Agenda

- Ittiam Introduction
- Cortex A8 Architecture
- Product Development Cycle – Traditional Approach
- Disadvantages of Traditional Approach
- Vectorizing and Non Vectorizing Compilers
- Optimizations in Non-Vectorizing Compiler – Basic
- Optimizations in Non-Vectorizing Compiler – Advanced
- Optimizations in Vectorizing Compiler
- Results
- Conclusion
Introduction to Ittiam Systems

**what we do**
- Embedded Software IP
- Silicon IP

**who we are**
- Digital Signal Processing (DSP) Systems Company
- Embedded Applications in Media Processing & Communication
- Business Model of IP Licensing

**our history**
- Founded in 2001; venture capital of $12.5M raised till date.
- Rated the world’s most preferred DSP IP supplier four years in a row
- 190+ customers worldwide, ~80% OEM and ~20% Semiconductor.
- Red Herring 100 Asia Award 2005.
- Red Herring 100 Global finalist 2007.
- 225 people on board in 2009.
- 40+ patents filed till date.
- 6 Patents awarded

**our customers**
- Customer: Electronics OEM Company
- Customer: SC Company
Ittiam Offerings

- Production Quality System Design
- Reference Design
- Application Software
- Framework / Middleware
- Basic Codecs + Components
- OS + BSP
- HW Platform (DSP, SoC and/or ARM + HW Accelerator)

Ittiam is supporting key customers with Reference Design/Application.

Ittiam is supporting a worldwide customer base on a wide range of digital media & communication applications with this offering.

These are critical components for system quality and performance. One need to be expert to maximize Silicon capability by optimizing Algorithm. Application specific codecs also enables competitive advantages.
## Ittiam Video Codecs on ARM

<table>
<thead>
<tr>
<th></th>
<th>ARM9E</th>
<th>ARM11</th>
<th>Cortex-A8</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIVX</td>
<td></td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>H.263 Baseline</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>H.264 BP</td>
<td></td>
<td>D, E</td>
<td></td>
</tr>
<tr>
<td>H.264 MP</td>
<td></td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>H.264 BP / MP Multi-channel</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H.264 MP / HP</td>
<td></td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>JPEG Baseline / MJPEG</td>
<td>D, E</td>
<td>D, E</td>
<td>D, E</td>
</tr>
<tr>
<td>MPEG2 MP @ ML</td>
<td></td>
<td></td>
<td>In Dev</td>
</tr>
<tr>
<td>MPEG4 ASP</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>MPEG4 SP</td>
<td>D</td>
<td>D</td>
<td>D, E</td>
</tr>
<tr>
<td>MPEG4 SP Multi-channel</td>
<td>D</td>
<td>D</td>
<td>D, E</td>
</tr>
<tr>
<td>WMV9 / VC1 SP / MP</td>
<td></td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>WMV9 / VC1 SP / MP / AP</td>
<td></td>
<td></td>
<td>In Dev</td>
</tr>
</tbody>
</table>
## Ittiam Audio Codecs on ARM

<table>
<thead>
<tr>
<th>Codec</th>
<th>ARM9E</th>
<th>ARM11</th>
<th>Cortex-M3</th>
<th>Cortex-A8</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAC-LC</td>
<td>D, E</td>
<td>D, E</td>
<td>D</td>
<td>D, E</td>
</tr>
<tr>
<td>AAC-LD</td>
<td>D, E</td>
<td>D, E</td>
<td></td>
<td>D, E</td>
</tr>
<tr>
<td>HE-AAC v1, v2 (eAAC+)</td>
<td>D, E</td>
<td>D, E</td>
<td></td>
<td>D, E</td>
</tr>
<tr>
<td>MP3</td>
<td>D, E</td>
<td>D, E</td>
<td>D, E</td>
<td>D, E</td>
</tr>
<tr>
<td>BSAC</td>
<td></td>
<td></td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>FLAC</td>
<td></td>
<td></td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>Ogg Vorbis</td>
<td></td>
<td></td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>WMA</td>
<td>D, E</td>
<td>D, E</td>
<td>D</td>
<td>D, E</td>
</tr>
<tr>
<td>WMA lossless</td>
<td></td>
<td></td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>WMA Pro LBR</td>
<td></td>
<td></td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>MP2 5.1</td>
<td></td>
<td></td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>AAC-LC 5.1</td>
<td>D, E</td>
<td>D, E</td>
<td></td>
<td>D, E</td>
</tr>
<tr>
<td>Dolby/AC-3 5.1 *</td>
<td>D, E</td>
<td>D, E</td>
<td></td>
<td>D, E</td>
</tr>
<tr>
<td>Enhanced AAC+ 5.1</td>
<td>D, E</td>
<td>D, E</td>
<td></td>
<td>D, E</td>
</tr>
<tr>
<td>WMA Pro 5.1/7.1</td>
<td></td>
<td></td>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>

* Dolby/AC-3 5.1 Encoder is Consumer Encoder
Cortex A8 - Architecture

- Super scalar architecture
- **Dual issue** capability
- **NEON media engine**, a separate hardware unit that executes advanced SIMD Instructions for accelerating media and DSP applications
- V7 architecture
- 13 stage ARM pipeline
- 10 stage Neon Pipeline
- **Thumb-2** for excellent code density
- **Two level Cache** system: L1 & L2
- Flexibility in Cache Configuration: 128/256/512/1024 KB
Cortex A8 - Salient features

- Dual-issue
  - Can issue up to 2 ARM Integer Instructions per cycle
  - Both pipelines (ALU-0 and ALU-1) are symmetrical
  - ALU-0 and ALU-1 can execute almost all data processing instructions

- Single instruction multiple data (SIMD) instruction
  - Huge Register File that can be utilized as
    - 32 x 64-bit registers (D0 – D31) OR
    - 16 x 128-bit registers (Q0 – Q15)
  - Can process up to 128 bits as 8/16/32/64 bit data types in a single cycle

Efficient use of these capabilities of Cortex-A8 pose a challenge to the developer
Cortex A8 based Multimedia applications

- Home Entertainment
- Mobile Phones
- In Car Entertainment
- Digital Radio
- Portable Media Player
- Camcorder
IP in Multimedia Applications

- Algorithms used
  - Audio processing (MP3, AAC, effects …)
  - Video processing (MPEG4, H.264 …)
  - Image processing (JPEG)
  - Speech processing (GSM AMR, G.726 …)

- Computationally intensive algorithms
- Resource constraints drive the need for highly optimal implementations
- Approach to achieve Optimal performance becomes critical
Product Development Cycle – Traditional Approach

1. Fixed Point C model
2. Cross compilation
3. Profiling
4. Reached Target?
   - Yes: Opt complete
   - No: Performance Analysis
5. Assembly Coding
6. Debugging
Product Development Cycle – Traditional Approach

- Fixed Point C model
- Cross compilation
- Profiling
- Reached Target?
  - Yes: Opt completed
  - No: Performance Analysis
    - Assembly Coding
    - Debugging

Problem Area!!
Disadvantages of Traditional Approach

- Assembly Coding is Laborious
- Not portable across platforms
- Debugging assembly code is cumbersome

Solution:

*Good understanding and efficient usage of the ARM compiler*
Non Vectorizing and Vectorizing Compilers

- **Non vectorizing Compiler**
  - *armcc* compiler
  - Uses only ARM instructions
  - Example: RVCT 3.1

- **Vectorizing Compiler**
  - *armcc –vectorize* compiler
  - Vectorizing version of the ARM compiler
  - Generates NEON Vectorizing instructions directly from C/C++ code
  - Example: RVCT 4.0 Professional
## Optimizations in Non-Vectorizing Compiler – A bird’s eye view

### Basic – Generic optimization techniques:
- Compiler Options
- Intrinsics addition
- Decrementing Loop Counter
- Structure Accesses elimination
- Array to pointer access conversion
- Inlining
- Simplify complex statements
- Avoid stack access
- Unrolling
- Avoid temporary arrays

### Advanced – Cortex A8 specific optimization techniques:
- Re code in C to use SIMD intrinsics
- Re-arrange the data to remove delay slots
- Look for dual issue possibilities

*Computationally intensive Sparse FIR filter is chosen as a Case study to illustrate the techniques*
Case Study – Sparse FIR Filter

- Sparse FIR filter used in Audio Post Processing

```c
for(i = 0; i < N; i++)
{
    *input = *out;
    *out = (input[0]) +
        (input[-tau[0]] * p[0]) +
        (input[-tau[1]] * p[1]) +
        ... +
        (input[-tau[n]] * p[n]);

    out++;  
    input++;  
}
```
Optimizations in Non-Vectorizing Compiler – Basic

- Give Appropriate Compiler Options
  - -Otime
  - -O3

- Add Intrinsics
  - ARM provides intrinsics for basic set of operations defined in dspfns.h
  - Extend the intrinsics to include a wider range of basic ops [32x16 wide(mult32x16in32) and 32x32 long (mult32) are not present in dspfns.h]
Optimizations in Non-Vectorizing Compiler – Basic(2)

- Use Decrementing Loop Counter
  - Ensures that the compiler uses *SUBS* instruction instead of *ADD* and *CMP*

```c
for(i = 0; i < 1024; i++)
{
    ..... 
}
```

```c
for(i = 1023; i >= 0; i--)
{
    ..... 
}
```

- Avoid Structure Accesses in the loops
  - Use temporary variables
  - This avoids multiple loads

```c
for(i = 1023; i >=0; i--)
{
    x[i] = a\rightarrow b[i] + c\rightarrow d[i];
    ..... 
}
```

```c
for(i = 1023; i >=0; i--)
{
    x[i] = temp1[i] + temp2[i];
    ..... 
}
```
Optimizations in Non-Vectorizing Compiler – Basic(3)

- Array to pointer conversion
  - Avoids offset calculation
  - Makes the statements in the loop independent of loop index

```c
for(i = 1023; i >=0; i--)
{
    x[i] = temp1[i] + temp2[i];
    *x++ = (*temp1++) + (*temp2++);
    .......
}
```

- Inline small functions
  - Avoids Function Call Overheads
Optimizations in Non-Vectorizing Compiler – Basic(4)

- Simplify all the complex statements
  - Compiler understands better

```c
for(i = 0; i < 1024; i++)
{
  *x = (a[0]) +
      (mult32(a[1], b[0])) +
      (mult32(a[2], b[1])) +
      (mult32(a[3], b[2]));
  x++;
  a++;
}
```

Note:
`mult32` is equivalent of 32x32 long multiplication

```c
for(i = 0; i < 1024; i++)
{
  int output;
  output = (a[0]) + (mult32(a[1], b[0]));
  output += (mult32(a[2], b[1]));
  output += (mult32(a[3], b[2]));
  *x++ = output;
  a++;
}
```
Optimizations in Non-Vectorizing Compiler – Basic(5)

- Avoid Stack Access
  - Reduce the life of variables if possible so that the compiler need not store them in stack

- Unroll the loops
  - Reduces the branch overheads

- Avoid arrays for storing intermediate results
  - Avoids offset calculation
Case Study – Sparse FIR Filter

- After Basic optimizations

```c
temp_tau0 = -tau[0];
temp_tau1 = -tau[1];
.. temp_taun = -tau[n];
for(j = N - 1; j >= 0; j--)
{
    temp_load = *out;
    temp_load1 = input[temp_tau0];
    temp_p1 = p[0];
    *input = temp_load;
    output = (temp_scr_buf0) + (mult32(temp_load1, temp_p1));
    ...
    temp_load = input[temp_taun_1];
    temp_load1 = input[temp_taun];
    temp_p1 = p[n-1];
    temp_p2 = p[n];
    output += (mult32(temp_load, temp_p1)) + (mult32(temp_load1, temp_p2));
    *out++ = output;
    input++;
}
```

- Avoid array access inside loops
- Decrementing loop counter
- `mult32` is a user defined intrinsic
- Complex statements simplification
- Array access converted to pointer access
Optimizations in Non-Vectorizing Compiler – Advanced

- Re code in C to use SIMD intrinsics

Single MAC operation of the loop in sparse filter:

```c
temp_load = input[temp_tau4];
output += mult32(temp_load, temp_p1);
```

Mac Operation unrolled to facilitate the use of intrinsics:

```c
temp_load[0] = input[temp_tau4 + 0];
temp_load[1] = input[temp_tau4 + 1];
temp_load[2] = input[temp_tau4 + 2];
temp_load[3] = input[temp_tau4 + 3];

temp_load[0] = mult32(temp_load[0], temp_p1[0]);
temp_load[1] = mult32(temp_load[1], temp_p1[1]);
temp_load[3] = mult32(temp_load[3], temp_p1[3]);

output[0] += temp_load[0];
output[1] += temp_load[1];
output[2] += temp_load[2];
output[3] += temp_load[3];
```
Optimizations in Non-Vectorizing Compiler – Advanced(2)

- Use SIMD intrinsics
- Identify delay slots

```c
temp_load[0] = input[temp_tau3 + 0];
temp_load[1] = input[temp_tau3 + 1];
temp_load[2] = input[temp_tau3 + 2];
temp_load[3] = input[temp_tau3 + 3];

temp_load[0] = mult32(temp_load[0], temp_p1[0]);
temp_load[1] = mult32(temp_load[1], temp_p1[1]);
temp_load[3] = mult32(temp_load[3], temp_p1[3]);

output[0] += temp_load[0];
output[1] += temp_load[1];
output[2] += temp_load[2];
output[3] += temp_load[3];

vec_temp_load = vld2_s32(input[temp_tau3]);
/*Delay slot here*/
vec_out64x2_1 = vmlal_s32(vec_out64x2_1, vec_temp_load.val[0], p1_32x4);
vec_out64x2_2 = vmlal_s32(vec_out64x2_2, vec_temp_load.val[1], p1_32x4);

vec_temp_load = vld2_s32(input[temp_tau4]);
/*Delay slot here*/
vec_out64x2_1 = vmlal_s32(vec_out64x2_1, vec_temp_load.val[0], p2_32x4);
vec_out64x2_2 = vmlal_s32(vec_out64x2_2, vec_temp_load.val[1], p2_32x4);
```

- Use SIMD intrinsics
- Identify delay slots
Optimizations in Non-Vectorizing Compiler – Advanced(3)

- Re-arrange the data to remove delay slots
- Look for dual issue possibilities

```
vec_temp_load = vld2_s32(input[temp_tau3]);

/*Delay slot here*/
vec_temp_load2 = vld2_s32(input[temp_tau4]);

vec_out64x2_1 = vmlal_s32(vec_out64x2_1, vec_temp_load.val[0], p1_32x4);
vec_out64x2_2 = vmlal_s32(vec_out64x2_2, vec_temp_load.val[1], p1_32x4);

vec_out64x2_1 = vmlal_s32(vec_out64x2_1, vec_temp_load2.val[0], p2_32x4);
vec_out64x2_2 = vmlal_s32(vec_out64x2_2, vec_temp_load2.val[1], p2_32x4);
```
Vectorizing Compiler – Introduction

- RVCT has *armcc –vectorize*, the Vectorizing version of ARM compiler, which can automatically convert C/C++ code into NEON instructions.

- Only basic instructions available in dspfns.h can be vectorized by the compiler.

- Loops containing other basic ops that are not defined in dspfns.h will be disqualified from vectorization.
Optimizations in Vectorizing Compiler

- Appropriate Compiler Options
  - **-O3** – Highest Optimization Level
  - **-Otime** – Fastest Code
  - **--vectorize** – Indicates the compiler to use vectorizing version
  - **--restrict** – Indicates that the pointers in the code never point to overlapping memory regions
  - **--asm** – The generated assembly code can be viewed by the developer
Simple Code Handling

- Vectorizing compiler does a brilliant job with simple loops!

```c
void add_array(int *restrict x, int *restrict y, int *restrict z, int *restrict a, int *restrict b, int *restrict c)
{
    int i;
    for(i = 0; i < 256*4; i++)
    {
        z[i] = x[i] + y[i];
        c[i] = a[i] + b[i];
    }
}
```

```
add_array PROC
    PUSH   {r4,r5}
    MOV    r5,#0x100
    LDR    r4,[sp,#0xc]
    LDR    r12,[sp,#8]
|L1.1144|
    SUBS   r5,r5,#1
    VLD1.32 {d0,d1},[r0]!
    VLD1.32 {d2,d3},[r1]!
    VADD.I32 q0,q0,q1
    VLD1.32 {d4,d5},[r3]!
    VLD1.32 {d6,d7},[r12]!
    VST1.32 {d0,d1},[r2]!
    VADD.I32 q0,q2,q3
    VST1.32 {d0,d1},[r4]!
    BNE    |L1.1144|
    POP    {r4,r5}
    BX     lr
    ENDP
```
Complex Code Handling

- Loops containing multiplications with 32bit operands can be further optimized using SIMD intrinsics

"Sparse FIR filter with 32x32 multiplications is disqualified from vectorization"

"Sparse FIR filter with 32x32 multiplications replaced by shifts is fully optimized by the compiler"
Non Vectorizing Compiler - Optimization Results for Sparse FIR Filter

MCPS is Million Cycles Per Second
Vectorizing Compiler - Optimization Results for Sparse FIR Filter

MCPS is Million Cycles Per Second
Conclusion

- Optimal Numbers for MCPS intensive modules in the algorithm can be obtained by tuning the C code for the compiler

- Easier Portability across different Platforms

- Better Code Maintenance as there are no hand coded assemblies

- Reduced Time to Market
QUESTIONS
Rohit Bhuva
VP Business Development
Rohit.bhuva@ittiam.com
972 202 0744 Off
214 212 0104 Cell

THANK YOU