IF-CONVERSION FOR A PARTIALLY PREDICATED VLIW ARCHITECTURE

GNU Cauldron 2022

Benoît Dupont de Dinechin, CTO
AGENDA

1. Kalray MPPA Processor and KVX Core
2. KVX Code Generation Features
3. GCC IF-Conversion Framework
4. Extending GCC IF-Conversion
5. First Results and Outlook
KALRAY DPU-BASED ACCELERATION CARD
MPPA® COOLIDGE V1
Block Diagram & Feature List

80 VLIW Application Cores
- 64-bit/32-bit 6-issue VLIW core
- From 600MHz to 1.2 GHz
- 16KB I/D cache with MMU
- IEEE 754 FP16, FP32, FP64 FPU
- Up to 256-bits per cycle Load/Store

80 Tensor Co-processors (per core)
- INT8.32, INT16.64, FP16.32
- Up to 128 MAC equivalent per cycle

Compute Clusters (5)
- +1 Management/Security Core
- 4 MB of Memory / L2 Cache
- 600GB/s bandwidth

2x100GbE Ethernet Interface & Mger
- 8x1/8x10/8x25/4x40/4x50/2x100 GbE
- Jumbo Frame Support (9.6KB)
- Support for PTP/IEEE 1588v2
- Priority Flow Control (PFC), IEEE 802.1Qbb
- Checksum offload Header & Payload
- Hash & Round-robin dispatcher

Security
- Secure boot with authentication & encryption
- TRNG, RSA, Diffie-Hellman,DSA, ECC, EC-DSA and EC-DH acceleration

PCIe Gen4 Interface
- 16-lane PCIe GEN4 Endpoint (EP) or Root Complex (RC)
- Bifurcation up to 8 downstream ports in RC mode
- SR-IOV up to 8 PF / 248 VF
- Address translation and protection
- Up to 2048 MSI-X & 64 MSI
- Support for Hot Plug
- Up to 512 DMAs for multi queues / kernel bypass drivers
- Direct PCIe-to-clusters and PCIe-to-DDR transfers

LPDDR4/DDR4 Interface
- 64-bit DDR4/LPDDR4-3200 channels with sideband/inline ECC
- Up to two ranks per DDR4 Channel
- 2 DDR channels (up to 32GB) with channel interleaving

Cryptography Accelerators (optional)
# MPPA® PROCESSORS

## Product Family

<table>
<thead>
<tr>
<th>Year</th>
<th>AnDEY</th>
<th>BOstAN</th>
<th>COOLIDgE v1</th>
<th>COOLIDgE v2</th>
<th>DOLOMITES $^{(3)}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2012</td>
<td>28 nm</td>
<td>28 nm</td>
<td>16 nm</td>
<td>16 nm</td>
<td>6/5 nm</td>
</tr>
<tr>
<td>2015</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2020</td>
<td></td>
<td></td>
<td>20 TOPS $^{(1)}$</td>
<td>50 TOPS $^{(1)}$</td>
<td>200 TOPS $^{(1)}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4 TFLOPS $^{(2)}$</td>
<td>25 TFLOPS $^{(2)}$</td>
<td>100 TFLOPS $^{(2)}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>190 KDMIPS</td>
<td>190 KDMIPS</td>
<td>380 KDMIPS</td>
</tr>
<tr>
<td>1H 2023</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2025</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Specify the table.

### PROCESS
- **ANDEY**: 28 nm
- **BOSTAN**: 28 nm
- **COOLIDGE v1**: 16 nm
- **COOLIDGE v2**: 16 nm
- **DOLOMITES**: 6/5 nm

### PERFORMANCE
- **ANDEY**: 1 TOPS
- **BOSTAN**: 1.3 TOPS
- **COOLIDGE v1**: 20 TOPS $^{(1)}$
- **COOLIDGE v2**: 50 TOPS $^{(1)}$
- **DOLOMITES**: 200 TOPS $^{(1)}$

### USE CASES / MARKET
- **ANDEY**: Prototyping
- **BOSTAN**: 40G Data Center Auto Prototypes
- **COOLIDGE v1**: Data Center / Edge Automotive (proto)
- **COOLIDGE v2**: Data Center / Edge Automotive (proto)
- **DOLOMITES**: Data Center Edge Computing 5G

### CONSUMPTION (WATTS)
- **ANDEY**: 25W
- **BOSTAN**: 25W
- **COOLIDGE v1**: 20W $^{(4)}$
- **COOLIDGE v2**: 20W $^{(4)}$
- **DOLOMITES**: 20W $^{(4)}$

### PROTOTYPING / PRODUCTION
- **ANDEY**: AVAILABLE
- **BOSTAN**: UNDER DEVELOPMENT
- **COOLIDGE v1**: UNDER DEVELOPMENT
- **COOLIDGE v2**: UNDER DEVELOPMENT
- **DOLOMITES**: UNDER SPECIFICATION

---

1. INT8.32
2. FP16.32
3. Initial target – may changes
4. 50W maximum compute workload
## VERY LONG INSTRUCTION WORD (VLIW) ARCHITECTURES

Compiler-driven instruction-level parallel execution
Simple, energy-efficient, time-predictable implementations

### CLASSIC VLIW ARCHITECTURE (J. A. FISHER)

<table>
<thead>
<tr>
<th>Key architecture features</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>• SELECT operation on Boolean value</td>
<td></td>
</tr>
<tr>
<td>• Conditional load/store/FPU operations</td>
<td></td>
</tr>
<tr>
<td>• Dismissible loads (non-trapping)</td>
<td></td>
</tr>
<tr>
<td>• [Multi-way conditional branches]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key compiler techniques</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>• Trace scheduling (global instruction scheduling)</td>
<td></td>
</tr>
<tr>
<td>• Partial predication (S. Freudenberger if-conversion)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Main examples</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>• Multiflow TRACE processors</td>
<td></td>
</tr>
<tr>
<td>• HP Labs Lx « Embedded Computing: a VLIW Approach »</td>
<td></td>
</tr>
<tr>
<td>• STMicroelectronics ST200 (media processor based on Lx)</td>
<td></td>
</tr>
</tbody>
</table>

### EPIC VLIW ARCHITECTURE (B. R. RAU)

<table>
<thead>
<tr>
<th>Key architecture features</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>• Fully predicated ISA</td>
<td></td>
</tr>
<tr>
<td>• Speculative loads (control speculation)</td>
<td></td>
</tr>
<tr>
<td>• Advanced loads (data speculation)</td>
<td></td>
</tr>
<tr>
<td>• Rotating registers</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key compiler techniques</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>• Modulo scheduling (software pipelining)</td>
<td></td>
</tr>
<tr>
<td>• Full predication (R-K algorithm, J. Fang algorithm)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Main examples</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>• Cydrome Cydra-5</td>
<td></td>
</tr>
<tr>
<td>• HP-intel IA64</td>
<td></td>
</tr>
<tr>
<td>• TI C6x DSPs</td>
<td></td>
</tr>
</tbody>
</table>
Kalray VLIW (KVX) architecture is co-designed to appear as an in-order superscalar to compilers

- Every scheduler parallel instruction group is a valid bundle
- No need for vertical or horizontal no-op padding

Vector-scalar ISA

- 64x 64-bit general-purpose registers
- Operands can be single registers, register pairs (128-bit) or register quadruples (256-bit)
- 128-bit/256-bit SIMD instructions by dual-issuing/quad-issuing 64-bit instructions on the ALUS or by using the FPU data-path

DSP capabilities

- Counted or while hardware loops with early exits
- Non-temporal loads (L1 cache bypass / preload)
- Non-trapping memory loads (faulting bytes return 0)

CPU capabilities

- 4 privilege levels (rings), MMU (runs Linux kernel)
- Recursive ISA virtualization (Popek & Goldberg)
AGENDA

1. Kalray MPPA Processor and KVX Core
2. KVX Code Generation Features
3. GCC IF-Conversion Framework
4. Extending GCC IF-Conversion
5. First Results and Outlook
ACCESSCORE® SOFTWARE DEVELOPMENT KIT

A Complete Toolchain & Standard Libraries

**ACCESSCORE® SDK**
- Compiler, Simulator, Debugger & System Trace
  - GCC, GDB, LLVM, QEMU
  - SLEEF, SIMDe

**AccessCore® Runtime**
- Operating Systems & Libraries
  - (Linux / ClusterOS)
  - Exokernel, Open Source
  - POSIX RTOS, Linux, Communication Libs

**Optimized Librairies**
- Deep Learning Mathematics
- Computer Vision
- OPEN CV, BLAS, LAPACK
- CNN Inference Code Gen.

**Standard Programming Environment**
- (C/C++/OpenCL)
- POSIX THREAD, OpenMP
- OpenCL, Eclipse

**KAF™**
- Software framework for offloading numerical, signal and image processing

**KaNN™**
- CNN inference code generator compatible with standard CNN frameworks (Caffe, TensorFlow..)

**3rd Party OS**
- RTOS

**3rd Party Tools**
- Model-Based Development

AccessCore® for a seamless integration
C/C++ COMPILER SUPPORT OF KVX CORE

GCC 10 for lightweight POSIX OS and for Linux (all TLS models)

- Mapping of hardware loops using GCC doloop patterns
- Sched2 does instruction scheduling and instruction bundling
- Most high-gain optimizations apply (such as auto-vectorization)

```c
/*
 * kvx-cos-gcc -O2 -ftree-vectorize vector_add.c -c -fopt-info-vec
 * vector_add.c:4:3: optimized: loop vectorized using 32 byte vectors
 */

void
vector_add(long n, float a[n], float b[n], float c[restrict n])
{
    for (long i = 0; i < n; i++)
    {
        c[i] = a[i]+ b[i];
    }
}
```

- On-going developments in software pipelining (derived from C6x)

```
.align 8
.global vector_add
.type vector_add, @function
vector_add:
    add $r4 = $r0, -1
    cb.dlez $r0? .L1
    ;;       # (end cycle 0)
    compd.leu $r4 = $r4, 6
    srlq $r5 = $r0, 3
    ;;       # (end cycle 1)
    cb.dnez $r4? .L7
    ;;       # (end cycle 3)
    loopdo $r5, .L14
    ;;       # (end cycle 4)

.L4:
    l0.xs $r8r9r10r11 = $r4[$r1]
    ;;       # (end cycle 0)
    l0.xs $r32r33r34r35 = $r4[$r2]
    ;;       # (end cycle 1)
    faddwq $r8r9 = $r8r9, $r32r33
    ;;       # (end cycle 4)
    faddwq $r10r11 = $r10r11, $r34r35
    ;;       # (end cycle 5)
    so.xs $r4[$r3] = $r8r9r10r11
    add $r4 = $r4, 1
    ;;       # (end cycle 8)
    # loopdo end
    ...

.L1:
    ret
    ;;       # (end cycle 0)
.size vector_add, .-vector_add

```

© Kalray SA. Confidential - All Rights Reserved.
CACHE BYPASS LOADS AND NON-TRAPPING LOADS

Reuse of the GCC named address spaces (not available in C++)

```c
short cached(short *p, int i) { return p[i]; }

int bypass(int __bypass *p, int i) { return p[i]; }

long preload(long __preload *p, int i) { return p[i]; }

__int128 speculate(__int128 __speculate *p, int i) { return p[i]; }
```

```assembly
.align 8
.global cached
.type cached, @function

cached:
sxwd $r1 = $r1
;;   # (end cycle 0)
lhz.xs $r0 = $r1[$r0]
ret
;;   # (end cycle 1)
.size cached, -.cached

.align 8
.global bypass
.type bypass, @function

bypass:
sxwd $r1 = $r1
;;   # (end cycle 0)
lwz.u.xs $r0 = $r1[$r0]
ret
;;   # (end cycle 1)
.size bypass, -.bypass

.align 8
.global preload
.type preload, @function

preload:
sxwd $r1 = $r1
;;   # (end cycle 0)
l.d.us.xs $r0 = $r1[$r0]
ret
;;   # (end cycle 1)
.size preload, -.preload

.align 8
.global speculate
.type speculate, @function

speculate:
sxwd $r1 = $r1
;;   # (end cycle 0)
lq.s.xs $r0$1 = $r1[$r0]
ret
;;   # (end cycle 1)
.size speculate, -.speculate
```
EXPLOITATION OF THE VECTOR-SCALAR ARCHITECTURE

128-bit and 256-bit vectors are operated and passed as 64-bit register pairs and quadruples

typedef float float32_t;
typedef float32_t float32x8_t __attribute__((vector_size(8*sizeof(float32_t))));

float32x8_t __attribute__((noinline))
kvx_float32x8_adds(float32x8_t a, float32x8_t b)
{
    return a + b;
}

- Full support of the GCC vector syntax extensions
- Align vectors on register pair/quad on ABI boundaries
- SIMD lane splatting and shuffling rely on the BMM8 (8x8 Bit-Matrix Multiply) operations, exposed as SBMM8 instructions (swapped operands)
- To improve register allocation, vector instructions are kept as machine instruction pairs or quadruples until after register allocation
- Use of partial instruction bundles in output templates, with suitable scheduling type

```
.align 8
.global kvx_float32x8_adds
.type   kvx_float32x8_adds, function

kvx_float32x8_adds:
    make $r5 = 6X0804020188040201
    # (end cycle 0)
    sbmm8 $r4 = $r4, $r5
    # (end cycle 1)
    copys $r8 = $r4
    copys $r9 = $r4
    copys $r10 = $r4
    copys $r11 = $r4
    # (end cycle 2)
    faddwq $r0r1 = $r8r9, $r0r1
    # (end cycle 3)
    faddwq $r2r3 = $r10r11, $r2r3
    ret
    # (end cycle 4)
.size   kvx_float32x8_adds, .-kvx_float32x8_adds
```
SIMDE EMULATION OF X86 BUILTINS

SIMDe translates the x86 builtin functions into native call on x86 (SIMDE_X86_SSSE3_NATIVE) and plain C code on other architectures (SIMDE_VECTORIZE)

Kalray port of SIMDe provides an optimized translation on KVX using the GCC/LLVM KVX builtin functions (SIMDE_KVX_NATIVE)
KVX CONDITIONAL BRANCH TEMPLATES

KVX condition codes live in the general-purpose registers

The "cstore<m>" standard pattern produces 0 or 1 (STORE_FLAG_VALUE)

The "*cbsi" pattern matches a conditional branch depending on the comparison (EQ, NE, LE, LT, GE, GT) of a source register to zero

The KVX can compare two integer or floating-point values and instruction variants negate the result (0 or -1)

```asm
;; example: compw.gtu $r2 = $r0, 4
(define_insn (*cstoresi4*)
  [(set (match_operand:SI 0 ("register_operand") ("r","r"))
        (match_operator:SI 1 ("comparison_operator")
         [(match_operand:SI 2 ("register_operand") ("r","r"))
          (match_operand:SI 3 ("kvx_r_any32_operand") ("r","1"))
        ]))]
  ("compw的同时%=%2,%3")

  [(set_attr ("type") ("alu_tiny,alu_tiny_x"))
   (set_attr ("length") ("4", "8"))]
)

;; example: cb.wmez $r3? .L7
(define_insn (*cbusi*)
  [(set (pc)
        (if_then_else (match_operand 0 ("zero_comparison_operator")
                       [(match_operand:SI 1 ("register_operand") ("r"))
                        (const_int 0 [0])
                       ])
         (label_ref (match_operand 2 ("" [""])
                    (pc)))]
  ("cb.wmez %1,%2")

  [(set_attr ("type") ("bcu"))]
)
```
### KVX CONDITIONAL MOVE TEMPLATES

Conditional moves can be produced in two ways:

The "*cmovsi.df" SET source is an IF_THEN_ELSE that relies on a zero_comparison_operator

Genconfig outputs in insn-config.h

```
#define HAVE_conditional_move
```

The "*cond_exec_movedf" pattern is a COND_EXEC wrapping of a simple SET expression

Genconfig outputs in insn-config.h

```
#define HAVE_conditional_execution
```

---

```
;; example: cmoved.weqz $r16? $r11
(define_insn ("*cmovsi.df")
  [set (match_operand:DF 0 ("register_operand") ("=r,r,r,r"))
   (if_then_else:DF (match_operator 2 ("zero_comparison_operator")
     [set (match_operand:SI 3 ("register_operand") ("r,r,r,r")
        (const_int 0 [0]))]
     (match_operand:DF 1 ("kvx_r_s10 s37 s64 operand") ("r,110,B37,i")
      (match_operand:DF 4 ("register_operand") ("0,0,0,0"))))])
   (""") ("cmoved.wk2 %37 %w0 = %k1")
]

(set_attr ("type") ("alu_thin,alu_thin,alu_thin_x,alu_thin_y"))
(set_attr ("length") ("4, 4, 8, 12"))
)
```

```
;; example: cmoved.wnez $r46? $r45 = 0
(define_insn ("*cond_exec_movedf")
  [cond_exec (match_operator 2 ("zero_comparison_operator")
    [match_operand:SI 3 ("register_operand") ("r,r,r,r")
      (const_int 0 [0])]
    (set (match_operand:DF 0 ("register_operand") ("=r,r,r,r")
      (match_operand:DF 1 ("kvx_r_s10 s37 s64 operand") ("r,110,B37,i")))]
   (""") ("cmoved.wk2 %37 %w0 = %k1")
]

(set_attr ("type") ("alu_thin,alu_thin,alu_thin_x,alu_thin_y"))
(set_attr ("length") ("4, 4, 8, 12"))
)```
KVX CONDITIONAL LOAD AND STORE TEMPLATES

Load instructions format sub-64-bit values with zero or sign extension.

Plain load and store addressing modes include [reg], offset[reg], reg[reg], reg*size[reg]:

Plain loads to 64-bit registers

`; example: ld.dnez $r8? $r9 = -8[$r10]
(define_insn (**cond_exec_loaddf")
[ (cond_exec (match_operator 2 ("zero_comparison_operator")
  [ (match_operand:SI 3 ("register_operand") ("r,r,r"))
   (const_int 0 [0]))
  (set (match_operand:DF 0 ("register_operand") ("=r,r,r"))
   (match_operand:DF 1 ("memsimple_operand") ("c,d,e"))))
] ("ld%V1,w%2z %3? %0 = %01")
[ (set_attr ("type") ("lsu_auxw_load,lsu_auxw_load_x,lsu_auxw_load_y"))
  (set_attr ("length") ("4, 8, 12"))]
])

`; example: sd.dltz $r3? 88[$r4] = $r5
(define_insn (**cond_exec_storedf")
[ (cond_exec (match_operator 2 ("zero_comparison_operator")
  [ (match_operand:SI 3 ("register_operand") ("r,r,r"))
   (const_int 0 [0]))
  (set (match_operand:DF 0 ("memsimple_operand") ("=c,d,e"))
   (match_operand:DF 1 ("register_operand") ("r,r,r"))))
] ("sd%V0,w%2z %3? %00 = %1")
[ (set_attr ("type") ("lsu_auxr_store,lsu_auxr_store_x,lsu_auxr_store_y"))
  (set_attr ("length") ("4, 8, 12"))]
])

Conditional load and store addressing modes are restricted to [reg], offset[reg]:

(define_predicate "memsimple_operand"
  (match_code "mem")
  { return indirect_operand (op, mode)
    | || kvx_has_27bit_immediate_p (op)
    | || kvx_has_54bit_immediate_p (op);
  })

# Plain loads to 64-bit registers
ld $r8 = -8[$r11]
ld.xs $r9 = $r10[$r11]
AGENDA

1. Kalray MPPA Processor and KVX Core
2. KVX Code Generation Features
3. GCC IF-Conversion Framework
4. Extending GCC IF-Conversion
5. First Results and Outlook
GCC AUTOMATED
COND_EXEC TEMPLATES

GCC can automate the writing of COND_EXEC instruction templates with the (define_cond_exec) template and the "predicable" attribute.

The (define_insn) templates with (eq_attr "predicable" "yes") have their RTL template wrapped into a COND_EXEC with the condition supplied by the (define_cond_exec).

The output template of the resulting instructions is prefixed by the output template of the (define_cond_exec).

Custom output may use the current_insn_predicate RTX.

```
(define_cond_exec
  ((match_operator 0 "zero_comparison_operator"
    ((match_operand:SI 1 "register_operand" "r")
     (const_int 0)))))
"" 
"<suffix>\%02 \%1?"
)

;; sample move template with predicable attribute
(define_insn ("move")
  [(set (match_operand:DF 0 ("register_operand") ("=r,r,r,r"))
         (match_operand:DF 1 ("kvx_r_s10_s37_s64_operand") ("r,I10,B37,i")))
   (**)("MOVE \%0 = \%1")
   [(set_attr ("predicable") ("yes"))])

;; resulting conditional execution template
(define_insn ("p *move")
  [(cond_exec (match_operator 2 ("zero_comparison_operator")
     [(match_operand:SI 3 ("register_operand") ("r,r,r,r")
       (const_int 0 [0]))
     )]
   (set (match_operand:DF 0 ("register_operand") ("=r,r,r,r"))
       (match_operand:DF 1 ("kvx_r_s10_s37_s64_operand") ("r,I10,B37,i"))))
   (**)("w%22 \%3? MOVE \%0 = \%1")
   [(set_attr ("ce_enabled") ("yes"))])
```
GCC IF-CONVERSION OVERVIEW (1)

Enabled with –fif-conversion and –fif-conversion2

Three passes:
- CE1 before combine
- CE2 after combine
- CE3 after reload

Information about the if-conversion region is passed with a ce_if_block structure

Top level (if_convert) iterates over if-conversion region header blocks by calling (find_if_header)

```c
static void
if_convert (bool after_combine)
{
  int pass = 0;
  ...
  do
  {
    df_analyze ();
    /* Only need to do dce on the first pass. */
    df_clear_flags (DF_LR_RUN_DCE);
    cond_exec_changed_p = FALSE;
    pass++;

    FOR_EACH_BB_FN (bb, cfun)
    {
      basic_block new_bb;
      while (!df_get_bb_dirty (bb)
        && (new_bb = find_if_header (bb, pass)) != NULL)
        bb = new_bb;
    }

    while (cond_exec_changed_p);
    ...
  }
}
GCC IF-CONVERSION OVERVIEW (2)

(find_if_header)

• Fill the ce_if_block structure
• Call IFCVT_MACHDEP_INIT
• Before reload (CE1 and CE2), call (noce_find_if_block)
• After reload (CE3) and if target has conditional execution, call (cond_exec_find_if_block)

Default target hook for TARGET_HAVE_CONDITIONAL_EXECUTION returns HAVE_conditional_execution

```c
static basic block
find_if_header (basic_block test_bb, int pass)
{
    ce_if_block ce_info;
    ...
    ce_info.test_bb = test_bb;
    ce_info.then_bb = then_edge->dest;
    ce_info.else_bb = else_edge->dest;
    ce_info.pass = pass;

#ifdef IFCVT_MACHDEP_INIT
    IFCVT_MACHDEP_INIT (&ce_info);
#endif

if (!reload_completed
    && noce_find_if_block (test_bb, then_edge, else_edge, pass))
    goto success;

if (reload_completed
    && targetm.have_conditional_execution())
    && cond_exec_find_if_block (&ce_info)
    goto success;
...
if (dom_info_state (CDI_POST_DOMINATORS) >= DOM_NO_FAST_QUERY
    && (reload_completed || !targetm.have_conditional_execution()))
{
    if (find_if_case_1 (test_bb, then_edge, else_edge))
        goto success;
    if (find_if_case_2 (test_bb, then_edge, else_edge))
        goto success;
}

success:
    cond_exec_changed_p = TRUE;
    return ce_info.test_bb;
}
```
GCC IF-CONVERSION OVERVIEW (3)

(noce_find_if_block)

- Determine the if-conversion region: IF-THEN-ELSE-JOIN or IF-THEN-JOIN or IF-ELSE-JOIN
- First try without, then with, using conditional moves

```c
static int
noce_find_if_block (basic_block test_bb, edge then_edge, edge else_edge, int pass)
{
    ...
    struct noce_if_info if_info;
    <recognize IF-THEN-ELSE-JOIN or IF-THEN-JOIN or IF-ELSE-JOIN regions>
    ...
    <initialize if_info>

    if (noce_process_if_block (&if_info))
        return TRUE;

    if (HAVE_conditional_move
        && cond_move_process_if_block (&if_info))
        return TRUE;

    return FALSE;
}

/* Given a simple IF-THEN-JOIN or IF-THEN-ELSE-JOIN block, attempt to convert it without using conditional execution. Return TRUE if we were successful at converting the block. */
static int
noce_process_if_block (struct noce_if_info *if_info)
{
    ...
}

/* Given a simple IF-THEN-JOIN or IF-THEN-ELSE-JOIN block, attempt to convert it using only conditional moves. Return TRUE if we were successful at converting the block. */
static int
cond_move_process_if_block (struct noce_if_info *if_info)
{
    ...
}
```
GCC IF-CONVERSION OVERVIEW (4)

(cond_exec_find_if_block)
- Identify cases of && tests (jump to ELSE block) or || tests (jump to THEN block)
- In case of && or || tests, try to combine then into the conditional expression
- If no or failed on multiple test region, process IF-THEN-ELSE-JOIN etc.

(cond_exec_process_if_block)
- Find common head or tail sequences in IF-THEN-ELSE-JOIN
- Dispatch to (cond_exec_process_insns)

```c
static int
cond_exec_find_if_block (struct ce_if_block * ce_info)
{
    ...
    if (cond_exec_process_if_block (ce_info, TRUE))
        return TRUE;
    if (ce_info->num_multiple_test_blocks)
    {
        cancel_changes (0);
        if (cond_exec_process_if_block (ce_info, FALSE))
            return TRUE;
    }
    return FALSE;
}

static int
cond_exec_process_if_block (ce_if_block * ce_info, int do_multiple_p)
{
    ...
    /* Go through the THEN and ELSE blocks converting the insns if possible to conditional execution. */
    if (then_end
        && (! false_expr
            || ! cond_exec_process_insns (ce_info, then_start, then_end,
                                          false_expr, false_prob_val,
                                          then_mod_ck)))
        goto fail;
    if (else_bb && else_end
        && ! cond_exec_process_insns (ce_info, else_start, else_end,
                                      true_expr, true_prob_val, TRUE))
        goto fail;
    ...
```
GCC IF-CONVERSION OVERVIEW (5)

(cond_exec_process_insns)

- Process instructions from START to END, as there can be matching head and tail sequences in the THEN and ELSE blocks
- If instruction pattern code is already COND_EXEC, build a new condition by ANDing with the block condition
- Generate COND_EXEC pattern
- Call IFCVT_MODIFY_INSN which can modify the pattern or abort if-conversion

```
static int
cond_exec_process_insns (ce_if_block *ce_info ATTRIBUTE UNUSED,
   /* if block information */rtx insn *start,
   /* first insn to look at */rtx end,
   /* last insn to look at */rtx test,
   /* conditional execution test */profile_probability prob_val,
   /* probability of branch taken. *//int mod_ok)
{
   ...
   for (insn = start; ; insn = NEXT_INSN (insn))
   {
   ...
      /* Now build the conditional form of the instruction. */
      pattern = PATTERN (insn);
      xtest = copy_rtx (test);
      ...
      pattern = gen_rtx_COND_EXEC (VOIDmode, xtest, pattern);
      /* If the machine needs to modify the insn being conditionally executed,
       say for example to force a constant integer operand into a temp
       register, do so here. */
      #ifdef IFCVT_MODIFY_INSN
      IFCVT_MODIFY_INSN (ce_info, pattern, insn);
      if (!pattern)
         return FALSE;
      #endif
      validate_change (insn, &PATTERN (insn), pattern, 1);
   ...
   }
   ...
```
AGENDA

1. Kalray MPPA Processor and KVX Core
2. KVX Code Generation Features
3. GCC IF-Conversion Framework
4. Extending GCC IF-Conversion
5. First Results and Outlook
## KVX IF-CONVERSION OBJECTIVES AND CONSTRAINTS

Focus on scalar instructions, as the GIMPLE auto-vectorization takes care of generating masked vector operations.

Complement the if-conversion provided the standard patterns for conditional operations: move<
\text{m}>\text{cc}, \text{add}<\text{m}>\text{cc}, \text{neg}<\text{m}>\text{cc}, \text{not}<\text{m}>\text{cc}

No changes to the target-independent GCC code.

<table>
<thead>
<tr>
<th>PREDICATION OF LOADS AND STORES</th>
<th>PSEUDO-PREDICATION OF INSTRUCTIONS</th>
<th>SPECULATIVE EXECUTION OF INSTRUCTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extend GCC conditional execution to the KVX predicated load and stores instructions that have addressing mode restrictions.</td>
<td>Unconditionally compute the original result into a scratch register then conditionally move the result to the original destination register.</td>
<td>Eliminate the need for computing into a scratch register and conditional move if the destination is only locally used in the THEN or ELSE block.</td>
</tr>
</tbody>
</table>

Can only expose the predicated instructions after register allocation.

Unconditional assignments to scratch registers must not clobber registers in use.
KVX IF-CONVERSION OVERVIEW (1)

Implemented with four target hooks

- MAX_CONDITIONAL_EXECUTE
- IFCVT_MACHDEP_INIT (kvx.h) called in CE1, CE2, CE3 from (find_if_header)
- IFCVT_MODIFY_INSN (kvx.h) called in CE3 from (cond_exec_process_insns)
- TARGET_HAVE_CONDITIONAL_EXECUTION (kvx.c) called in CE1, CE2, CE3

In combination with COND_EXEC patterns and helper patterns in the kvx .md files

```c
#define BRANCH COST(speed, predictable) 6

/* Set MAX_CONDITIONAL_EXECUTE to 2*BRANCH_COST */
#define MAX_CONDITIONAL_EXECUTE 12

#define IFCVT_MACHDEP_INIT(ce info) \ 
  (kvx_ifcvt_machdep_init (ce info, ifcvt_after_combine, &cond_exec_changed_p))

#define IFCVT_MODIFY_INSN(CE INFO, PATTERN, INSN) \ 
  ((PATTERN) = kvx_ifcvt_modify_insn (CE INFO, PATTERN, INSN))

#define TARGET_HAVE_CONDITIONAL_EXECUTION
#define TARGET_HAVE_CONDITIONAL_EXECUTION kvx_haveConditionalExecution

#undef TARGET_HAVE_CONDITIONAL_EXECUTION
#define TARGET_HAVE_CONDITIONAL_EXECUTION kvx_haveConditionalExecution

... 

/* Implements TARGET_HAVE_CONDITIONAL_EXECUTION. */
static bool
kvx_haveConditionalExecution (void)
{
  // Enable NOCE before combine and COND_EXEC after reload.
  return reload_completed;
}
```
KVX IF-CONVERSION OVERVIEW (2)

(kvx_ifcvt_machdep_init)

- Let CE1 and CE2 do if-conversion without conditional execution
- In CE2, prepare for CE3, focusing on IF-THEN-ELSE-JOIN, IF-THEN-JOIN, IF-ELSE-JOIN regions identified with same logic as in (noce_find_if_block)

The idea is to insert USEs and pseudo-DEFs in CE2 so that the CE3 if-conversion will have the spare hard registers it needs for pseudo-predication and speculation

```c
void
kvx_ifcvt_machdep_init (struct ce_if_block *ce_info, bool after_combine,
                          int *cond_exec_changed_p)
{
    kvx_ifcvt_ce_level = reload_completed ? KVX_IFCVT_CE3 :
                              (after_combine ? KVX_IFCVT_CE2 :
                               KVX_IFCVT_CE1);

    // Nothing else to do in CE1 and CE3.
    if (kvx_ifcvt_ce_level != KVX_IFCVT_CE2 || !flag_if_conversion2)
    {
        ...
        return;
    }

    ...
    // Reset context for each if-conversion region.
    kvx_ifcvt_reset_context ();

    // Detect IF-THEN-ELSE-JOIN as in (noce_find_if_block) in ifcvt.c.
    ...
    // Check basic block candidates and prepare for conditional execution. */
    basic_block test_bb = ce_info->test_bb;
    rtx reg_test = kvx_ifcvt_get_reg_test (test_bb);
    if (reg_test & kvx_ifcvt_count-- -> 0)
    {
        bool candidate = true;

        if (candidate & then_bb)
            candidate &= kvx_ifcvt_ce2_candidate_ce3 (then_bb, else_bb, reg_test);

        if (candidate & else_bb)
            candidate &= kvx_ifcvt_ce2_candidate_ce3 (else_bb, then_bb, reg_test);

        <prepare for CE3 if-conversion>

        if (changed)
            *cond_exec_changed_p = true;
    }
}
 KVX IF-CONVERSION
 FIND CANDIDATES (1)

(kvx_ifcvt_ce2_candidate_ce3)

- Scan the non-jump instructions
- Bail-out if complex instruction or instructions with side-effects
- Try conditional moves (with COND_EXEC), if fail will have a second chance as arithmetic
- Try conditional memory accesses (irrespective of addressing mode)
- Try to speculate the non-trapping arithmetic instructions
- Try to pseudo-predicate the non-trapping arithmetic instructions

```c
static bool
kvx_ifcvt_ce2_candidate_ce3 (basic_block block, basic_block other,
rtx reg_test)
{
  int index = block->jump_index;
  if ((unsigned) index >= (unsigned) kvx_ifcvt.block_count
      || bitmap_bit_p (kvx_ifcvt.block_visited, index))
    return false;

  int count = 0;
  rtx insn *insn = 0;
  FOR_BB_INSNS (block, insn)
  {
    if (NONJUMP_INSN_P (insn))
    {
      rtx pattern = PATTERN (insn);
      if (count++ >= MAX_CONDITIONAL_EXECUTE)
        return false;
      if (side_effects_p (pattern))
        return false;
      if (GET_CODE (pattern) != SET)
        return false;

      if (kvx_ifcvt_ce2_cond_move_ce3 (insn, reg_test, block))
        continue;
      if (contains_mem_rtx_p (pattern))
      {
        if (kvx_ifcvt_ce2_cond_mem_ce3 (insn, reg_test, block))
          continue;
      }
      else if (may_trap_p (pattern))
      {
        if (kvx_ifcvt_ce2_spec_arith_ce3 (insn, reg_test, block))
          continue;
        if (kvx_ifcvt_ce2_cond_arith_ce3 (insn, reg_test, block))
          continue;
      }
      return false;
    }
    else if (INSN_P (insn))
      return false;
  }
  return count;
}
```
KVX IF-CONVERSION FIND CANDIDATES (2)

(kvx_ifcvt_ce2_cond_mem_ce3)
- If need a scratch register to compute address, reserve it by wrapping the original pattern and a USE inside a PARALLEL

(kvx_ifcvt_ce2_cond_arith_ce3)
- Similar, except that always wrap with USE of a scratch register that has the mode of destination

(kvx_ifcvt_ce2_spec_arith_ce3)
- If the destination register is only locally used (not live-out), may speculatively execute unchanged

```c
static bool
kvx_ifcvt_ce2_cond_mem_ce3 (rtx_insn *insn, rtx reg_test, basic_block block)
{
    rtx pattern = PATTERN (insn);
    rtx set_src = SET_SRC (pattern);
    rtx set_dest = SET_DEST (pattern);
    enum rtx_code src_code = GET_CODE (set_src);

    // Cases of loads with zero extension or sign extension.
    if ((src_code == ZERO_EXTEND || src_code == SIGN_EXTEND) &
        MEM_P (XEXP (set_src, 0)))
    {
        set_src = XEXP (set_src, 0);
        src_code = MEM;
    }

    rtx mem = 0;
    if (src_code == MEM)
        mem = set_src;
    if (GET_CODE (set_dest) == MEM)
        mem = set_dest;
    if (!mem)
        return false;

    bool memsimple = memsimple_operand (mem, VOIDmode);
    if (!memsimple)
    {
        rtx reg = gen_reg_rtx (Pmode);
        rtx parallel = gen_rtx PARALLEL (VOIDmode, rtvec_alloc (2));
        XVECEXP (parallel, 0, 0) = copy_rtx (pattern);
        XVECEXP (parallel, 0, 1) = gen_rtx USE (VOIDmode, reg);
        kvx_ifcvt.prep_insn[kvx_ifcvt.prep_insns_count].parallel = parallel;
        kvx_ifcvt.prep_insn[kvx_ifcvt.prep_insns_count].block = block;
        kvx_ifcvt.prep_insn[kvx_ifcvt.prep_insns_count].insn = insn;
        kvx_ifcvt.prep_insn[kvx_ifcvt.prep_insns_count].insn_count++;
        pattern = parallel;
    }

    rtx new_pattern = gen_rtx CONDITIONAL (VOIDmode, reg_test, pattern);
    return kvx_ifcvt_ce2_recog_pattern (new_pattern, true);
}
```
KVX PREPARE FOR CE3 IF-CONVERSION (1)

<prepare for CE3 if-conversion> in (kvx_ifcvt_machdep_init)

- Extend the live-range of tested register by inserting its USE at end of THEN and ELSE blocks
- Update pattern of the pseudo-predicated memory and arithmetic insns to the one recognized in (kvx_ifcvt_ce2_cond_mem_ce3) or (kvx_ifcvt_ce2_cond_arith_ce3)

```c
if (then_bb)
{
    // Find the last insertion point in then_bb.
    rtx_insn *last_insn = BB_END (then_bb);
    if (JUMP_P (last_insn))
        last_insn = PREV_INSN (last_insn);

    // Insert a use of the tested register.
    rtx tested_reg = XEXP (reg_test, 0);
    gen_rtx USE (VOIDmode, tested_reg);
    df_set_bb_dirty (then_bb);
}

if (else_bb)
{
    // Find the last insertion point in else_bb.
    rtx_insn *last_insn = BB_END (else_bb);
    if (JUMP_P (last_insn))
        last_insn = PREV_INSN (last_insn);

    // Insert a use of the tested register.
    rtx tested_reg = XEXP (reg_test, 0);
    gen_rtx USE (VOIDmode, tested_reg);
    df_set_bb_dirty (else_bb);
}

// Update the pattern of the pseudo-predicated insns.
for (int index = 0; index < kvx_ifcvt.prep_insn_count; index++)
{
    rtx_insn *insn = kvx_ifcvt.prep_insn[index].insn;
    rtx parallel = kvx_ifcvt.prep_insn[index].parallel;
    basic_block block = kvx_ifcvt.prep_insn[index].block;
    gcc_checking_assert (GET_CODE (parallel) == PARALLEL);
    PATTERN (insn) = parallel;
    INSN_CODE (insn) = -1;
    df_set_bb_dirty (block);
    df_insn_rescan (insn);
    changed = true;
}
```
KVX PREPARE FOR CE3 IF-CONVERSION (2)

<prepare for CE3 if-conversion> in (kvx_ifcvt_machdep_init)

- Flag the speculated instructions with REG_NONNEG note (hack, unused otherwise in this port)
- Insert USE of speculated destination register in JOIN block
- Insert DEFs of scratch registers in TEST block and USEs of scratch registers in JOIN block

These DEFs and USEs prevent the allocation of the same hard registers to the scratch registers in one path and live variables on the other path.
KVX FINALIZE IF-CONVERSION

(kvx_ifcvt_modify_insn)

• Implements target hook IFCVT_MODIFYInsn (CE3)

• Undo the COND_EXEC of pattern by (cond_exec_process_insn) in case of the inserted pseudo-DEFs

• Undo the COND_EXEC of pattern by (cond_exec_process_insn) in case of speculated instructions

```c
rtx
kvx_ifcvt_modify_insn (ce_if_block *ce_info ATTRIBUTE_UNUSED,
                      rtx pattern, rtx_insn *insn)
{
  rtx old_pattern = PATTERN (insn);

  // Ignore (SET (...) UNSPEC_DEF) at this point.
  rtx x = GET_CODE (old_pattern) == SET? SET_DEST (old_pattern) : 0;
  if (x && GET_CODE (x) == UNSPEC && (XINT (x, 1) == UNSPEC_DEF))
    return old_pattern;

  // No changes if the insn was flagged as speculative in CE2.
  for (rtx link = REG_NOTES (insn); link; link = XEXP (link, 1))
    if (REG_NOTE_KIND (link) == REG_NONNEG)
      return old_pattern;

  return pattern;
}
```
COND_EXEC OF MEMORY LOADS

(cond_exec_process_insns) tries to CON_EXEC instruction patterns

• COND_EXEC of loads with the "memsimple" operand predicate must appear first (shown earlier)

• COND_EXEC of loads with a "memory" operand predicate not "memsimple" is not valid, so use a (define_insn_and_split) to simplify the addressing mode.

• In case CE3 fails, provide another (define_insn_and_split) to undo the PARALLEL wrapping done by (kvx_ifcvt_ce2_cond_mem_ce3)

Similar patterns for loading with zero/sign extension

```
;; COND_EXEC LOAD
(define_insn_and_split "*cond_exec_load<ALLIFV:mode>"
[(cond_exec
  (match_operator 2 "zero_comparison_operator"
  [(match_operand:SIDI 3 "register_operand" "r,r,r")
   (const_int 0)])
  (parallel
  [(set (match_operand:ALLIFV 0 "register_operand" "=r,r,r")
     (match_operand:ALLIFV 1 "memory_operand" "a,b,m")))
  (use (match_operand:P 4 "register_operand" "r,r,r"))))]]
"kvx_ifcvt_ce_level => KVX_IFCVT_CE2"
"#"
"kvx_ifcvt_ce_level => KVX_IFCVT_CE3"
[(cond_exec
  (match_op_dup 2
  [(match_dup 3) (const_int 0)])
  (set (match_dup 0) (match_dup 1)))]
{ if (!memsimple_operand (operands[1], VOIDmode))
  { rtx address = copy_rtx (XEXP (operands[1], 0));
    emit_insn (gen_rtx_SET (operands[4], address));
    operands[1] = gen_rtx_MEM (<ALLIFV:MODE>mode, operands[4]);
    gcc_checking_assert (memsimple_operand (operands[1], VOIDmode));
  }
}

(define_insn_and_split "*wrapped_load<ALLIFV:mode>"
[(set (match_operand:ALLIFV 0 "register_operand" "=r,r,r")
   (match_operand:ALLIFV 1 "memory_operand" "a,b,m"))
  (use (match_operand:P 2 "register_operand" "r,r,r")))
"kvx_ifcvt_ce_level => KVX_IFCVT_CE2"
"#"
"kvx_ifcvt_ce_level => KVX_IFCVT_CE3"
[(set (match_dup 0) (match_dup 1))]
```
COND_EXEC OF MEMORY STORES

Similar to the COND_EXEC of memory loads

• COND_EXEC of stores with the "memsimple" operand predicate must appear first (shown earlier)

• Use a (define_insn_and_split) to simplify the addressing mode in case of a "memory" operand predicate which is not "memsimple"

• In case CE3 fails, provide another (define_insn_and_split) to undo the wrapping with PARALLEL done in CE2 by

```assembly
;; COND_EXEC STORE

(define_insn_and_split "*cond_exec_store<ALLIFV:mode>"
  [(cond_exec
      (match operator 2 "zero comparison operator"
        [(match_operand:SIDI 3 "register_operand" "r,r,r")
          (const int 0)])
      (parallel
        [(set (match_operand:ALLIFV 0 "memory_operand" "=a,b,m")
          (match_operand:ALLIFV 1 "register_operand" "r,r,r")
          (use (match_operand:P 4 "register_operand" "r,r,r")))]))]
  "kvx_ifcvt_ce_level >= KVX_IFCVT_CE2"
  
  "kvx_ifcvt_ce_level >= KVX_IFCVT_CE3"
  [(cond_exec
      (match op dup 2
        [(match dup 3) (const int 0)]
        (set (match dup 0) (match_dup 1)))]

  { if (!memsimple_operand (operands[0], VOIDmode))
    {
      rtx address = copy rtx (XEXP (operands[0], 0));
      emit insn (gen_rtx_SET (operands[4], address));
      operands[0] = gen_rtx_MEM (<ALLIFV:MODE>mode, operands[4]);
      gcc_checking_assert (memsimple_operand (operands[0], VOIDmode));
    }
  }
})

(define_insn_and_split "*wrapped_store<ALLIFV:mode>"
  [(set (match operand:ALLIFV 0 "memory_operand" "=a,b,m")
    (match operand:ALLIFV 1 "register_operand" "r,r,r")
    (use (match_operand:P 2 "register_operand" "r,r,r"))]
  "kvx_ifcvt_ce_level >= KVX_IFCVT_CE2"
  
  "kvx_ifcvt_ce_level >= KVX_IFCVT_CE3"
  [(set (match_dup 0) (match_dup 1))]
)
COND_EXEC OF NON-PREDICABLE ARITHMETIC

(cond_exec_process_insns) tries to CON_EXEC the instruction patterns

- As there are no predicated arithmetic instructions in the KVX ISA, pseudo-predicate them
- Use a (define_insn_and_split) to compute into scratch register, then conditionally move it to the original destination
- In case CE3 fails, provide another (define_insn_and_split) to undo the wrapping with PARALLEL done in CE2 by (kvx_ifcvt_ce2_cond_arith_ce3)

Similar patterns are needed for most of the scalar ISA subset

```
;; COND_EXEC WI BINARY
(define_insn_and_split "*cond_exec_binary_<WI:mode>"
 [(cond_exec
   (match operator 5 "zero_comparison_operator"
     [(match operand:SIDI 4 "register_operand" "r,r,r,r")
      (const_int 0)]))
  (parallel
   [(set (match operand:WI 0 "register_operand" "=r,r,r,r")
     (match operator:WI 3 "pred_binary_operator"
       [(match operand:WI 1 "register_operand" "r,r,r,r")
        (match operand:WI 2 "kvx_r_s10_s37_s64 operand" "r,I10,B37,i"))])
    (use (match operand:WI 6 "register_operand" "r,r,r,r"))]]))
 "kvx_ifcvt_ce_level >= KVX_IFCVT_CE2"
 "#"
 "kvx_ifcvt_ce_level >= KVX_IFCVT_CE3"
 [(set (match dup 6)
        (match_op_dup 3 [(match_dup 1) (match_dup 2)]))
  (cond_exec
   (match op_dup 5 [(match_dup 4) (const_int 0)]))
  (set (match_dup 9) (match_dup 6)))]
)

(define_insn_and_split "*wrapped_binary_<WI:mode>"
 [(set (match operand:WI 0 "register_operand" "r,r,r,r")
      (match operator:WI 3 "pred_binary_operator"
        [(match operand:WI 1 "register_operand" "r,r,r,r")
         (match operand:WI 2 "kvx_r_s10_s37_s64 operand" "r,I10,B37,i"))])
   (use (match operand:WI 4 "register_operand" "r,r,r,r"))
   "kvx_ifcvt_ce_level >= KVX_IFCVT_CE2"
   "#"
   "kvx_ifcvt_ce_level >= KVX_IFCVT_CE3"
   [(set (match dup 0)
      (match_op_dup 3 [(match_dup 1) (match_dup 2)]))])
)
```
CLEANUPS OF NON IF-CONVERTED REGIONS

Undo the PARALLEL wrapping done by (kvx_ifcvt_ce2_cond_arith_ce3) with the (define_insn_and_split) patterns previously shown

Also deactivate the previously inserted UNSPEC_DEFs by splitting them into USE

As CE3 is not always run, set the kvx_ifcvt_ce_level to enable splitting of the UNSPEC_DEFs and the unwrapping the pseudo-predicated instructions

This is done in the machine reorg pass, which requires that all splitting be done before doloop finalization and sched2
AGENDA

1. Kalray MPPA Processor and KVX Core
2. KVX Code Generation Features
3. GCC IF-Conversion Framework
4. Extending GCC IF-Conversion
5. First Results and Outlook
EXAMPLE OF KVX IF-CONVERSION (BEFORE)

```c
long defaultfValue;

long calculateSomething(long *a, long *b, long m_p, long m_q)
{
  long result;
  if (m_p > m_q + 1) {
    result = a[*b];
  } else {
    result = defaultfValue;
    a[*b+1] = result;
  }
  return result;
}
```

```assembly
.align 8
.global calculateSomething
.type calculateSomething, @function
addi $r3, $r3, 1  # 10  [c=4 l=4] adddi3/1
ld $r4 = 0[$r1]  # 9   [c=10 l=4] *movdi_all/7
copyd $r1 = $r0   # 2   [c=4 l=4] *movdi_all/8
  ;;  # (end cycle 8)
comps.ge $r3 = $r3, $r2  # 11  [c=4 l=4] cstoredi4/0
  ;;  # (end cycle 1)
cb.dnez $r3 # .L2  # 12  [c=8 l=4] *cbdi
  ;;  # (end cycle 3)
ld.xs $r0 = $r4[$r8]  # 14  [c=10 l=4] *movdi_all/7
  ret  # 44  [c=8 l=4] ret
  ;;  # (end cycle 9)
make $r0 = defaultfValue  # 19  [c=4 l=12] *movdi_all/16
addi $r4, $r4, 1  # 21  [c=4 l=4] adddi3/1
  ;;  # (end cycle 8)
ld $r0 = 0[$r8]  # 20  [c=10 l=4] *movdi_all/7
  ;;  # (end cycle 1)
std.xs $r4[$r1] = $r0  # 22  [c=8 l=4] *movdi_all/4
  ret  # 51  [c=8 l=4] ret
  ;;  # (end cycle 3)
.size calculateSomething, .-calculateSomething
```

CONV_MEM (# 14)
COND_MOVE (# 19)
COND_MEM (# 20)
SPEC_ARITH (# 21)
COND_MEM (# 22)

.L2:
EXAMPLE OF KVX IF-CONVERSION (AFTER)

```c
long defaultValue;

long calculateSomething(long *a, long *b, long m_p, lon calculatesomething:
{
    long result;
    if (m_p > m_q + 1) {
        result = a[*b];
    } else {
        result = defaultValue;
        a[*b+1] = result;
    }
    return result;
}

.ALIGN 8
.GLOBAL calculateSomething
.TYPE calculateSomething, @function

ld $r7 = 0[$r1]          # 9  [c=10 l=4]  *movdi_all/7
add   $r3 = $r3, 1      # 10 [c=4 l=4]  adddi3/l
cpypd $r1 = $r0          # 2  [c=4 l=4]  *movdi_all/0
;;  # (end cycle 0)
compd.ge $r3 = $t3, $r2 # 11 [c=4 l=4]  cstoredi4/0
;;  # (end cycle 1)
addx8d $r5 = $r7, $r8   # 55 [c=4 l=4]  *addx8d_m/0
cmved.dnez $r3? $r0 = defaultValue # 19  [c=4 l=12]  *cond_exec_move
add    $r6 = $r7, 1     # 21 [c=4 l=4]  adddi3/l
;;  # (end cycle 3)
ld.dnez $r3? $r0 = [$r0] # 20 [c=10 l=4]  *cond_exec_loadi0/0
addx8d $r4 = $r6, $r1   # 57 [c=4 l=4]  *addx8d_m/0
;;  # (end cycle 4)
ld.deqz $r3? $r0 = [$r5] # 56 [c=10 l=4]  *cond_exec_loadi0/0
;;  # (end cycle 5)
sd.dnez $r3? [r4] = $r0 # 58 [c=8 l=4]  *cond_exec_storedi0/0
ret   # 49  [c=8 l=4]  ret
;;  # (end cycle 7)
.size    calculateSomething, .-calculateSomething
```

**CONV CLS (# 14)**
**COND_MOVE (# 19)**
**COND_MEM (# 20)**
**SPEC_ARITH (# 21)**
**COND_MEM (# 22)**
void
t vector_cond1(int n, float a[restrict])
{
    for (int i = 0; i < n; i++) {
        if (a[i] < 0.0f) a[i] = -a[i];
        else a[i] += 1.0;
    }
}

.COND_ARTITH (# 26)

.L11:
    lwz.xs $r2 = $r0[$r1]
    faddw $r4 = $r2, $r8
    fnegw $r5 = $r2
    fcompw.olt $r6 = $r2, $r7
    cmoved.eqz $r6? $r5 = $r4
    sw.xs $r8[$r1] = $r5
    add $r0 = $r0, 1
    # loopdo end

.L6:
    ret
    # (end cycle 0)
}
void vector_cond2 (int n, float a[restrict], float b[restrict])
{
    for (int i = 0; i < n; i++) {
        if (a[i] < 0.0f) a[i] = -a[i];
        else a[i] = b[i];
    }
}
MORE EXAMPLES OF KVX IF-CONVERSION (3)

```c
void vector_cond3(int n, float a[restrict], float b[restrict])
{
    for (int i = 0; i < n; i++) {
        if (a[i] < 0.0f) a[i] = -a[i];
        else a[i] *= b[i];
    }
}
```

```
.align 8
.global vector_cond3
.type vector_cond3, @function
vector_cond3:
    cb.wle $r0, $a0
    ; ; ; # (end cycle 0)
    zimd $s5 = $r0
    make $s9 = 0x00000000
    make $s0 = 0
    ; ; ; # (end cycle 1)
    loopdo $r5, $a0
    ; ; ; # (end cycle 2)

.L27:
    lwz.xs $r4 = $a0[$s1]
    addx4d $r7 = $r0, $r2
    ; ; ; # (end cycle 0)
    fneqv $r3 = $r4
    fcmpeq.ltu $s8 = $r4, $s9
    ; ; ; # (end cycle 3)
    lwz.weqz $s8? $r3 = [$s7]
    ; ; ; # (end cycle 4)
    fmulw $s6 = $r4, $s3
    ; ; ; # (end cycle 7)
    cmovle.weqz $s8? $r3 = $s6
    ; ; ; # (end cycle 11)
    sw.xs $s8[$s1] = $s3
    addi $s0 = $s8, $1
    ; ; ; # (end cycle 12)
    # loopdo end

.L22:
    ret
    ; ; ; # (end cycle 0)
.size vector cond3, .-vector cond3
```

COND_MEM (# 26)
COND_ARITH (# 27)
### SUMMARY AND OUTLOOK

Implemented scalar if-conversion in GCC for the partially predicated KVX architecture

Relies on the IFCVT framework, but activate it before (CE1, CE2) and after (CE3) reload

### EXISTING SCALAR IF-CONVERSION APPROACHES

- On the SSA form: requires extensions such as Psi-SSA
- Before register allocation
  - CMOVE only: GEM compiler
  - Fully predicated: R-K and J. Fang algorithms (IA64)
  - Partially predicated: S. Freudenberger (TRACE)
- After register allocation: GCC ports IA64, C6x, FRV

### KEY FEATURES OF THE KVX SCALAR IF-CONVERSION IN GCC

- Apply pseudo-predication and local speculation after register allocation
- The scratch registers that will be unconditionally defined are reserved before register allocation with UNSPEC_DEFs and USEs
- The GCC FRV port looks for unused hard registers after reload instead

### NEXT STEPS

- Cannot reuse the existing GCC (define_cond_exec) machinery, as it may only generate (define_insn) patterns
- Automate generation of the (define_insn_and_split) patterns that enable pseudo-predication
- Performance tuning for scalar (while) loop pipelining