DEVELOPING AN LLVM BACKEND FOR THE KV3 KALRAY VLIW CORE

Focus on Scheduling

EuroLLVM 2022
Cyril SIX,
Compiler Engineer, Kalray

www.kalrayinc.com
KALRAY IN A NUTSHELL
Intelligent Data Processing, from Cloud to Edge

Kalray offers a new type of **processor** (DPU\(^1\)) and solutions targeting the booming markets of **edge computing** and **intelligent data processing**

---

**LEADER IN MANYCORE TECHNOLOGY**

- **3rd Generation** of MPPA® processor
- + €100m R&D investment
- 30 Patent families

---

**A GLOBAL PRESENCE**

(\(^1\) DPU : Data Processor Unit)

---

**INDUSTRIAL INVESTORS**

- Public Company (ALKAL)
- Support from European Govts
- Working with 500 fortune companies

---

EuroLLVM 2022 © Kalray SA. - All Rights Reserved.
MPPA® COOLIDGE ARCHITECTURE

MANYCORE PROCESSOR
Architecture updates
• 80 CPU cores
• 600 to 1200 MHz frequency modes
• Network on Chip (NoC)

COMPUTE CLUSTER
Architecture updates
• 16 cores
• Safety/Security 64-bit core
• DMA for asynchronous read/writes

3RD GENERATION VLIW CORE (KV3)
Architecture updates
• 64-bit core
• 6-issue VLIW architecture
• 16-bit/32-bit/64-bit IEEE 754-2008 FPU
• Vision/CNN Co-processor (TCA)
AGENDA

1. KV3 VLIW core
2. Scheduling the KV3 VLIW core in LLVM
3. Performance Comparison vs GCC
KV3 VLIW CORE

- 6-issue VLIW with interlocked pipeline (Very Large Instruction Word)
  - 2 ALUs, 1 MAU, 1 LSU, 1 BCU, 1 TCA

- User register bank: 64x64-bit
  - Up to 128-bits operands

- Coprocessor with its register bank: 48x256-bit
  - Specialized matrix multiplication instructions
  - 512-bit and 1024-bit operands
  - Data to/from coprocessor must be moved explicitly with instructions (reg-reg moves or reg-mem moves)
  - Coprocessor can be turned off to save energy

ALU = Arithmetic and Logic Unit
BCU = Branch and Control Unit
MAU = Multiply-Accumulate Unit
LSU = Load/Store Unit
FPU = Floating-Point Unit
## KV3 PIPELINE EXAMPLE

<table>
<thead>
<tr>
<th>Cycle</th>
<th>ID (Instruction Decode)</th>
<th>RR (Read Registers)</th>
<th>E1 (Execute (1))</th>
<th>E2 (Execute (2))</th>
<th>E3 (Execute (3))</th>
</tr>
</thead>
</table>
| 0     | LD $r0 = 50[$r4]
ADDD $r1 = $r2, $r3 |                     |                 |                 |                 |
| 1     | SD 50[$r4] = $r1
ADDD $r5 = $r6, $r7 | LD $r0 = 50[$r4]
ADDD $r1 = $r2, $r3 |                 |                 |                 |
| 2     | ADDD $r8 = $r0, $r2
ADDD $r9 = $r5
LD $r10 = 90[$r4] | SD 50[$r4] = $r1
ADDD $r5 = $r6, $r7 | LD $r0 = 50[$r4]
ADDD $r1 = $r2, $r3 |                 |                 |
| 3     | MULW $r1 = $r1, $r9   | ADDD $r8 = $r0, $r2
ADDD $r9 = $r5
LD $r10 = 90[$r4] | SD 50[$r4] = $r1
ADDD $r5 = $r6, $r7 | LD $r0 = 50[$r4]
ADDD $r1 = $r2, $r3 |                 |
| 4     | MULW $r1 = $r1, $r9   | ADDD $r8 = $r0, $r2
ADDD $r9 = $r5
LD $r10 = 90[$r4] | STALL            | SD 50[$r4] = $r1
ADDD $r5 = $r6, $r7 | LD $r0 = 50[$r4]
ADDD $r1 = $r2, $r3 |                 |

The whole bundle is stalled, not just the stalling instruction
EXAMPLE OF VLIW ASSEMBLY CODE

C code:

```c
struct {
    int16x16_t a;
    int16x16_t b;
    int16x16_t tr;
    int16x16_t tr2;
} test_vec[VECTOR_SIZE];
for (size_t i = 0; i < VECTOR_SIZE; i++) {
    test_vec[i].r = test_vec[i].a + test_vec[i].b;
}
```

KV3 vectorized VLIW code:

```mla
loopdo $r20, __LOOPDO_0_END_
.LBB0_4:
lo $r0r1r2r3 = 0[$r21]
   ;; // [... ellipsed code ...]
lo $r40r41r42r43 = 288[$r21]
addhq $r1 = $r5, $r1
addhq $r0 = $r4, $r0
addhq $r2 = $r6, $r2
   ;;
lo $r44r45r46r47 = 384[$r21]
addhq $r3 = $r7, $r3
   ;;
lo $r48r49r50r51 = 416[$r21]
addhq $r4 = $r32, $r8
addhq $r6 = $r34, $r10
   ;; // [... ellipsed code ...]
addhq $r36 = $r56, $r52
addhq $r38 = $r58, $r54
addhq $r39 = $r59, $r55
so 320[$r21] = $r8r9r10r11
   ;;
so 448[$r21] = $r32r33r34r35
   ;;
so 576[$r21] = $r36r37r38r39
add $r21 = $r21, 640
   ;;
__LOOPDO_0_END__: <-- hardware loop end
```

- Ideally, good VLIW code should:
  - Have hardware loops
  - Have big bundles
  - Little to no stall
  - Profit of vectorized instructions
OFFICIALLY SUPPORTED COMPILERS ON KV3

- The most used compiler in our applications
- Can compile Linux kernel for our architecture

Main backend developers:
- Benoît Dupont de Dinechin
- Paul Iannetta

Previously:
- Marc Poulhiès

- Newest addition
- Has OpenCL support

Main backend developers:
- Diogo Sampaio
- Cyril Six

Previously:
- Laurent Thévenoux
AGENDA

1. KV3 VLIW core

2. Scheduling the KV3 VLIW core in LLVM

3. Performance Comparison vs GCC
HOW TO SCHEDULE OUR VLIW CORE?

- **Bundles** group instructions to be scheduled at the same time.

- Two components for deciding the schedule times: **latencies** and **reservation tables (RT)**.

- Both are given by the architecture manual

Some examples of RTs:

<table>
<thead>
<tr>
<th>Resource</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISSUE</td>
<td>8</td>
</tr>
<tr>
<td>TINY (ALU)</td>
<td>4</td>
</tr>
<tr>
<td>MAU</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resource</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISSUE</td>
<td>1</td>
</tr>
<tr>
<td>TINY (ALU)</td>
<td>1</td>
</tr>
<tr>
<td>MAU</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>LO $r56r57r58r59 = 544[$r21]</td>
<td>0</td>
</tr>
<tr>
<td>ADDHQ $r10 = $r42, $r38</td>
<td>0</td>
</tr>
<tr>
<td>ADDHQ $r11 = $r43, $r39</td>
<td>1</td>
</tr>
<tr>
<td>ADDHQ $r9 = $r41, $r37</td>
<td>0</td>
</tr>
<tr>
<td>SO 64[$r21] = $r0r1r2r3</td>
<td>1</td>
</tr>
<tr>
<td>ADDHQ $r33 = $r49, $r45</td>
<td>1</td>
</tr>
<tr>
<td>ADDHQ $r43 = $r48, $r44</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code before bundling:</th>
</tr>
</thead>
<tbody>
<tr>
<td>lo $r56$r57$r58$r59 = 544[$r21]</td>
</tr>
<tr>
<td>addhq $r10 = $r42, $r38</td>
</tr>
<tr>
<td>addhq $r9 = $r41, $r37</td>
</tr>
<tr>
<td>si 64[$r21] = $r0r1r2r3</td>
</tr>
<tr>
<td>addhq $r33 = $r49, $r45</td>
</tr>
<tr>
<td>addhq $r43 = $r48, $r44</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code after bundling:</th>
</tr>
</thead>
<tbody>
<tr>
<td>lo $r56$r57$r58$r59 = 544[$r21]</td>
</tr>
<tr>
<td>addhq $r10 = $r42, $r38</td>
</tr>
<tr>
<td>addhq $r9 = $r41, $r37</td>
</tr>
<tr>
<td>si 64[$r21] = $r0r1r2r3</td>
</tr>
<tr>
<td>addhq $r33 = $r49, $r45</td>
</tr>
<tr>
<td>addhq $r43 = $r48, $r44</td>
</tr>
<tr>
<td>; } cycle 0</td>
</tr>
<tr>
<td>; } cycle 1</td>
</tr>
<tr>
<td>/* … */</td>
</tr>
</tbody>
</table>

Total resources*

<table>
<thead>
<tr>
<th>Resource</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISSUE</td>
<td>8</td>
</tr>
<tr>
<td>TINY (ALU)</td>
<td>4</td>
</tr>
<tr>
<td>MAU</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RT of MUL(reg, reg)</th>
<th>Resource</th>
<th>Cycle 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISSUE</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>TINY (ALU)</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MAU</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RT of MUL(reg, imm64)</th>
<th>Resource</th>
<th>Cycle 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISSUE</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>TINY (ALU)</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MAU</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

* not complete, only shows a subset of our units
SCHEDULING MODEL

Like Hexagon, we use itineraries

The resources are all allocated at Cycle 1

The operand reads occur at 2nd stage of pipeline

The operand writes happen at 3rd to 6th stage
SCHEDULING OPTIMIZATIONS USED IN OUR BACKEND

- intrablock DAG linearization
- Source scheduler (for now)
- can use either itineraries or WriteRes
- DFA generated with itineraries
- greedy

• All schedulers are at the level of basic blocks
• Each uses TargetSchedule.td

Pre-RA

ScheduleDAG → MachineScheduler → RegAllocGreedy

Post-RA

KVXVLIWPacketizer → PostRAScheduler

DFAPacketizer

- needs itineraries for hazard recognizer
- uses either itineraries or WriteRes for latencies
SHORTCOMINGS OF THE PACKETIZER

- LLVM's Packetizer is greedy: as long as it fits, it gets added
- This may break the schedule by:
  1. Inserts instructions in a bundle making it stall.
  2. Not conforming to the initial schedule planned by the scheduler

### Instruction Cycle

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>R56=R57R58R59 = LOAD(544[R21])</td>
<td>0</td>
</tr>
<tr>
<td>R9 = ADDHQ(R41, R37)</td>
<td>0</td>
</tr>
<tr>
<td>R8 = ADDHQ(R40, R46)</td>
<td>0</td>
</tr>
<tr>
<td>R10 = ADDHQ(R42, R38)</td>
<td>0</td>
</tr>
<tr>
<td>R11 = ADDHQ(R43, R39)</td>
<td>1</td>
</tr>
<tr>
<td>STORE(64[R21], R0R1R2R3)</td>
<td>1</td>
</tr>
<tr>
<td>R33 = ADDHQ(R49, R45)</td>
<td>1</td>
</tr>
<tr>
<td>R32 = ADDHQ(R48, R44)</td>
<td>1</td>
</tr>
<tr>
<td>R34 = ADDHQ(R50, R46)</td>
<td>2</td>
</tr>
<tr>
<td>R35 = ADDHQ(R51, R47)</td>
<td>2</td>
</tr>
<tr>
<td>STORE(192[R21], R4R5R6R7)</td>
<td>2</td>
</tr>
<tr>
<td>R37 = ADDHQ(R57, R53)</td>
<td>3</td>
</tr>
<tr>
<td>...</td>
<td>3+</td>
</tr>
</tbody>
</table>

### Instruction Issue Cycle

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Issue Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>R56=R57R58R59 = LOAD(544[R21])</td>
<td></td>
</tr>
<tr>
<td>R9 = ADDHQ(R41, R37)</td>
<td>0</td>
</tr>
<tr>
<td>R8 = ADDHQ(R40, R46)</td>
<td></td>
</tr>
<tr>
<td>R10 = ADDHQ(R42, R38)</td>
<td>0</td>
</tr>
<tr>
<td>R11 = ADDHQ(R43, R39)</td>
<td></td>
</tr>
<tr>
<td>STORE(64[R21], R0R1R2R3)</td>
<td></td>
</tr>
<tr>
<td>R33 = ADDHQ(R49, R45)</td>
<td>0</td>
</tr>
<tr>
<td>R32 = ADDHQ(R48, R44)</td>
<td></td>
</tr>
<tr>
<td>R34 = ADDHQ(R50, R46)</td>
<td>1</td>
</tr>
<tr>
<td>R35 = ADDHQ(R51, R47)</td>
<td></td>
</tr>
<tr>
<td>STORE(192[R21], R4R5R6R7)</td>
<td></td>
</tr>
<tr>
<td>R37 = ADDHQ(R57, R53)</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>4+</td>
</tr>
</tbody>
</table>
ONGOING WORK - A MORE ACCURATE PACKETIZER

• Merging Packetizer and post-RA scheduler in one pass

• (PoC) First candidate pass: PostRATDLList
  • Decorate SchedulePostRATDLList::{enterRegion, exitRegion, ScheduleNodeTopDown}
    • enterRegion: initialize data structures
    • ScheduleNodeTopDown: store the cycle
    • exitRegion: emit bundles
  • Problem: hard to encode ISSUE resources (no NumMicroOps, can't add ISSUE to itineraries)

• (WIP) New candidate pass: MachineScheduler

```
enterRegion: initialize data structures
ScheduleNodeTopDown: store the cycle
exitRegion: emit bundles
```

Scheduled code with bundles

```
KXVScheduler
```

RegAllocGreedy

```
MachineScheduler
```
AGENDA

1. KV3 VLIW core
2. Scheduling the KV3 VLIW core in LLVM
3. Performance Comparison vs GCC
Disclaimer:

- LLVM's KV3 backend is less mature than GCC's
- Removed "static" from all kernels to prevent GCC aggressive inter-function constant propagation.

GEOMEAN(gcc_cycles/clang_cycles) = 0.938

Not that bad! Can be improved:

- KV3-backend-specific improvements
- General improvements

Higher is better for clang
**IMPROVE FLOATING-POINT ASSOCIATIVITY?**

Example: seidel-2d, -ffast-math

**Kernel code:**

```c
for (t = 0; t <= _PB_TSTEPS - 1; t++)
for (i = 1; i <= _PB_N - 2; i++)
for (j = 1; j <= _PB_N - 2; j++)
```

**LLVM code:**

```llvm
fadd $r33 = $r34, $r33
// cycle 0
fadd $r33 = $r33, $r16
// cycle 4
fadd $r32 = $r33, $r32
// cycle 8
fadd $r32 = $r32, $r9
ld $r9 = 0x1f50[$r11]
copyd $r33 = $r15
// cycle 12
/* … */
```

**GCC code:**

```assembly
ld $r0 = 16[$r4]
fadd $r1 = $r5, $r10
add $r4 = $r4, 16
zcwd $r3 = $r2
// cycle 0
ld.xs $r11 = $r2[$r7]
// cycle 1
ld.xs $r15 = $r2[$r7]
// cycle 2
fadd $r9 = $r0, $r9
ld $r10 = 8[$r4]
// cycle 3
fadd $r1 = $r1, $r9
// cycle 7
/* … */
```

- LLVM computes the additions from left to right
- GCC reorganizes it into \((A[i-1][j-1] + A[i-1][j]) + (A[i-1][j+1] + A[i][j-1]) + \ldots\)
- 36% performance difference because of stalls
OPPOSITE EXAMPLE: BETTER CODE ON LLVM

Example: syrk

Kernel code:

for (i = 0; i < _PB_NI; i++)
for (j = 0; j < _PB_NI; j++)
for (k = 0; k < _PB_NJ; k++)
C[i][j] += alpha * A[i][k] * A[j][k];

LLVM code:

ld $r40 = 16[$r38]
;;
ld $r41 = 16[$r39]
fmuld $r40 = $r40, $r2
;;
ffmad $r36 = $r40, $r41
;;
sd 0[$r35] = $r36
;;
/* … repeated … */

GCC code:

ld.xs $r49 = $r38[$r6]
;;
ld.xs $r50 = $r38[$r4]
fmuld $r46 = $r49, $r50
;;
ffmad $r3 = $r7, $r46
;;
sd.xs $r43[$r5] = $r3
;;
/* … repeated … */

• The order of operation is different:
  • GCC computes alpha * (A[i][k] * A[j][k])
    --> A[j][k] needs to be loaded before the first operation
  • LLVM computes (alpha * A[i][k]) * A[j][k]
    --> A[j][k] can be loaded later
• Same number of stalls, but one more bundle = 1 more cycle per iteration
**IMPROVE VECTORIZATION**

**Example: jacobi-1d-imper**

**Kernel code:**

```
for (i = 1; i < n - 1; i++)
```

**LLVM code:**

```
.LBB0_4:
    ld $r8 = 0[$r7]
    addd $r5 = $r5, -1
    ld $r9 = 8[$r7]
    fadd $r8 = $r9, $r8
    ld $r9 = 16[$r7]
    addd $r7 = $r7, 8
    fadd $r8 = $r8, $r9
    addd $r9 = $r6, 8
    fmul $r8 = $r8,
        0x3fd55475a31a4be
    sd 8[$r6] = $r8
    copyd $r6 = $r9
    cb.dnez $r5 ? .LBB0_4
```

**GCC code:**

```
loopdo $r5, .L34
    ld $r8 = 0[$r7]
    addd $r5 = $r5, -1
    ld $r9 = 8[$r7]
    fadd $r8 = $r9, $r8
    ld $r9 = 16[$r7]
    add $r7 = $r7, 8
    fadd $r8 = $r8, $r9
    add $r9 = $r6, 8
    fmul $r8 = $r8,
        0x3fd55475a31a4be
    sd 8[$r6] = $r8
    copyd $r6 = $r9
    cb.dnez $r5 ? .LBB0_4
```

- GCC was able to use the vectorized version of fadd and ld: fadddp and lq
- Probably some more tuning is needed in our backend
- Almost 2x performance difference
- (also: GCC used hardware loop)
C code:

int abs_ssub(int a, int b){
    long sub = (long)(a) - (long)(b);
    if (sub < -2147483648)
        sub = -2147483648;
    else if (sub > 2147483647)
        sub = 2147483647;
    if (sub < 0)
        return (int)(-sub);
    return (int)(sub);
}

GCC code:

abs_ssub:
    sxwd $r2 = $r0
    make $r0 = 0x0000000080000000
    ;;
    sbfwd $r1 = $r1, $r2
    addd $r2 = $r0, -1
    ;;
    zxwd $r3 = $r1
    negw $r6 = $r1
    compd.gt $r5 = $r1, 0x000000007fffffff
    compd.lt $r4 = $r1, 0xffffffff80000000
    ;;
    cmoved.dltz $r1? $r3 = $r6
    ;;
    cmoved.deqz $r5? $r2 = $r3
    ;;
    cmoved.deqz $r4? $r0 = $r2
    ret
    ;;

LLVM code:

abs_ssub:
    sbfsw $r0 = $r1, $r0
    ;;
    absw $r0 = $r0
    ret
    ;;

LLVM IR code:

define i32 @abs_ssub(i32 %0, i32 %1)
{
    %3 = tail call i32 @llvm.ssub.sat.i32(i32 %0, i32 %1)
    %4 = tail call i32 @llvm.abs.i32(i32 %3, i1 false)
    ret i32 %4
}
THANK YOU

GitHub: https://github.com/kalray/llvm-project
Email: csix@kalrayinc.com, dsampaio@kalrayinc.com
DISCLAIMER

Kalray makes no guarantee about the accuracy of the information contained in this document. It is intended for information purposes only and shall not be incorporated into any contract. It is not a commitment to deliver any material, code or functionality, and should not be relied upon in making purchasing decisions. The development, release and timing of any features or functionality described for Kalray products remains at the sole discretion of Kalray.

Trademarks and logos used in this document are the properties of their respective owners.
KV3 BACKEND INSTRUCTION SELECTION

• Supports two versions of kv3: kv3-1 and kv3-2 (newest version).
• Supports 570 different instructions in total. Most are common to kv3-1 and kv3-2.
• Example below: integer addition

<table>
<thead>
<tr>
<th>Instruction(inttype1, inttype2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDW(i32, i32)</td>
</tr>
<tr>
<td>ADDD(i64, i64)</td>
</tr>
<tr>
<td>ADDHQ(v4i16, v4i16)</td>
</tr>
<tr>
<td>ADDWD(i32, i64)</td>
</tr>
<tr>
<td>ADDWP(v2i32, v2i32)</td>
</tr>
</tbody>
</table>

• More complex examples: complex multiplication, multiply-add, saturated arithmetic, dot product..

• About four variants per instruction:
  • Register-Register format
  • Register-Signed10 format
  • Register-Signed37 format
  • Register-Signed64 format

Example of Register-Signed10

$r0 = addw \, r1, 42$

• In total, we have ~1700 "def" patterns, ~450 "defm" and 66 multiclasses.