J3: Compilers Challenges for Computing In Memory

Compilation for Strange applications and In Memory Processing
ACACES july 2022 summer school, HiPEAC, Fiuggi, Italy

Henri-Pierre CHARLES

CEA DSCIN department / Grenoble

july 13, 2022

Outline

Scientific Evolution : Compilation Research Domains

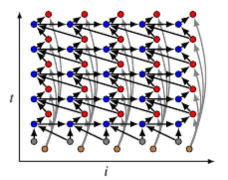
Compilation Topics Map

- **Find code structure** Extract parallelism : polyhedral approach
- **Assertion on legacy code** correctness proof, hard realtime, model checking
- **Security** HW attach counter mesure, obfuscation
- **Tools for scalability** Systems and library for big parallel machines
- **Reproductibility** Statistics tools and reproductible research
- **Ad hoc code optimization** Application driven code optimization
- **Legacy compiler optimization** follow the HW evolution
- **New code generation paradigm** JIT, Dynamic code generation

https://top500.org

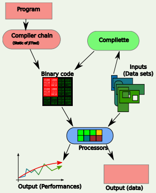
- **LAPACK** June 2002 #1 : HPE Cray 8,730,112 cores, Linpack perf 1,102.00 PFlop/s, 65% of peak performance (usualy better)
- **HPCG** Fugaku 7630848 cores, 16,004 TFlops 2.98 % of peak performance !

Polyhedral model



(a) CDAG for $T = 4, N = 7$

Algorithmic accelerator



HybroGen : Initial Objectives : New Code Generation Paradigm

Objectives: Application with Binary Automodification

- Without external libraries
- As fast as possible : Code generation speed $\tilde{1}0$ clock cycles per instruction)
- As small as possible : Code generators $\tilde{1}$ KB
- Portable accross architecture : RISCV, IBM Power 8 / 9, Kalray, CxRAM

Benefits

- Data set code generation dependant : values, size, strides
- User programmable code generation
- Heterogeneous ISA code generation

HybroGen : Code Generation Options for new architectures

New architecture = new ISA

- High level simulation + Performance model : using LLVM IR + interpretation
  - Pro : very fast, no platform dependency,
  - Cons : no real code generation, rough performance estimations
- Intrinsics
  - Pro : real code generation
  - Cons : need manual code instrumentation with low level ASM intrinsics, big benchmarking effort
- DSL (Domain Specific Language)
  - Pro : real code generation, automatic code generation, lower benchmarking effort
  - Cons : need initial programming effort
- Industrial compiler
  - Pro : real code generation, automatic code generation, no benchmarking effort, huge impact if sucessfull, need open source approach
  - Cons : need huge programming effort (LLVM = 1.6M loc, GCC 5 M loc))

HybroGen : Compilette Principle

Compilette

- Embed in an application a code generator able to regenerate part of the code
- Term invented in 2005 [Ref]Compilette

Challenges

- Do it fast (Cycle / instructions)
- Use a small memory footprint (KB)

Motivation : Static Compiler Versus Compilette

Static Compiler

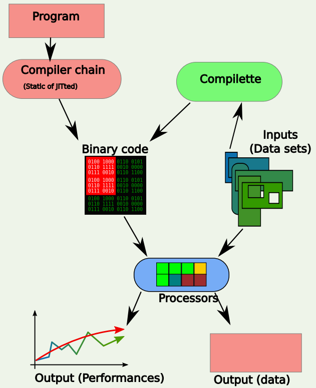
- Run once
- Does not know data set characteristics
- Slow compilation (even with JIT)

Compilette

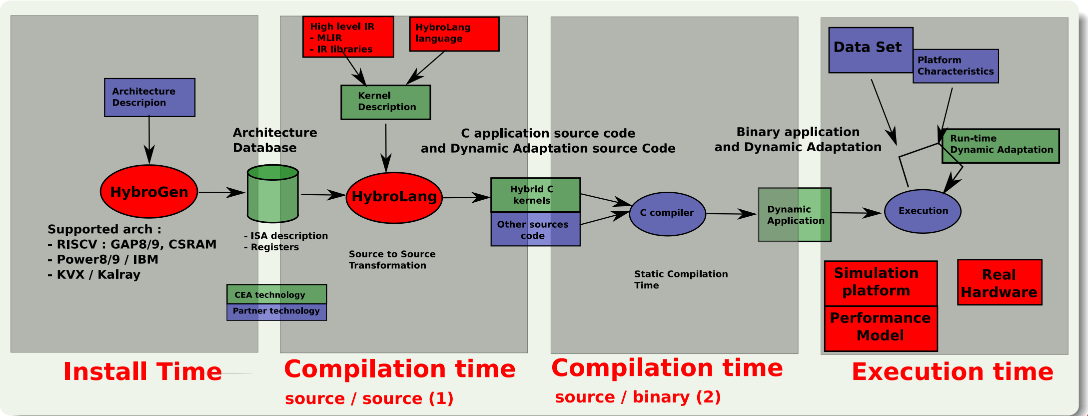
- Adapt on the fly
- Knowledge of the architecture
- Knowledge of the application

Need tools !

Static compilation versus dynamic adaptation



HybroGen: General View



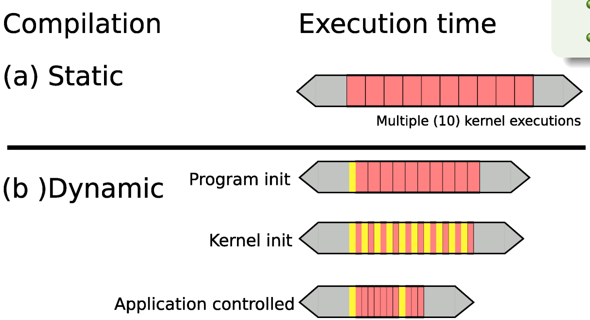
HybroGen : Objectives

Application domains

- Stochastic number support
- Packet filtering : Datatype ipv4, ipv6 addresses
- Transprecision algorithms : usage on mathematical iterative methods
- Stencil processing

Already done

- On the fly code generation for heterogeneous architectures
- Transprecision support : on the fly code generation for precision adaptation. Working demonstration on Newton algorithm : Power / RISCV / Kalray
- Support for In Memory Computing (next slides)
- Target processor modeling (QEMU plugin)



HybroGen : Simple-Add-Source

Simple Addition with specialization

```
typedef  int  (*pifi)(int);

h2_insn_t * genAdd(h2_insn_t * ptr, int  b)
{
  #[
  int 32 1 add (int 32 1 a)
  {
    int 32 1 r;
    r = #(b) + a; // b values will be included in code generation
    return r;
  }
  ]#
  return (h2_insn_t *) ptr;
}
```

HybroGen : Simple-Add-Source

Simple Addition with specialization

```
int main(int argc, char * argv[])
{
  h2_insn_t * ptr;
  int in0, in1, res;
  pifi fPtr;

  if (argc < 3)
    {
      printf("Give␣2␣values\n");
      exit(-1);
    }
  in0  = atoi (argv[1]);    // Get the users values in1 & in2
  in1  = atoi (argv[2]);
  ptr  = h2_malloc (1024);  // Allocate memory for 1024 instructions
  printf("//␣Compilette␣for␣simple␣addition␣between␣1␣variable␣with\n");
  printf("//␣code␣specialization␣on␣value␣=␣%d\n", in0);
  fPtr = (pifi) genAdd (ptr, in0); // Generate instructions
  res  = fPtr(in1);  // Call generated code
  printf("%d␣+␣%d␣=␣%d\n", in0, in1, res);
```

HybroGen : Simple how to Build

How to run example

```
gre061041:CodeExamples/>./RunDemo.py -a riscv -i Add-With-Specialization
Namespace(arch=['riscv'], clean=False, debug=False, inputfile=['Add-With-Specializat
-->rm -f Add-With-Specialization Add-With-Specialization.c
-->which riscv32-unknown-elf-gcc
-->../HybroLang.py --toC --arch riscv --inputfile Add-With-Specialization.hl
-->riscv32-unknown-elf-gcc -Wall -o Add-With-Specialization Add-With-Specialization.
('3', '25')
-->qemu-riscv32 Add-With-Specialization 3 25
gre061041:CodeExamples/>qemu-riscv32 Add-With-Specialization 3 25
// Compilette for simple addition between 1 variable with
// code specialization on value = 3
3 + 25 = 28
```

HybroGen : Simple-Add : Generated Source

Code Macro instructions

```
#define riscv_G32(INSN){ *(h2_asm_pc++) = (INSN);}

#define RV32I_RET__I_32_1() /* RET */ \
do { \
        riscv_G32(((0x8067 >> 0) & 0xffffffff)); \
} while(0)

#define RV32I_ADDI_RRI_I_32_1(r1,r0,i0) /* ADD */ \
do { \
        riscv_G32(((i0 & 0xfff) << 20)|((r0 & 0x1f) << 15)|((0x0 & 0x7) << 12)|((r1
} while(0)
```

HybroGen : Simple-Add-Generated

Instruction Selector

```
void riscv_genADD_3(h2_sValue_t P0, h2_sValue_t P1, h2_sValue_t P2)
{

    if ((P0.arith == 'i') && (P0.wLen <= 32) && (P0.vLen == 1) && P0.ValOrReg == REG
    {
        RV32I_ADDI_RRI_I_32_1(P0.regNro, P1.regNro, P2.valueImm);
    }

    else if ((P0.arith == 'i') && (P0.wLen <= 32) && (P0.vLen == 1) && P0.ValOrReg =
    {
        RV32I_ADD_RRR_I_32_1(P0.regNro, P1.regNro, P2.regNro);
    }
```

HybroGen : Simple-Add Generated code

Main code generator

```
h2_insn_t * genAdd(h2_insn_t * ptr, int  b)
{
/* Code Generation of 4 instructions */
/* Symbol table :*/
        /*VarName = { ValOrLen, arith, vectorLen, wordLen, regNo, Value} */
        h2_sValue_t a = {REGISTER, 'i', 1, 32, 10, 0};
        h2_sValue_t h2_outputVarName = {REGISTER, 'i', 1, 32, 10, 0};
        h2_sValue_t r = {REGISTER, 'i', 1, 32, 5, 0};
        h2_sValue_t h2_00000003 = {REGISTER, 'i', 1, 32, 6, 0};


/* Label  table :*/
#define riscv_genLABEL(LABEL_ID) labelAddresses[LABEL_ID] = h2_asm_pc;
h2_insn_t   * labelAddresses []={
        };


        h2_asm_pc = (h2_insn_t *) ptr;
```

HybroGen : Simple-Add Generated Code

Compilette

```
h2_asm_pc = (h2_insn_t *) ptr;
h2_codeGenerationOK = true;
riscv_genMV_2(h2_00000003, (h2_sValue_t) {VALUE, 'i', 1, 32, 0, (b)});
riscv_genADD_3(r, h2_00000003, a);
riscv_genMV_2(h2_outputVarName, r);
riscv_genRET_0();
/* Call back code for loops */
h2_save_asm_pc = h2_asm_pc;
h2_asm_pc = h2_save_asm_pc;
h2_iflush(ptr, h2_asm_pc);
```

HybroGen : Simple-Add-Exec

Simple run

```
gre061041:CodeExamples/>qemu-riscv32 Add-With-Specialization 3 40
// Compilette for simple addition between 1 variable with
// code specialization on value = 3
3 + 40 = 43
```

Run with debug

```
qemu-riscv32 Add-With-Specialization 3 25
// Compilette for simple addition between 1 variable with
// code specialization on value = 3
0x19008 : RV32I_MV_RI_I_32_1
0x1900c : RV32I_ADD_RRR_I_32_1
0x19010 : RV32I_MV_RR_I_32_1
0x19014 : RV32I_RET__I_32_1
3 + 25 = 28
```

HybroGen : Simple-Add Debug

1 shell to Run / interact

```
gre061041:CodeExamples/>qemu-riscv32 -g \
   7777 Add-With-Specialization 3 25
// Compilette for simple addition
// between 1 variable with
// code specialization on value = 3
0x19008 : RV32I_MV_RI_I_32_1
0x1900c : RV32I_ADD_RRR_I_32_1
0x19010 : RV32I_MV_RR_I_32_1
0x19014 : RV32I_RET__I_32_1
```

1 shell to Debug / observe

```
riscv32-unknown-elf-gdb Add-With-Specialization
GNU gdb (GDB) 9.2
../..
(gdb) target remote :7777
(gdb) break main
Breakpoint 1 at 0x107c6: file Add-With-Specialization.c, line 201.
(gdb) c
Continuing.
Breakpoint 1, main (argc=3, argv=0x40800374) at Add-With-Specializat
201         if (argc < 3)
(gdb) n
206         in0   = atoi (argv[1]);
// Get the users values in1 & in2
211         fPtr = (pifi) genAdd (ptr, in0); // Generate instructions
(gdb)
212         res   = fPtr(in1);   // Call generated code
(gdb) x/4i fPtr
   0x19008:     ori     t1,zero,3
   0x1900c:     add     t0,t1,a0
   0x19010:     mv      a0,t0
   0x19014:     ret
(gdb)
```

HybroGen : Transprecision Source H2

Transprecision square root source code

```
/* Newton square root demonstration with variable precision */
h2_insn_t * genIterate(h2_insn_t * ptr, int FloatWidth)
{
  #[
    flt #(FloatWidth) 1 iterate(flt #(FloatWidth) 1 u, flt #(FloatWidth) 1 val, flt
    {
        flt #(FloatWidth) 1 r, tmp1, tmp2;
        tmp1 = val / u;
        tmp2 = u + tmp1;
        return tmp2 / div;
    }
  ]#      /* r = (u + (#(value) / u))  / 2.0*/

  return (h2_insn_t *) ptr;
}
```

HybroGen : Transprecision Source Main

Transprecision square root source code (main control)

```
fPtr1 = (piff) genIterate (ptr, FLOAT);
do
{
  if ((diff < precf) && isFloat)
    {                    /* Code re-generation with double for better precision */
      fPtr2 = (pidd) genIterate (ptr, DOUBLE);
      isFloat = False;
    }
  value = next;
  next = (isFloat)?fPtr1(value, af, 2.0):fPtr2(value, af, 2.0);
  diff = ABS(next - value);
```

HybroGen : Transprecision Generated Macros

Macro instruction generation

```
/* Begin Header autogenerated part */
#include "h2-power-power.h"
#define power_G32(INSN){ *(h2_asm_pc++) = (INSN);}

void P1_BLR__I_32(void){ /* ret */
#ifdef H2_DEBUG
    printf("%p␣:␣P1_BLR__I_32\n", h2_asm_pc);
#endif
        power_G32(((0x4e800020 >> 0) & 0xffffffff)); \
}

void PPC_FADDS_RRR_F_32(int r0, int r1, int r2){ /* add */
#ifdef H2_DEBUG
    printf("%p␣:␣PPC_FADDS_RRR_F_32\n", h2_asm_pc);
#endif
        power_G32(((0x3b & 0x3f) << 26)|((r0 & 0x1f) << 21)|((r1 & 0x1f) << 16)|((r2
}
```

HybroGen : Transprecision-Generated-InstructionSelector

Macro instruction generation

```
void power_genADD_3(h2_sValue_t P0, h2_sValue_t P1, h2_sValue_t P2)
{
    if ((P0.arith == 'f') && (P0.wLen <= 32) && (P0.vLen == 1) && P0.ValOrReg == REG
    {
        PPC_FADDS_RRR_F_32(P0.regNro, P1.regNro, P2.regNro);
    }
    else if ((P0.arith == 'f') && (P0.wLen <= 32) && (P0.vLen == 1) && P0.ValOrReg =
    {
        PPC_FADDS__RRR_F_32(P0.regNro, P1.regNro, P2.regNro);
    }
    else if ((P0.arith == 'f') && (P0.wLen <= 32) && (P0.vLen == 4) && P0.ValOrReg =
    {
        V2_03_VADDFP_RRR_F_32(P0.regNro, P1.regNro, P2.regNro);
    }
    else if ((P0.arith == 'f') && (P0.wLen <= 64) && (P0.vLen == 1) && P0.ValOrReg =
    {
        P1_FADD_RRR_F_64(P0.regNro, P1.regNro, P2.regNro);
    }
    else if ((P0.arith == 'f') && (P0.wLen <= 64) && (P0.vLen == 1) && P0.ValOrReg =
    {
        P1_FADD__RRR_F_64(P0.regNro, P1.regNro, P2.regNro);
```

HybroGen : Transprecision-Generated-Compilette

Compilette Generation

```
/* Newton square root demonstration with variable precision */
h2_insn_t * genIterate(h2_insn_t * ptr, int FloatWidth)
{
/* Code Generation of 4 instructions */
/* Symbol table :*/
        /*VarName = { ValOrLen, arith, vectorLen, wordLen, regNo, Value} */
        h2_sValue_t u = {REGISTER, 'f', 1, (FloatWidth), 1, 0};
        h2_sValue_t val = {REGISTER, 'f', 1, (FloatWidth), 2, 0};
        h2_sValue_t div = {REGISTER, 'f', 1, (FloatWidth), 3, 0};
        h2_sValue_t h2_outputVarName = {REGISTER, 'f', 1, (FloatWidth), 1, 0};
        h2_sValue_t r = {REGISTER, 'f', 1, (FloatWidth), 14, 0};
        h2_sValue_t tmp1 = {REGISTER, 'f', 1, (FloatWidth), 15, 0};
        h2_sValue_t tmp2 = {REGISTER, 'f', 1, (FloatWidth), 16, 0};
        h2_sValue_t h2_00000000 = {REGISTER, 'f', 1, (FloatWidth), 17, 0};
```

HybroGen : Transprecision-Generated-Compilette

Compilette Generation Code generator

```
h2_asm_pc = (h2_insn_t *) ptr;
h2_codeGenerationOK = 1;
power_genDIV_3(h2_00000000, val, u);
power_genMV_2(tmp1, h2_00000000);
power_genADD_3(h2_00000000, u, tmp1);
power_genMV_2(tmp2, h2_00000000);
power_genDIV_3(h2_00000000, tmp2, div);
power_genMV_2(h2_outputVarName, h2_00000000);
power_genRET_0();
/* Call back code for loops */
h2_save_asm_pc = h2_asm_pc;
h2_asm_pc = h2_save_asm_pc;
iflush(ptr, h2_asm_pc);
  /* r = (u + (#(value) / u))  / 2.0*/
```

HybroGen : Transprecision-Exec

Macro instruction generation

```
qemu-ppc64le Newton.power 65536 1e-13
Compute square root of 65536.000000
With precision of 1.000000e+01 (float)
With precision of 1.000000e-13 (double)
0x100212a0 : PPC_FDIVS_RRR_F_32
0x100212a4 : P1_FMR_RR_F_32
0x100212a8 : PPC_FADDS_RRR_F_32
0x100212ac : P1_FMR_RR_F_32
0x100212b0 : PPC_FDIVS_RRR_F_32
0x100212b4 : P1_FMR_RR_F_32
0x100212b8 : P1_BLR__I_32
  0 float  : 32768.500000000000000000, 1.000000e+01
  1 float  : 16385.250000000000000000, 1.000000e+01
  2 float  : 8194.625000000000000000, 1.000000e+01
  3 float  : 4101.311035156250000000, 1.000000e+01
  4 float  : 2058.645263671875000000, 1.000000e+01
  5 float  : 1045.239868164062500000, 1.000000e+01
  6 float  : 553.969665527343750000, 1.000000e+01
```

HybroGen : Transprecision Exec 2nd part

Macro instruction generation

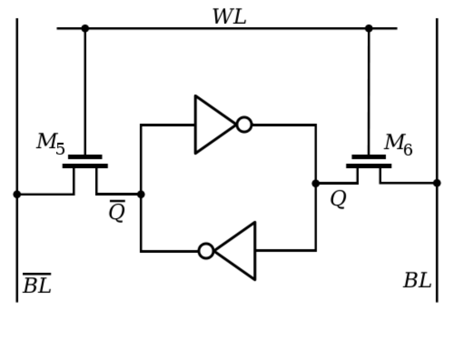
```
        7 float   : 336.1360778808593750000 0, 1.000000e+01
        8 float   : 265.5523681640625000000 0, 1.000000e+01
        9 float   : 256.1718139648437500000 0, 1.000000e+01
0x100212a0 : P1_FDIV_RRR_F_64
0x100212a4 : P1_FMR_RR_F_64
0x100212a8 : P1_FADD_RRR_F_64
0x100212ac : P1_FMR_RR_F_64
0x100212b0 : P1_FDIV_RRR_F_64
0x100212b4 : P1_FMR_RR_F_64
0x100212b8 : P1_BLR__I_32
       10 double  : 256.0000576176552158358 4, 1.000000e-13
       11 double  : 256.0000000000064801497 5, 1.000000e-13
       12 double  : 256.0000000000000000000 0, 1.000000e-13
       13 double  : 256.0000000000000000000 0, 1.000000e-13
```

Introduction : Remember Memory Cell 101

SRAM memory cell depicting Inverter Loop as gates

- 6T memory cell
- Only 1 stable mode
- Read : "open" `WL`, read value
- Write : "open" `WL`, write value

Illustration



W Memory_cell_(computing)

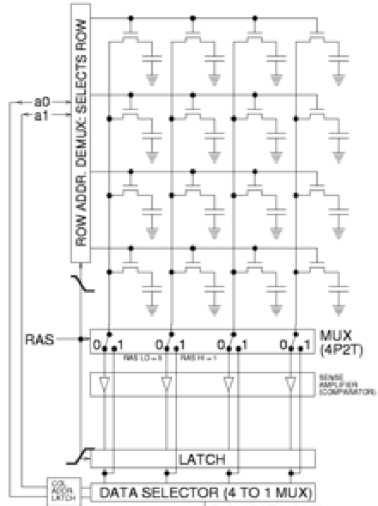
Introduction : Remember Memory 102

Functions

- Select line
- Read or write
- Potentially select word in a line
- Low voltage used; “Sense amp” to normalize

W Sense_amplifier
What every programmer should know about memory

Memory array



Intro : Classical Memory

Memory technology

- **DRAM** Dense, but need refresh (external)
- **SRAM** Less dense but faster (caches)
- **Other** RRAM, STTRAM, . . .

Memory access

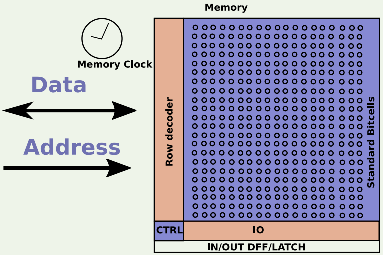
LOAD data or insn

1. Assert address
2. Read data

STORE data

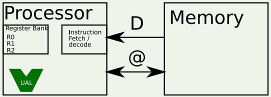
1. Assert address & data

Memory schematic

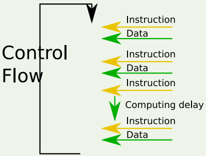


Inverted Von Neumann Programming Model

Choosen Programming model



Bus transactions



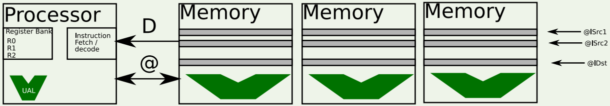
Code

ld r0 = @A

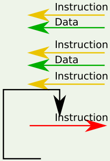
ld r1 = @B

add r2 = r0, r1

st r2 = @C



Bus transactions



Code

ld r0 = @lSrc1

ld r1 = @lSrc2

ld r2 = @lDst

add lDst = lSrc1, lSrc2

increment line adresses

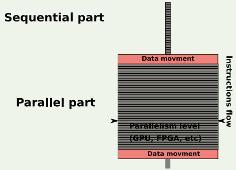
Why ?

- Allows scalability :
  - Any vector size
  - Any tile number
  - Any system configuration : near or far IMC
- Works with any processor

Von Neuman broked, what about Amhdal Law's ?

Ahmdal law's: "Speedup is limited by the sequential part"

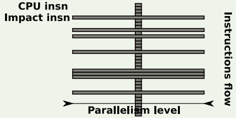
Classical approach



Programmer approach

- Has to maximize parallel part
- Deal with data "choregraphy" between CPU and GPU.

CSRAM approach

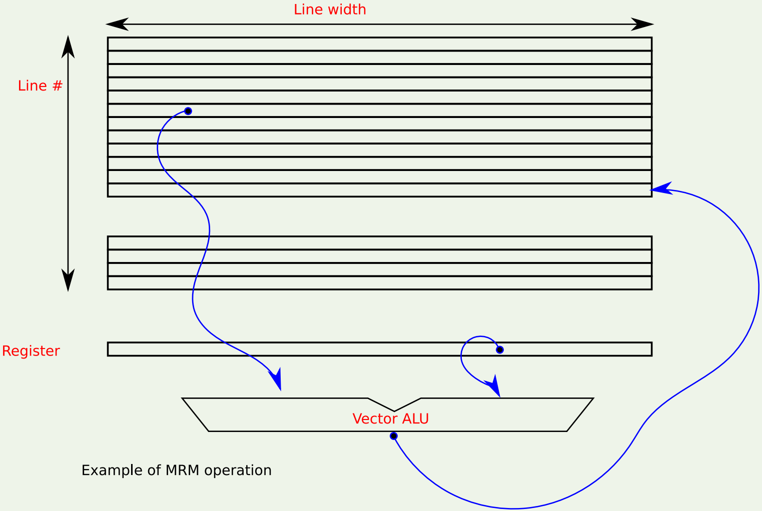


Programmer approach

- Ease to interlace scalar instruction and IMPACT instructions
- Do not move data

C-SRAM : InstructionSet

C-SRAM instruction principle



Example of MRM operation

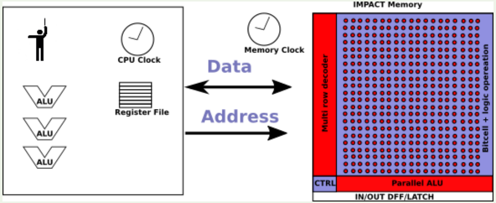
Instructions formats: release 2.2 width 64

		Opcode		Dest		Src2		Src1	
R-type	NE/6	opcode/8	RD/1	@dest/15	Mis./2	R2/1	@src2/15	R1/1	@src1/15
I-type	|c|NE/6	|c|opcode/8	|c|RD/1	|c|@dest/15	|c|Mis./2	|c|imm[15:0]/16	|c|R1/1	|c|@src1/15	
U-type	|c|NE/6	|c|opcode/8	|c|RD/1	|c|@dest/15	|c|Mis./2	|c|imm[31:0]/32			

- R-type: When all data used by the instruction are located in the memory
- I-type: When the instruction operates on a immediate value and a memory value
- U-type: When the instruction apply an immediate value directly to the memory

Architectures : IoT

Illustration IoT



- Data parallelism in IMPACT
- Loop, control, address computation in code

IoT device

- "In order" simple core (RISC-V/ROCKET, CORTEX M)
- 1 IMPACT memory, STD operators
- Instruction interleaving

Applications

- Edge IA
- Image analysis
- In memory crypto

Architectures : MPSoC

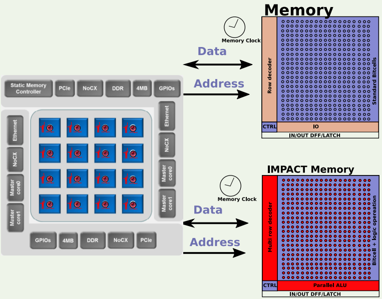
Multiple architectural choices

- One IMPACT memory per core : local synchronous computation
- One IMPACT memory per MPSoC : dedicated core / CSRAM, other for applications

Applications

- IMPACT for large parallel synchronous computation
- MPSoC for asynchronous computation

Architecture example



Architectures : Near Sensor Computing

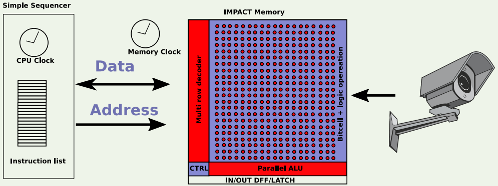
Near Sensor Computing

- 1 sensor, e.g. a camera
- 1 automaton to send instructions
- 1 IMPACT memory,
- special operators (stochastics ?)

Applications

- Motion detection
- Pattern detection
- Other

Illustration : edge computing



Architectures : Computing

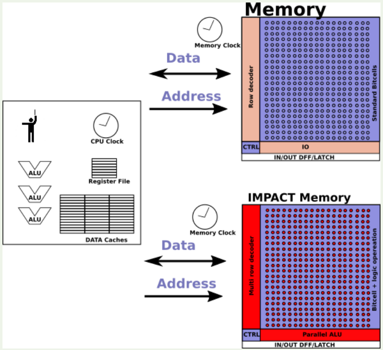
Computing node

- Complex core (RISC-V/BOOM, CORTEX A)
- Multiple IMPACT memory, STD operators
- Dynamic RAM

Applications

- Database
- Convolutions
- DNA matching

Illustration: complex hierarchy



Programming Model : Image Diff

Mini code Example : HybroLang code example

```
int 8 16 ImageDiff(int[] 8 16 a,
                   int[] 8 16 b,
                   int[] 8 16 res, int 31 1 len)
{
    int 31 1 i;                          /* Scalar variable */
    for(i = 0; i < len; i = i + 1)       /* CPU Control */
    {
        res[i] = a[i] - b[i];            /* IMC instructions */
    }
}
```

Compiler support

- Dynamic interleaving
- Instruction generator generator notion

HybroGen : ImageDiff-Build

Build and run an application

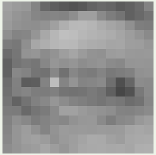
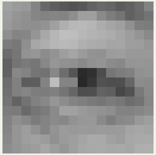
```
RunDemo.py -a cxram -i CxRAM-ImageDiff
Namespace(arch=['cxram'], clean=False, debug=False, inputfile=['CxRAM-ImageDiff'], v
-->rm -f CxRAM-ImageDiff CxRAM-ImageDiff.c
-->which riscv32-unknown-linux-gcc
-->../HybroLang.py --toC --arch cxram --inputfile CxRAM-ImageDiff.hl
-->riscv32-unknown-linux-gcc -o CxRAM-ImageDiff CxRAM-ImageDiff.c
('MonOeilGrisFerme.pgm', 'MonOeilGris.pgm')
-->qemu-riscv32 CxRAM-ImageDiff MonOeilGrisFerme.pgm MonOeilGris.pgm
```

HybroGen : ImageDiff-Run

CxRAM Usage

- Compute image difference
- Iterate on image lines (RISCV)
- Use difference operators / 16 pixels wide (CxRAM)

Dataset

Research Model Organization

Actual Organization

- Research in memory design
  - Build memory chips (integrate High Level evaluation, specialized instructions)
  - Characterize behavior
- Research in compilation
  - Build simulation model (integrate characterization)
  - Build Software Tools
  - Evaluation on High Level Applications
- Multiples targets :
  - Support current & old chips
  - Invent / Test / support futures features
  - Evaluate futures features, using previous characterization
  - Generate test codes

Lessons learned

- Do not work on "one project"
- Connect & Interact, ... frequently
- Sort of agile Research Model

Freedom to move but interact frequently