KALRAY
THE POWER OF MORE

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**

KALRAY
THE POWER OF MORE

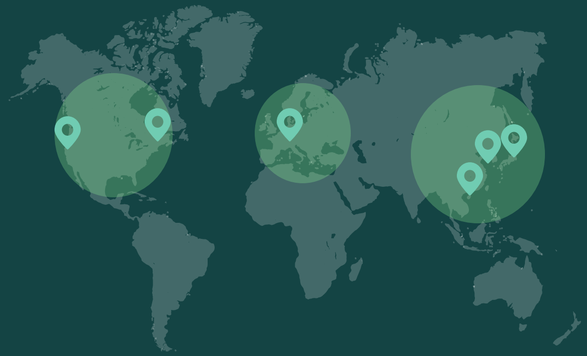
LEADER IN MANYCORE TECHNOLOGY

3rd
Generation of MPPA® processor

+ €100m
R&D investment

30
Patent families

A GLOBAL PRESENCE



INDUSTRIAL INVESTORS

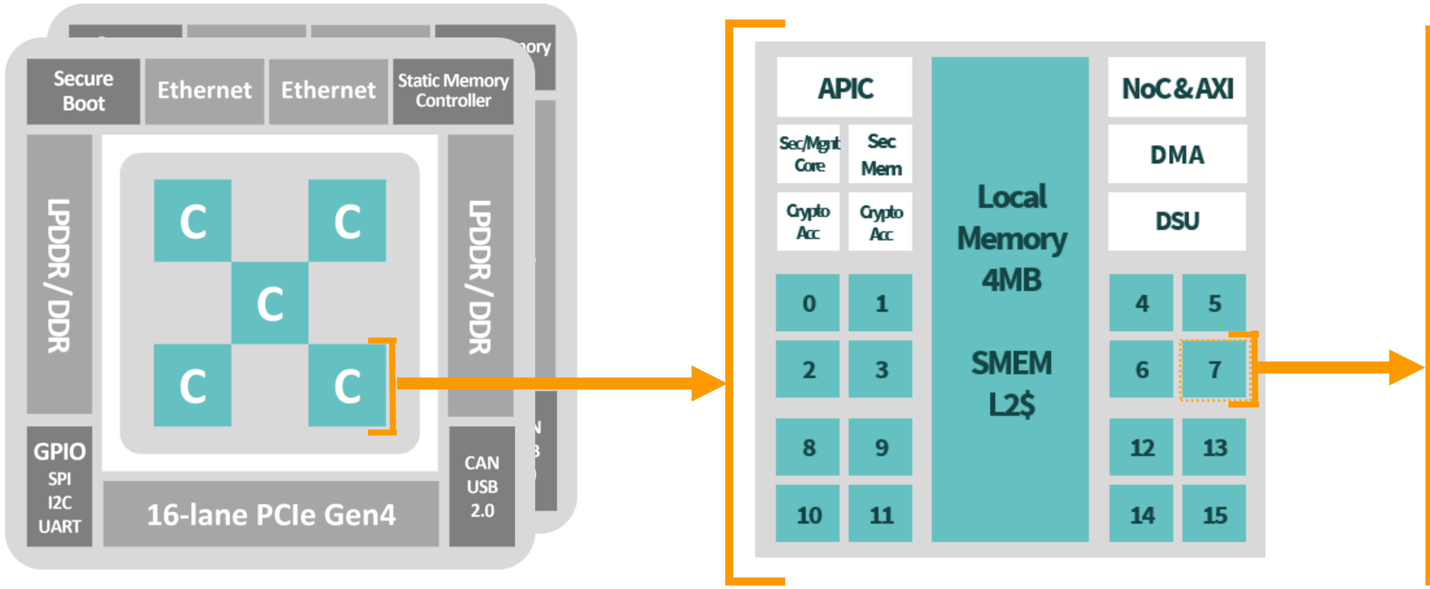
NXP
RENAULT NISSAN MITSUBISHI
SAFRAN
MBDA

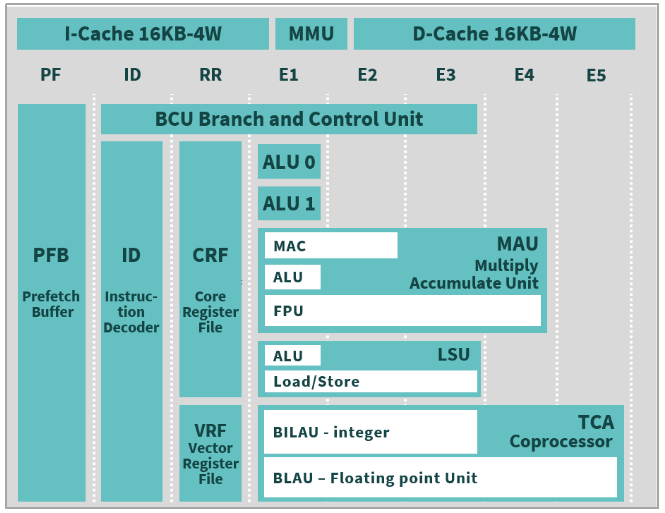
EURONEXT

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

(1) DPU : Data Processor Unit

MPPA® COOLIDGE ARCHITECTURE





MANYCORE PROCESSOR
Architecture updates

- 80 CPU cores
- 600 to 1200 MHz frequency modes
- Network on Chip (NoC)

COMPUTE CLUSTER
Architecture updates

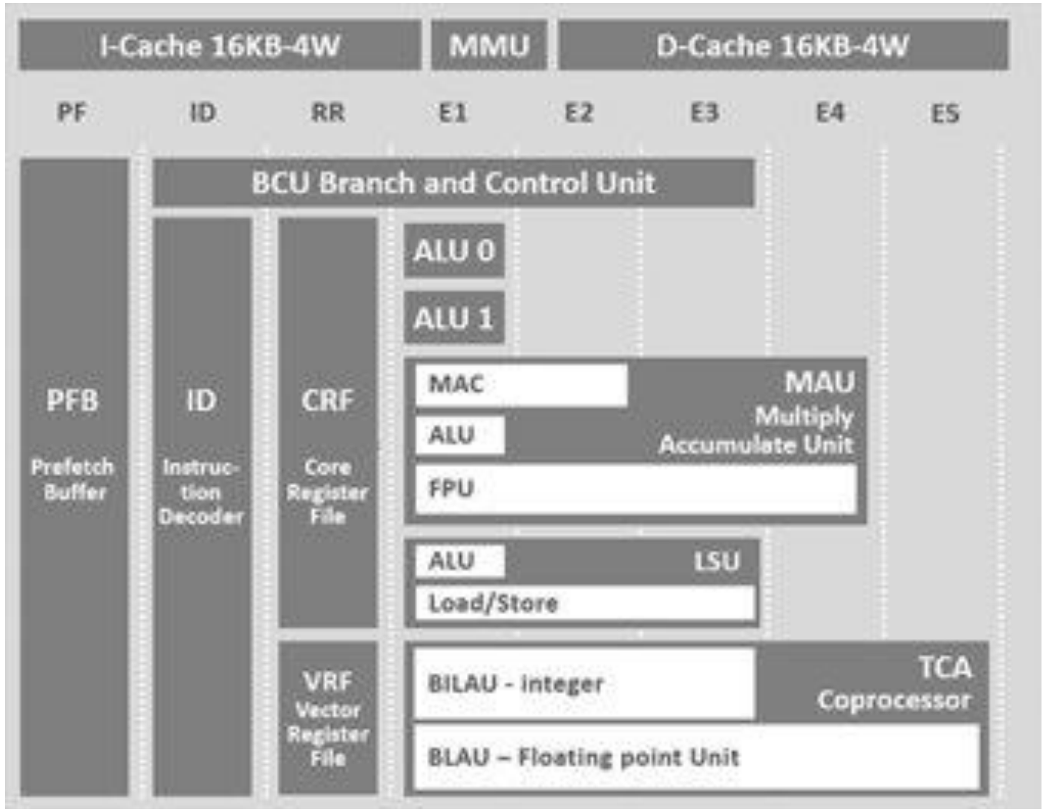
- 16 cores
- Safety/Security 64-bit core
- DMA for asynchronous read/writes

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

KV3 VLIW CORE



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

- 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

KV3 PIPELINE EXAMPLE

Cycle	ID (Instruction Decode)	RR (Read Registers)	E1 (Execute (1))	E2 (Execute (2))	E3 (Execute (3))
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:

```
struct {
  int16x16_t a;
  int16x16_t b;
  int16x16_t r;
  int16x16_t r2;
} 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;
}
```

A bundle -->

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

KV3 vectorized VLIW code:

```
    loopdo $r20, .__LOOPDO_0_END_          <-- hardware loop begin
    ;;
.LBB0_4:
    lo $r0r1r2r3 = 0[$r21]                 <-- loading values..
    ;; // [… ellipsed code …]
    lo $r40r41r42r43 = 288[$r21]
    addhq $r1 = $r5, $r1
    addhq $r0 = $r4, $r0
    addhq $r2 = $r6, $r2
    ;;
    lo $r44r45r46r47 = 384[$r21]           <-- summing current values
    addhq $r3 = $r7, $r3                    and fetching the next ones
    ;;
    lo $r48r49r50r51 = 416[$r21]
    addhq $r5 = $r33, $r9
    addhq $r4 = $r32, $r8
    addhq $r6 = $r34, $r10
    ;; // [… ellipsed code …]
    addhq $r36 = $r56, $r52
    addhq $r38 = $r58, $r54                <-- summing last values
    addhq $r39 = $r59, $r55                and saving the computations
    so 320[$r21] = $r8r9r10r11
    ;;
    so 448[$r21] = $r32r33r34r35
    ;;
    so 576[$r21] = $r36r37r38r39
    addd $r21 = $r21, 640
    ;;
.__LOOPDO_0_END_:                          <-- hardware loop end
```

OFFICIALLY SUPPORTED COMPILERS ON KV3

v9.4.0

- 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

v12.0.1

- 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

Code before bundling:

Instruction	Cycle
LO $r56r57r58r59 = 544[$r21]	0
ADDHQ $r10 = $r42, $r38	0
ADDHQ $r11 = $r43, $r39	1
ADDHQ $r9 = $r41, $r37	0
SO 64[$r21] = $r0r1r2r3	1
ADDHQ $r33 = $r49, $r45	1
ADDHQ $r43 = $r48, $r44	1
…	…

Code after bundling:

```
lo $r56r57r58r59 = 544[$r21]
addhq $r10 = $r42, $r38
addhq $r9 = $r41, $r37
;; // cycle 0
addhq $r11 = $r43, $r39
so 64[$r21] = $r0r1r2r3
addhq $r33 = $r49, $r45
addhq $r43 = $r48, $r44
;; // cycle 1
/* … */
```

Some examples of RTs:

Total resources*

Resource	Amount
ISSUE	8
TINY (ALU)	4
MAU	1

RT of MUL(reg, reg)

Resource	Cycle 0
ISSUE	1
TINY (ALU)	1
MAU	1

RT of MUL(reg, imm64)

Resource	Cycle 0
ISSUE	3
TINY (ALU)	1
MAU	1

* not complete, only shows a subset of our units

SCHEDULING MODEL

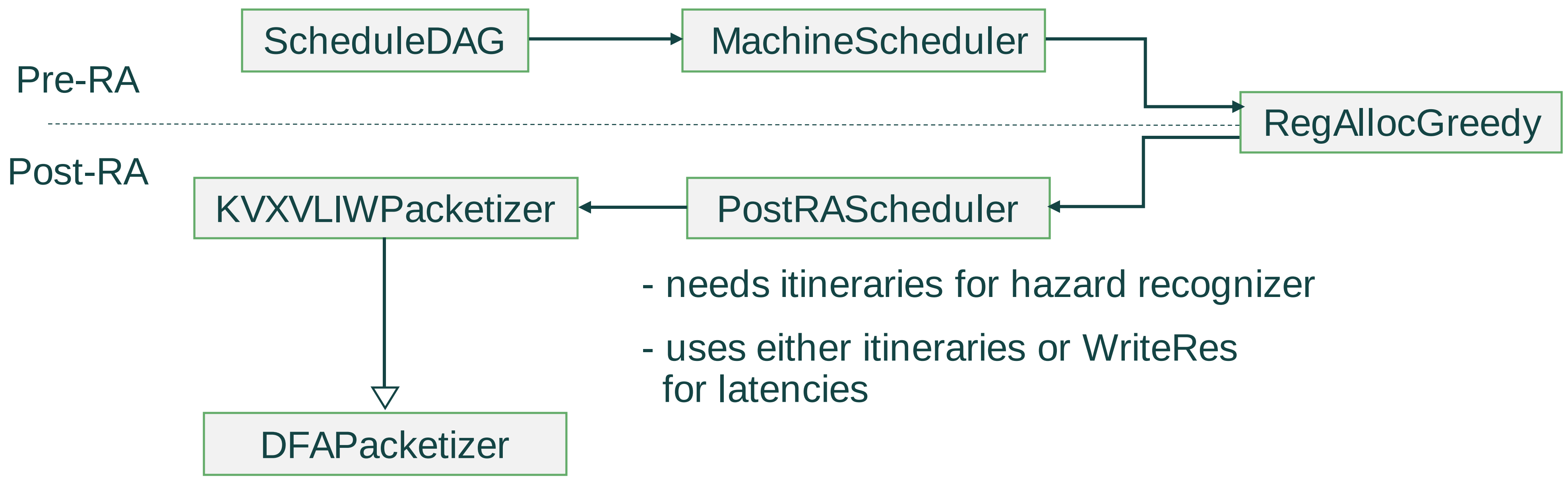
KVXSchedule.td

```
foreach I = 0-3 in def TINY#I#_FU : FuncUnit;
def MAU_FU : FuncUnit;
def LSU_FU : FuncUnit;

/* … */

InstrItinData<ALU_TINY, [
InstrStage<1, [TINY0_FU, TINY1_FU, TINY2_FU, TINY3_FU]>,
], [3, 2, 2]
>,
/* … */
InstrItinData<LSU_LOAD, [
InstrStage<1, [TINY0_FU, TINY1_FU, TINY2_FU, TINY3_FU], 0>,
InstrStage<1, [LSU_FU]>
], [5, 2, 2, 2, 2, 2]
>,
/* … */
InstrItinData<MAU, [
InstrStage<1, [TINY0_FU, TINY1_FU, TINY2_FU, TINY3_FU], 0>,
InstrStage<1, [MAU_FU]>
], [4, 2, 2]
>,
```

- 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

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



ScheduleDAG:
- intrablock DAG linearization
- Source scheduler (for now)

MachineScheduler:
- can use either itineraries or WriteRes

PostRAScheduler:
- needs itineraries for hazard recognizer
- uses either itineraries or WriteRes for latencies

DFAPacketizer:
- DFA generated with itineraries
- greedy

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
-------------------------------------
R56R57R58R59 = LOAD(544[R21])| 0
R9 = ADDHQ(R41, R37)          | 0
R8 = ADDHQ(R40, R46)          | 0
R10 = ADDHQ(R42, R38)         | 0
R11 = ADDHQ(R43, R39)         | 1
STORE(64[R21], R0R1R2R3)      | 1
R33 = ADDHQ(R49, R45)         | 1
R32 = ADDHQ(R48, R44)         | 1
R34 = ADDHQ(R50, R46)         | 2
R35 = ADDHQ(R51, R47)         | 2
STORE(192[R21], R4R5R6R7)     | 2
R37 = ADDHQ(R57, R53)         | 3
…                             | 3+
```

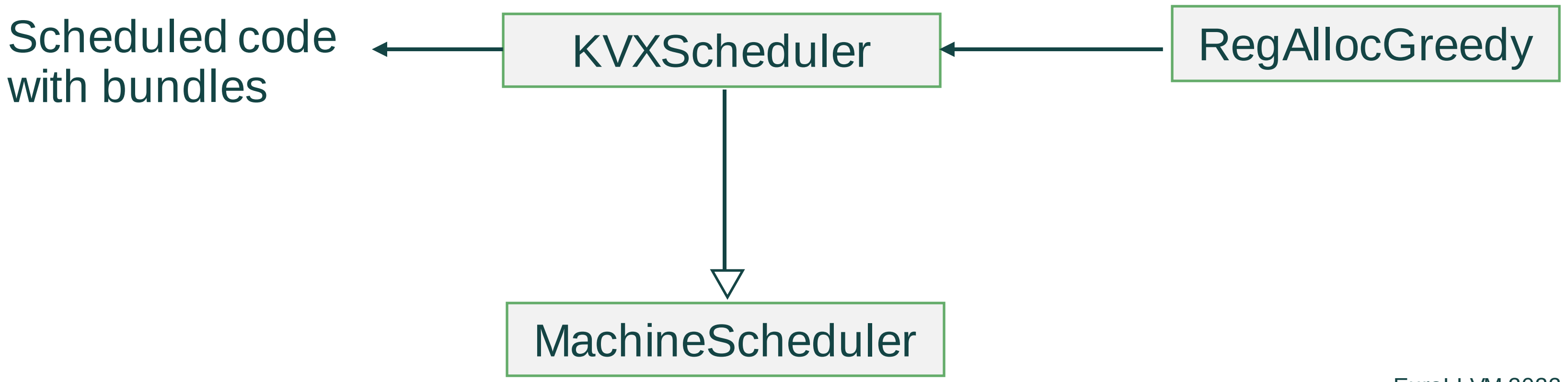
```
Instruction                    | Issue Cycle
--------------------------------------------
R56R57R58R59 = LOAD(544[R21])|
R9 = ADDHQ(R41, R37)          |
R8 = ADDHQ(R40, R46)          |
R10 = ADDHQ(R42, R38)         |
;;                            | 0
R11 = ADDHQ(R43, R39)         |
STORE(64[R21], R0R1R2R3)      |
R33 = ADDHQ(R49, R45)         |
R32 = ADDHQ(R48, R44)         |
;;                            | 1
R34 = ADDHQ(R50, R46)         |
R35 = ADDHQ(R51, R47)         |
STORE(192[R21], R4R5R6R7)     |
R37 = ADDHQ(R57, R53)         |
;;                            | 2
…                             | 4+
```

ONGOING WORK - A MORE ACCURATE PACKETIZER

- Merging Packetizer and post-RA scheduler in one pass

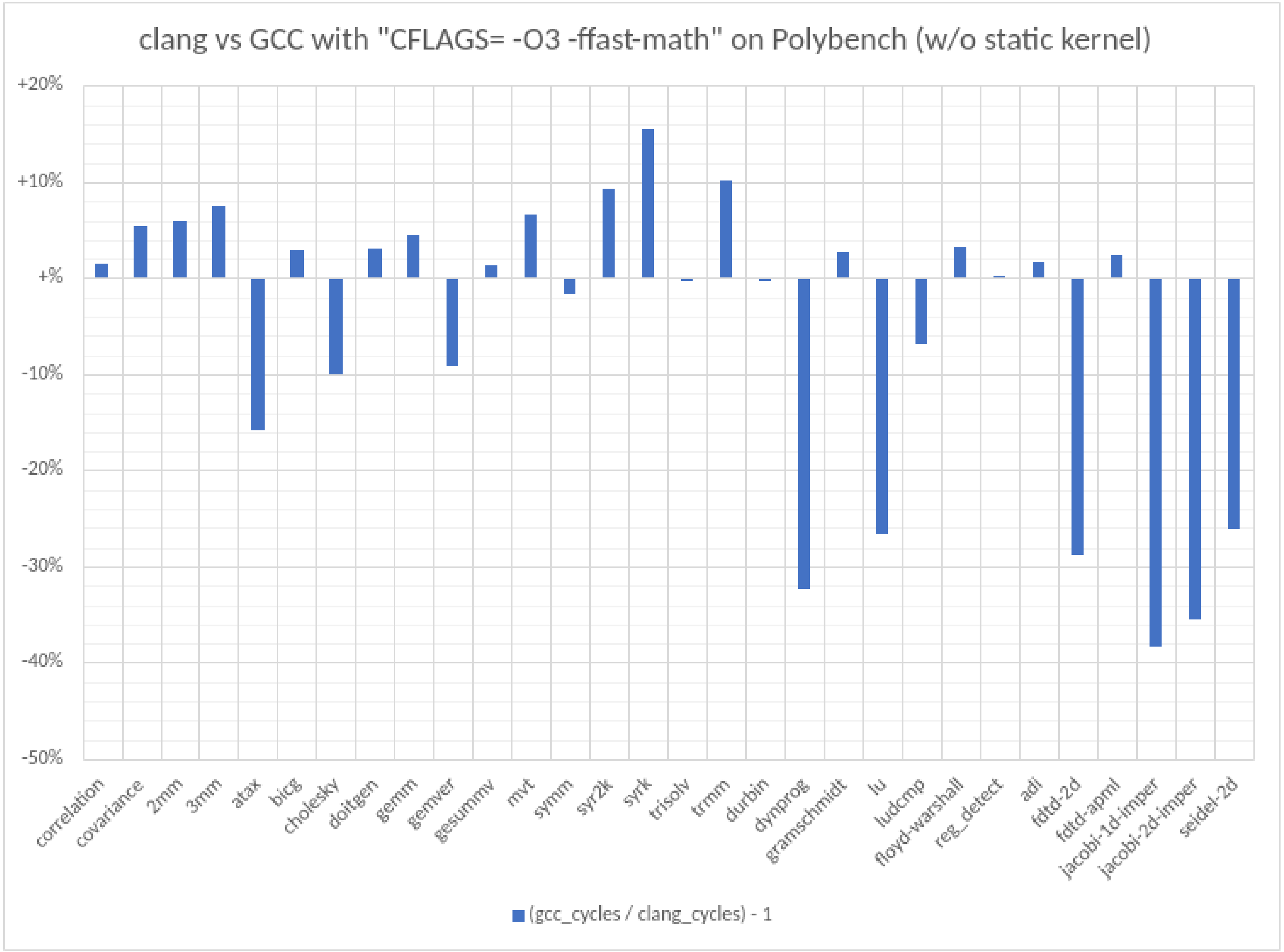
- (PoC) First candidate pass: PostRATDList
  - Decorate SchedulePostRATDList::{enterRegion, exitRegion, ScheduleNodeTopDown}
    - enterRegion: initialize data structures
    - ScheduleNodeTopDown: store the cycle
    - exitRegion: emit bundles
  - Problem: hard to encode ISSUE resources (no NumMicroOps,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

- (WIP) New candidate pass: MachineScheduler



AGENDA

LLVM VS GCC ON KV3 TARGET, POLYBENCH



Higher is better for clang

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

IMPROVE FLOATING-POINT ASSOCIATIVITY?

Example: seidel-2d, -ffast-math

Kernel code:

```
for (t = 0; t <= _PB_TSTEPS - 1; t++)
  for (i = 1; i<= _PB_N - 2; i++)
    for (j = 1; j <= _PB_N - 2; j++)
      A[i][j] = (A[i-1][j-1] + A[i-1][j] + A[i-1][j+1]
          + A[i][j-1] + A[i][j] + A[i][j+1]
          + A[i+1][j-1] + A[i+1][j] + A[i+1][j+1])/9.0;
```

LLVM code:

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

GCC code:

```
   ld $r0 = 16[$r4]
   faddd $r1 = $r5, $r10
   addd $r4 = $r4, 16
   zxwd $r3 = $r2
   ;; // cycle 0
   ld.xs $r11 = $r2[$r16]
   ;; // cycle 1
   ld.xs $r15 = $r2[$r7]
   ;; // cycle 2
   faddd $r9 = $r0, $r9
   ld $r10 = 8[$r4]
   ;; // cycle 3
   faddd $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]) + …
- 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++)
  B[i] = 0.33333 * (A[i-1] + A[i] + A[i + 1]);
```

LLVM code:

```
.LBB0_4:
        ld $r8 = 0[$r7]
        addd $r5 = $r5, -1
        ;;
        ld $r9 = 8[$r7]
        ;;
        faddd $r8 = $r9, $r8
        ld $r9 = 16[$r7]
        addd $r7 = $r7, 8
        ;;
        faddd $r8 = $r8, $r9
        addd $r9 = $r6, 8
        ;;
        fmuld $r8 = $r8,
         0x3fd555475a31a4be
        ;;
        sd 8[$r6] = $r8
        copyd $r6 = $r9
        cb.dnez $r5 ? .LBB0_4
        ;;
```

GCC code:

```
        loopdo $r5, .L34
        ;;
        lq.xs $r8r9 = $r4[$r33]
        ;;
        lq.xs $r0r1 = $r4[$r42]
        ;;
        lq.xs $r6r7 = $r4[$r38]
        ;;
        fadddp $r0r1 = $r8r9, $r0r1
        ;;
        fadddp $r0r1 = $r0r1, $r6r7
        ;;
        fmuld $r0 = $r0, $r16
        ;;
        fmuld $r1 = $r1, $r17
        ;;
        sq.xs $r4[$r35] = $r0r1
        addd $r4 = $r4, 1
        ;;
.L34:
```

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

LLVM INSTRUCTION-COMBINE ROCKS

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

KALRAY

THE POWER OF MORE

www.kalrayinc.com

DISCLAIMER

KALRAY

THE POWER OF MORE

www.kalrayinc.com

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

Instruction(inttype1, inttype2)
ADDW(i32, i32)
ADDD(i64, i64)
ADDHQ(v4i16, v4i16)
ADDWD(i32, i64)
ADDWP(v2i32, v2i32)

- 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.