function

**Function profiler markers**

**Function estimated cycles & size**

# Compiler Generated Assembly Structure

CEVA®

- Function code is arranged in basic blocks of sequential assembly code by the following rules:
  - Any branch type instruction (call/ret/brr/..) or pipeline break (e/o bkrep) must end a block
  - Any live code label (target of branch) must start a block
  - Inline assembly instructions get their own dedicated block

### Basic blocks structure

```
; BB1 cycle count: 4
BB1_foo:         ; 0x0
; 288  int  foo(int  *a)  {
        LS0.push {dw}  a12  ;;
; 289               int sum;
        SC.mov a0,  r1
||      LS0.pushd {2dw}  a10,  a11 ;;
        LS0.pushd {2dw}  a8,  a9 ;;
        SC.shifts r1,   #1,   modu0  ;;
; BB1 cycle count: 4
; BB1 code size: 0xe
```

Basic block label and estimated offset from beginning of function

Basic block estimated cycles & size

**Assembly generated code:**

**|| - parallel packet indication**

**;; - end of packet indication**

# Compiler Generated Assembly Structure

CEVA®

- Typically, users mostly care about loops cycle count and therefore basic blocks within loops have additional important indications:
  - Nesting level
  - Unrolling factor
  - Remainder loop (as a result of unrolling of the original loop)

```
        SC.mov a0,  r1
        SQ.bkrep lci0 ;;
; BB3 cycle count: 12
; BB3 code size: 0x34
        {  ; bkrep start - level 0
; <lentry>
; BB13 cycle count: 4
BB13_foo:        ; 0x46
;<loop> unrolled 4 times
        SC.mov a0,  r1
        …
; BB13 cycle count: 4
; BB13 code size: 0x28
        } ; bkrep end - level 0
```

**Bkrep start indication at the end of basic block including nesting level**

**Bkrep end indication at the end of basic block including nesting level**

**Loop entry block may include the following**

**<lentry >**

loop entry indication

**<loop>**

unrolling factor indication

**<loop remainder>**

maximal number of iterations

# Keep it small and simple

- Large functions might stress register allocation and cause register spills to memory
- Might reduce code size in many cases
- How?
  - Try to split large functions into small and simple ones
  - When splitting a function, take out a sequence of code in a way that the overhead of the call will be as minor as possible
  - Break dependencies

# Function Inlining

- Functions that are already compact might benefit from being integrated into the code of the calling function
- Especially beneficial when applied to the program’s critical path
- How?
  - Usage: -INLINE:<switches>
  - Switches:
    - ={on|off}
    - none
    - all
    - must=routine_name[,routine_name]*
    - never=routine_name[,routine_name]*
    - Static={on|off}

**Examples:**

xm4cc main.c –INLINE:never=foo:must=bar

xm4cc main.c –INLINE:static=on

# Minimize Function Arguments

- The Compiler passes the first 8 arguments on registers
- Additional parameters are passed on the stack triggering costly overhead
- How?
  - Functions that require many arguments could receive a pointer to a struct that contains them
  - Global variables can be used in certain cases

# Minimize Function Arguments – Example

**Instead of:**

```
void init_func(int n,
               short lim,
               int x,
               int y,
               int z,
               short *p1,
               short *p2,
               int *p3,
               int *p4,
               int *p5);
```

**Use:**

```
typedef struct {
    int n;
    short lim;
    int x;
    int y;
    int z;
    short *p1;
    short *p2;
    int *p3;
    int *p4;
    int *p5;
} params_t;

void init_func(params_t *args);
```

Minimize number of parameters – reduce function call overhead!

# Mix of Optimization Levels

- Apply the most effective optimization levels to each file
- Most aggressive for cycle count: –O4 –Os0
- Most aggressive for code size: –O3 –Os4
- How?
  - Use -O4 for critical code (kernels) in order to get best performance
  - Use -O3 -Os[1-4] for non-critical code, according to profiling information

# Variables in Computation Intensive Code

- It is preferable to assign a register to a variable that is used in computation intensive code sequences
- The compiler cannot assign a register to a variable in the following cases:
  - Address taken variable (&var)
  - Global variable
  - Static variable
- How?
  - Avoid the above variables for critical code sequences
  - In cases that one of the above is required, copy the value to a local variable for computation, and copy back at the end of the computation
  - Use local variables instead of small arrays.

# Variables in Computation Intensive Code – Example

**Instead of:**

```
int global_counter;

void func(int *p)
{
        int i;

        for (i=0; i<N; i++)
        {
                foo();
                global_counter++;
        }
}
```

**Use:**

```
int global_counter;

void func(int *p)
{
        int i;
        int local_counter=0;
        for (i=0; i<N; i++)
        {
                foo(); // foo() doesn't access global_counter
                local_counter++;
        }
        global_counter += local_counter;
}
```

**Minimize use of address taken, static & global variables!**

# Minimize Loop Content

- Some loops contain code that is not dependent on the loop
- Unnecessary code can damage performance severely, e.g. if-else statements, memory accesses
- How?
  - Remove any non-dependent code from loops
  - Memory accesses that do not change throughout the loop should be copied to local variables
    - and copied back, if necessary

# Set Known Limits to Loops

- Loop limits that may change each iteration prevent the compiler from generating ‘bkrep’ loops
  - ‘bkrep’ → block repeat (zero overhead loop)
- Examples of such limits are: global memory, function calls, complicated calculations
- How?
  - Copy the loop limit to a local variable in any of the above cases
  - Simplify the condition of the loop as much as possible

# Set Known Limits to Loops - Examples

CEVA

**Instead of:**

```
for (i=0; i<foo(); i++)
{
        …
}
```

**Use:**

```
int limit = foo();
for (i=0; i<limit; i++)
{
        …
}
```

**Instead of:**

```
while ((*p != 0) && (i<200))
{
        …
        i++;
}
```

**Use:**

```
while ( i<200 )
{
        if (*p == 0)
           break;
        …
        i++;
}
```

**Minimize use of address taken, static & global variables!**

# Use intrinsics

- The Compiler provides the programmer with access to almost all the instructions in the Architecture Instruction Set via the extended c language
- How?
  - Include the required header files:
    - #include <vec-c.h>   - for CEVA-XM4 intrinsics
  - Use the intrinsics instructions as macros with C variables
  - Use Special intrinsic e.g ffb/countbits etc.

# Control Unrolling of Loops

- The loop optimization process is based on internal Compiler heuristics when the number of iterations is unknown
- The heuristic information can be very different from the actual number of iterations
- Often the programmer has information on the loop that can help the optimization process
- How?
  - Supply additional loop info through pragma directives:
  - #pragma dsp_ceva_unroll=<n>
    - Tells the compiler to unroll the loop N times
  - #pragma dsp_ceva_trip_count =<n>
    - Tells the compiler that the estimated trip count is N
  - #pragma dsp_ceva_trip_count_factor =<n>
    - Tells the compiler that the number of iterations is divisible by N
  - #pragma dsp_ceva_trip_count_min =<n>
    - Tells the compiler that the loop iterates at least N times

# Control Unrolling of Loops - Example

## C code:

```
void foo(int* in, int* out, int N)   {
      int i = 0;
      for(i=0; i<N; i++)
      #pragma dsp_ceva_unroll=1
      {
            *out = *in++;
            out++;
      }
}
```

## Generated code without unroll:

```
; Guarding if may be created in
; cases where software pipeline
; optimization occurs
…
; Loop Body
PCU.bkrep {ds1} lci0.ui
SC0.nop
{
      LS0.ld (r3.ui).i +#4, modu0.i
      LS0.st modu0.ui, (r4.ui).i+#4
}
```

# Control Unrolling of Loops – Example (cont.)

## C code:

```
void foo(int* in, int* out, int N)   {
      int i = 0;
      for(i=0; i<N; i++)
      #pragma dsp_ceva_unroll=2
      {
            *out = *in++;
            out++;
      }
}
```

## Generated code with unroll 2:

```
; Guarding if may be created in
; cases where software pipeline
; optimization occurs
…
SC0.cmp {le} modu0.i, #0, pr0
        ||  SC1.mov r0.i, r4.i
        ||  SC2.mov r1.i, r3.i
        ||  SC3.shiftr r5.i, #0x1, r1.i
        SC0.nop
        PCU.brr {t} #BB18_foo ,?pr0
BB11_foo: ; Reminder loop
        LS0.ld (r4.ui).i +#4, modu0.i
        ||  SC0.shiftr r5.i, #0x1, r1.i
        LS0.st modu0.ui, (r3.ui).i+#4
…
; Loop Body
PCU.bkrep lci0.ui
{
        LS0.ld (r4.ui).i +#4, modu1.i
        LS0.st modu1.ui, (r3.ui).i+#4
        LS0.ld (r4.ui).i +#4, modu0.i
        LS0.st modu0.ui, (r3.ui).i+#4
}
```

# Control Unrolling of Loops – Example (cont.)

## C code:

```
void foo(int* in, int* out, int N)   {
	int i = 0;
	for(i=0; i<N; i++)
	#pragma dsp_ceva_unroll=2
	#pragma dsp_ceva_trip_count_min=2
	{
		*out = *in++;
		out++;
	}
}
```

## Generated code with unroll 2 with minimal trip count of 2:

```
; Guarding if will not be created in this case
…
SC0.cmp {le} modu0.i, #0, pr0
	||  SC1.mov r0.i, r4.i
	||  SC2.mov r1.i, r3.i
	||  SC3.shiftr r5.i, #0x1, r1.i
	SC0.nop
	PCU.brr {t} #BB18_foo ,?pr0
BB11_foo: ; Reminder loop
	LS0.ld (r4.ui).i +#4, modu0.i
	||  SC0.shiftr r5.i, #0x1, r1.i
	LS0.st modu0.ui, (r3.ui).i+#4
…
; Loop Body
PCU.bkrep lci0.ui
{
	LS0.ld (r4.ui).i +#4, modu1.i
	LS0.st modu1.ui, (r3.ui).i+#4
	LS0.ld (r4.ui).i +#4, modu0.i
	LS0.st modu0.ui, (r3.ui).i+#4
}
```

# Control Unrolling of Loops – Example (cont.)

## C code:

```
void foo(int* in, int* out, int N)   {
      int i = 0;
      for(i=0; i<N; i++)
      #pragma dsp_ceva_unroll=2
      #pragma dsp_ceva_trip_count_min=2
      #pragma dsp_ceva_trip_count_factor=2
      {
            *out = *in++;
            out++;
      }
}
```

## Generated code with unroll 2 with minimal trip count of 2, and trip count factor of 2:

```
; Loop Body
PCU.bkrep lci0.ui
{
      LS0.ld (r4.ui).i +#4, modu1.i
      LS0.st modu1.ui, (r3.ui).i+#4
      LS0.ld (r4.ui).i +#4, modu0.i
      LS0.st modu0.ui, (r3.ui).i+#4
}
```

# Supply Memory Aliasing Information

- The Compiler performs extensive memory analysis, in order to optimize scheduling of memory instructions
- Pointers that are received as function arguments are unknown to the Compiler, must be assumed as aliased (= possibly overlapping)
- If it known to user that pointers passed as arguments do not overlap, this information should be passed to the Compiler
- How?
  - Use the command line options:
  - ***-OPT:alias=restrict***
    - All pointers point to distinct memory
  - ***-OPT:alias=strongly_typed***
    - Pointers of different types point to distinct memory
  - Each pointer can also be assigned the attribute __***restrict***__ in the function prototype

# Supply Memory Aliasing Information - Example

## Original C code:

```
…
    for (i=0; i<n; i++)
    {
        *p1=*p2+80;
        p1++;
        p2++;
    }
…
```

## Methods:

```
void vec_mem_copy(        int *__restrict__ p1,
                          int *__restrict__ p2,
                          int n)
{   …
```

**OR**

**xm4cc -OPT:alias=restrict file.c**

Memory aliasing information dramatically reduces loop cycles!

# Supply Memory Aliasing Information - Example

## Without aliasing switches:

```
…
; 5 cycles per iteration
PCU.bkrep lci0.ui
{
    LS0.ld (r4.ui).i +#4, modu3.i
    SC0.nop
    SC0.nop
    SC0.add modu3.i, #80, modu3.i
    LS0.st modu3.ui, (r3.ui).i+#
    …
    ; Additional unrolled iterations
}
```

## Using last slide methods:

```
…
; 1 cycle per iteration
PCU.bkrep lci0.ui
{
    LS0.st r2.ui, (r3.ui).i+#4
    || LS1.ld (r4.ui).i +#4, r2.i
    || SC0.add r0.i, #80, modu2.i
    …
    ; Additional unrolled iterations
}
```

Memory aliasing information dramatically reduces loop cycles!

# Local Variable Types

- Architecture supports native 32-bit registers and computations
- Using 16-bit variables may require redundant instructions for sign/zero extension
- How?
  - Default local variables should be ‘int’ type, especially when used as iterators or array indexes

**C code:**

```
int/short i;
 for(i=0; i<n; i++)
{
      *a=i;
      a++;
}
```

**Using short:**

```
      SC0.add r2.i, #1, modu0.i
||    LS0.st r2.ui, (r0.ui).i+#4
      SC0.extract modu0.i, #0x10, r2.i
```

**Using int:**

```
      LS0.st r3.ui, (r2.ui).i+#4
||    SC0.add r3.i, #1, r3.i
```

# Combine store of same value

- Architecture supports native 32-bit registers and memory access
- How?
  - Combine ld/st

**C code:**

```
void foo(short *a) {
        int i;
        for (i=0;i<128;i++)
                a[i]=0;
}
```

**Generated code:**

```
PCU.bkrep {ds1} #0x7f
SC0.nop
{
        LS0.st r3.ui, (r2.ui).s+#4
||      LS1.st r3.ui, (r1.ui).s+#4
        LS1.st r3.ui, (r1.ui).s+#4
||      LS0.st r3.ui, (r2.ui).s+#4
}
```

```
void foo(short *a) {
        int i;
        int *b=(int *)a;
        for (i=0;i<128/2;i++)
                *b++=0;
}
```

```
PCU.bkrep {ds1} #0x3f
SC0.nop
{
        LS0.st r3.ui, r3.ui, (r2.ui).i2+#16
        LS0.st r3.ui, r3.ui, (r1.ui).i2+#16
}
```

# Use Library Functions

- Code duplication can be avoided by calling existing library functions
- How?
  - Call a library function for copy loops, initialization loops, etc.

**C code:**

```
int i;
for (i=0; i<LIMIT; i++)
{
	x[i] = y[i];
}
for (i=0; i<LIMIT; i++)
{
	arr[i] = 0;
}
```

**Library calls:**

```
memcpy( x,  y, LIMIT*sizeof( int ) );

memset( arr, 0, LIMIT*sizeof( int ) );
```

# Merge Similar Loops

- Loops are performed efficiently, but require overhead to calculate the number of cycles and prepare the loop mechanisms
- How?

**Instead of:**

```
int i;
for (i = 0; i < LIMIT; i++)
{
      *ptr = round(*ptr);
       ptr++;
}
ptr -= LIMIT;
for (i = 0; i < LIMIT; i++)
{
      *ptr = add(ADDITION, *ptr);
      ptr++;
}
```

**Write:**

```
for(i = 0; i < LIMIT; i++)
{
      *ptr = round(*ptr);
      *ptr = add(ADDITION, *ptr);
       ptr++;
}
```

# Minimize if-else Statements

- If-else statements require predicated instructions
- Code that is executed unconditionally can be more compact
- How?

**Instead of:**

```
if (y > 0)
{
      x = 1;
}
else
{
      x=0;
}
```

**Write:**

```
x=0;
if (y > 0)
{
      x = 1;
}
```

# Minimize if-else Statements - Caution!

- Be sure to check that functionality is kept

```
if (y > 0)
{
      x ++;
}
else
{
      x=0;
}
```

≠

```
x=0;
if (y > 0)
{
      x ++;
}
```

# Control Unrolling of Loops

- In high code size optimization levels loop unrolling is strictly limited
- In high cycle count optimization levels, loop unrolling can be controlled to further reduce code size
- For minimal code size the Compiler should avoid any loop unrolling (= code replication)
- How?
  - Use the command line option for all loops in the file:
    - -OPT:unroll_times_max=1
  - Use the pragma to avoid loop unrolling of a specific loop:
    - #pragma dsp_ceva_unroll=1

CEVA®
The DSP Powerhouse

# THANK YOU

www.ceva-dsp.com