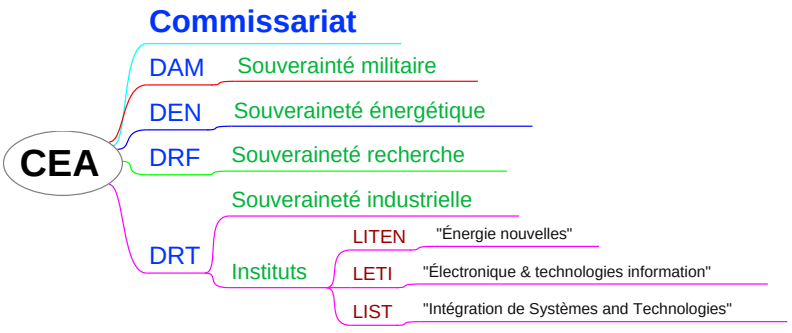
Quels choix possibles dans les modèles d'exécution bas niveau ?

ou comment réconcilier design et compilateurs

Henri-Pierre CHARLES

CEA DSCIN department / Grenoble

01 - 03 Février 2023


Commissariat
CEA
DAM
Souveraineté militaire
DEN
Souveraineté énergétique
DRF
Souveraineté recherche
Souveraineté industrielle
DRT
Instituts
LITEN
"Énergie nouvelles"
LETI
"Électronique & technologies information"
LIST
"Intégration de Systèmes and Technologies"

Résumé

Le besoin de réduire l'énergie consommée par les architectures de calcul conduit les concepteurs de circuit à concevoir des architectures dont le modèle d'exécution bas niveau est de plus en plus complexe et ne sont plus basés sur le classique modèle de Von Neumann.
Cet exposé montrera les modèles d'exécution classiques et moins classiques que l'on peut inventer pour l'utilisation de ces nouveaux accélérateurs.

En pratique

Exemple sur l'accélérateur CSRAM (Computational SRAM)

- Comment ça fonctionne
- Comment on compile

Peak versus sustained

- Peak performance is the max theoretical possible performance

TOP500 using dense arithmetic

- Linpack (dense matrix multiply)
- TOP 500 nov 2022
- Top 1 : 8,730,112 AMD EPYC 64 cores
- 1,102.00 PFlops / s
- Top 10 : Efficiency betwen 61% and 82%

TOP500 using indirect memory access

- HPCG (congugate gradient using indirect memory access)
- Top 10 : efficiency between **0.38% and 2.98%**
- 99% wasted energy

Dense arithmetic was driving computer design

Algorithms

- Sparse linear algebra
- Physical simulation
- Video filtering / compression
- IA learning
- IA inference

Challenges

- Architectural support ? Need new basic programming models
- Langage support, how to express new applications ? (Don't say with GPT, nor with C)
- Compiler support ? How to make link between langage programming model and hardware basic programing models
- Arithmetic support ? (for optimization)

N. Wirth : "Algorithm+ Data Structure = Programs"

- Old idea
- New architectures make performance very sensitive to data set characteritics (values, layont, size)
- Message 1 : binary code should depend of the run-time data set.

Hennessy & Patterson

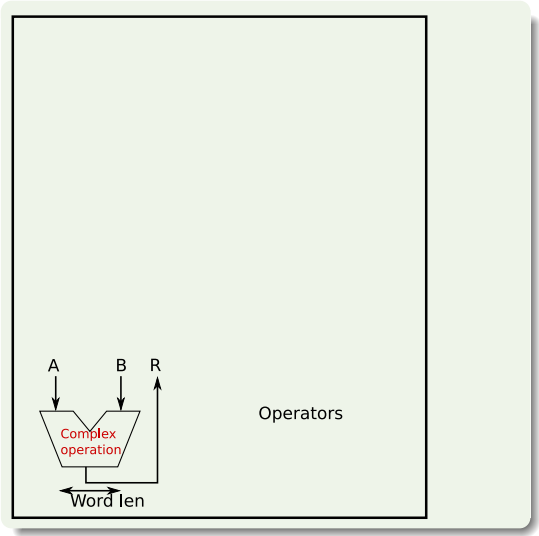
- Use new programming paradigms
- Use DSL (Domain Specific Languages)
- Message 2 : Use specific compilers (and code generators)

Simple example

- Programming language arithmetic are `int` or `float`. Why ?
- Word len could be between 8 and 512
- Vector length .. for another presentation

Expression example

- `r += a*b`
- The “famous” MAC instruction



Designer

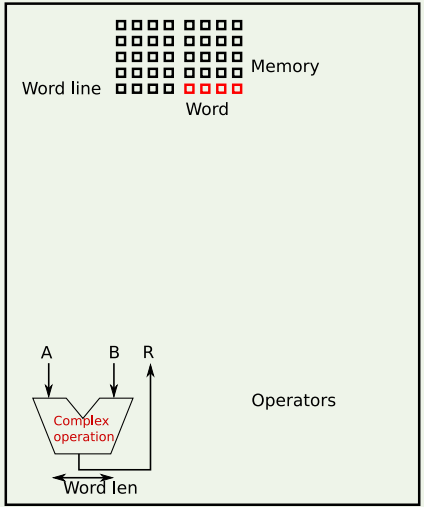
- Minimize transistor count / surface
- Minimize energy (technology)
- Minimize delay (design, pipeline)

Compiler writer

- Which arithmetic ?
- How to recognize in program source ?
- Operation timing
- Input / output operator timing access
- How to select

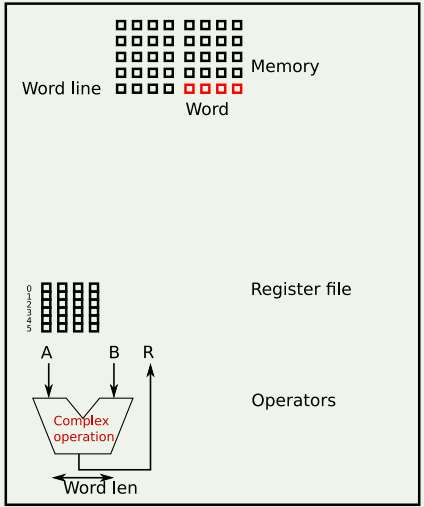
Arithmetic logic unit

Programming Model : Memory for data



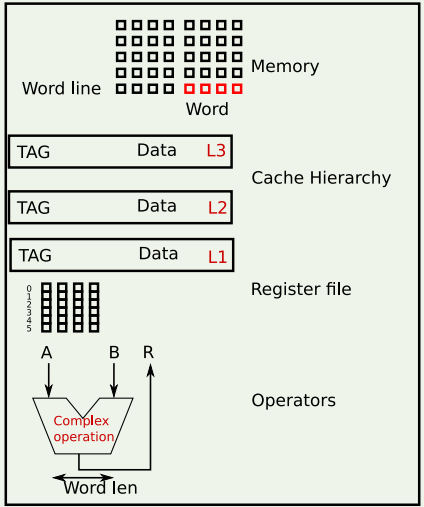
Illustration

- Compiler is reponsible of the data layout
- `a += b * c`
- Data allocation :
- `0x0000800000004970 +=`
  `0x0000800000004974 *`
  `0x0000800000004978`



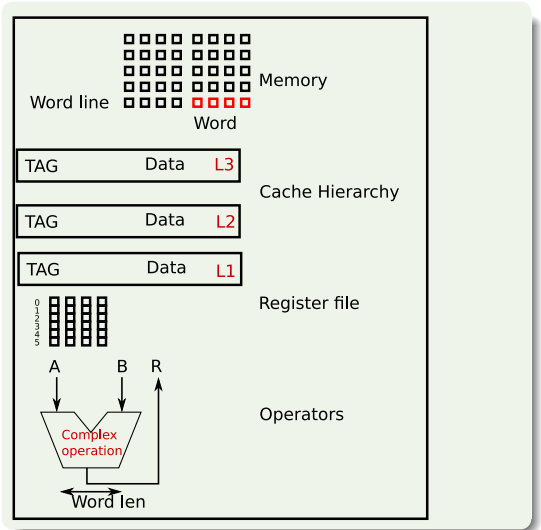
Registers as reduced address size

- `a += b * c`
- Data allocation :
- 0x0000800000004970 += 0x0000800000004974 * 0x0000800000004978
- Replaced by register allocation
- data move
  - `ld R4 @b`
  - `ld R3 [R4]`
  - `ld R4 @c`
  - `ld R2 [R4]`
  - `R1 += R2 * R3`
  - `ld R4 @a`
  - `st [R4] R1`

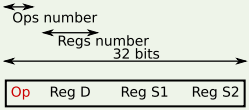


Behavior

- Cache as “memory accelerator”
- Statistical performance
- No programmer action... Really ?
- Nice way to use transistor budget



Illustration



- Where are stored the instructions ? In memory off course
- Single instruction flow is mandatory for compiler optimization & scheduling

CPU Algorithm : a “simple automata”

- Forever do :
  1. Fetch instruction (IF)
  2. Increment PC
  3. Decode instruction (DE)
  4. Execute (EXE)
  5. Store result (WB)

Comments

1. Algorithm duration = sum of each instructions
2. Irregular duration : memory access, complex
3. Each step can be divided in sub steps
4. X86 micro operations

Programming Model : Von Neumann

Modèle basique

- Processing unit with UAL & Registers
- Control unit with insn register & program counter
- Memory for data AND instructions
- Memory and I/O

Many other models

- Harvard
- DSP / GPU /
- New ones !

Von Neumann architecture

Programming Model : From Language To Instructions

Compiler for standard core

- Lexer + Parser (main focus of compiler course ...)
- Intermediate form is a key (success of LLVM)
- Find the correct optization flag

For new basic programming model

- Keep “instruction” model
- Keep the single instruction flow
- Make instructions “simple”

SOA ARCH : ARM big.LITTLE

big

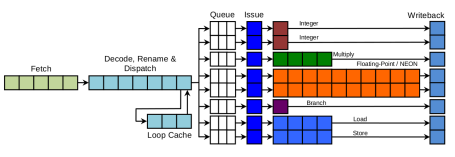


Figure 2 Cortex-A15 Pipeline

LITTLE

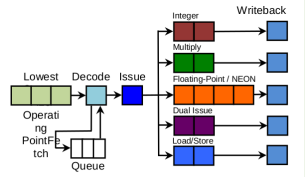


Figure 1 Cortex-A7 Pipeline

ARM : "more than 30 billion processors sold with more than 16M sold every day ARM" (Nov 2013) `http://www.arm.com/products/processors/index.php`

- 4 big processors + 4 little
- Same ISA, ...
- (even for vector operation)
- Low latencie switch

big.LITTLE notion

What every programmer should know about performance

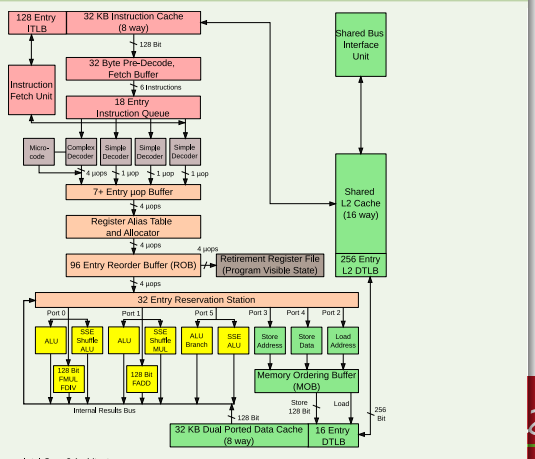
- Hidden micro architecture
- Memory hierarchy

Intel Example

- “700” simple high level instructions
- 17000+ instructions variants
- RISC internal uInstructions

Intel Micro architecture

Intel Core2 micro arch



Intel Core 2 Architecture

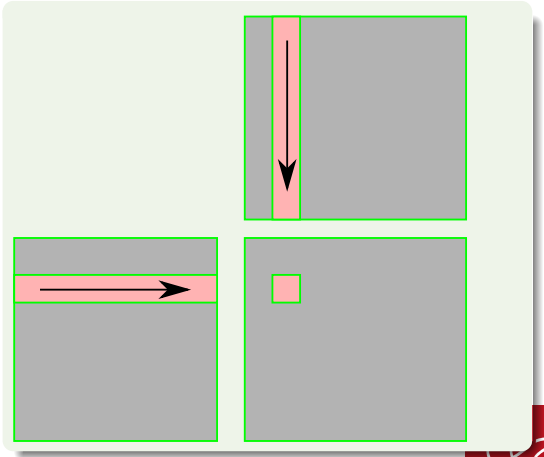
Matrix multiply (sketch)

```
for (int l = 0; l < SIZE; l++)
  for (int c = 0; c < SIZE; c++)
    for (int k = 0; k < SIZE; k++)
      R[l][c] += A[l][k] * B[k][c];
```

"Real world"

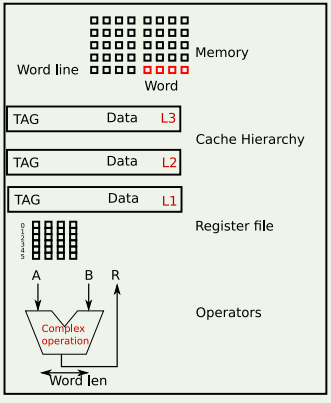
```
for (c= 0; c<NCOL; c+=cacheLineSize)
 for (l= 0; l<NLINE; l+=halfCacheLine)
  for (c2= 0; c2<NCOL; c2+=halfCacheLine)
    for (lk= 0; lk<halfCacheLine; lk++)
     for (c2k= 0; c2k<halfCacheLine; c2k++)
      for (ck= 0; ck<cacheLineSize; ck++)
        res[l+lk][c2+c2k]+=      a[l+lk] [c+ck]* b[c2+c2k][c+ck];
```

Learn to program = learn to serialize / schedule on defined hardware !



Other "interesting examples"

RISC



CISC

"Complex" versus "Reduced" has no meaning.

- RISC : compute instructions, memory instructions
- CISC : compute instructions with memory access (need microcode)

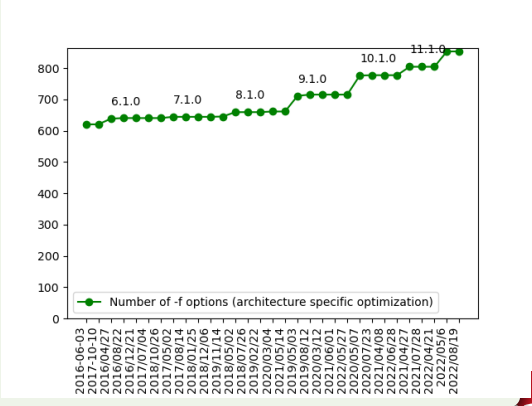
Compiler life (gcc)

- more than 40 year
- more than 100000 files
- precursor in terms of “eco conception”

Compiler Contains

- SSA form : program as transformable data. Program transformation : parallelization, vectorization ...
- Register allocation.
- Instruction scheduling : based on data type arithmetics
- Asumptions about target
- Pattern matching for low level instructions selection

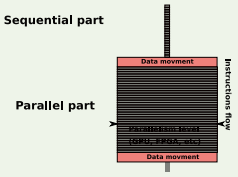
Illustration



Von Neuman broked, what about Amhdal Law's ?

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

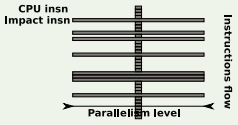
Classical approach



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

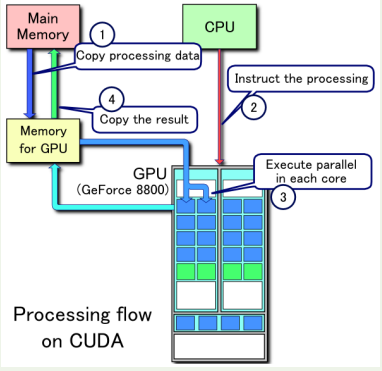
Characteritics

- Programming language for data parallelism
- CUDA

Illustration

- Normalized: No
- Portable : No (NVIDIA only)
- Scalable : No, why ?
- Asynchronous : no way to schedule
- At least 2 instruction flow
- Data dependent essential to performance (blocking)
- Need hardware thread support
- Specialized ISA (PTX)

Example of CUDA processing flow



GPU CUDA-Example

Header and CUDA code

```
import pycuda.compiler as comp
import pycuda.driver as drv
import numpy
import pycuda.autoinit

mod = comp.SourceModule("""
__global__ void multiply_them(float *dest,
{
  const int i = threadIdx.x;
  dest[i] = a[i] * b[i];
}
""")
```

Python code

```
multiply_them = mod.get_function("multiply_

a = numpy.random.randn(400).astype(numpy.f
b = numpy.random.randn(400).astype(numpy.f

dest = numpy.zeros_like(a)
multiply_them(
        drv.Out(dest), drv.In(a), drv.In(b
        block=(400,1,1))

print dest-a*b
```

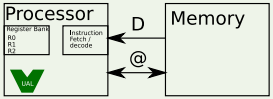
Examples : UpMEM

Characteritics

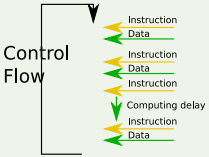
- Processing in DRAM
- Multipe instruction flows
- OpenMP support
- `https://github.com/upmem`
- Scalar computation
- Hardware thread support
- Specialized ISA

Inverted Von Neumann Programming Model

Choosen Programming model



Bus transactions Code

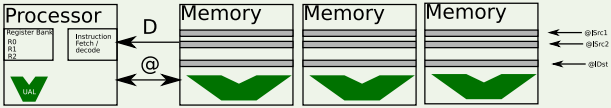


ld r0 = @A

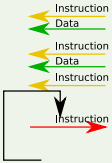
ld r1 = @B

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

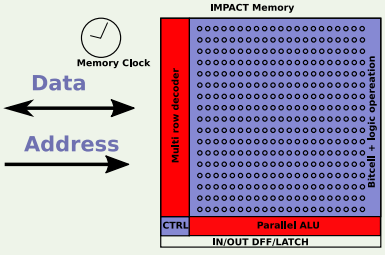
Memory with

- Bitcell 2 ports
- Vector like alu
- Multi-line selector
- No internal instruction execution

Design choices

- Minimize CPU modification
- Only one instruction sequence (CPU master)

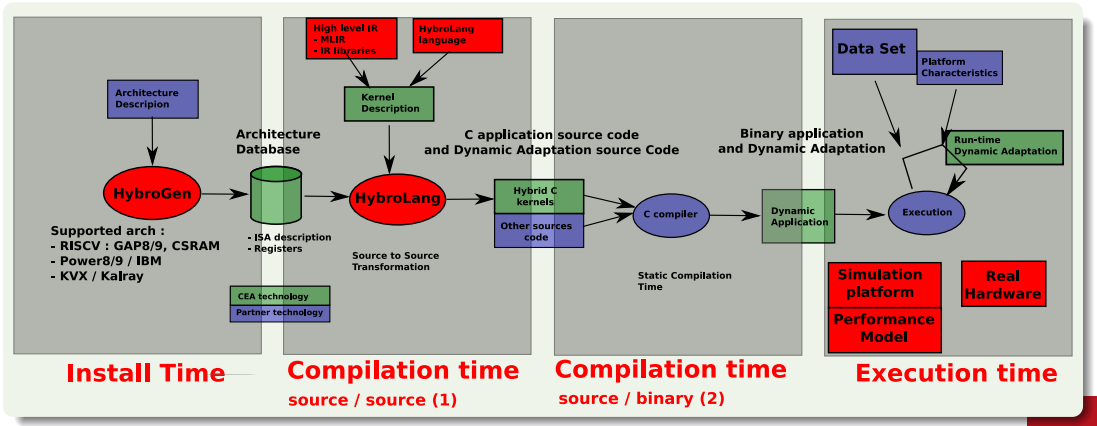
IMPACT memory



Technology

SRAM technology (FDSOI 22nm)

HybroGen: General View



Mini code Example : HybroLang code example

```
pifiii genSubImages(h2_insn_t * ptr)
{
#[
  int 32 1 subImage(int[] 16 8 a, int[] 16 8 b, int[] 16 8 res, int 32 1 le
        {
          int 32 1 i;
          for (i = 0; i < len; i = i + 1)
                {
                  res[i] = a[i] - b[i];
                }
        }
  return 0;
]#
  return (pifiii) ptr;
}
```

Compiler support

Main program

```
readImage (argv[1], in1);
readImage (argv[2], in2);
ptr = h2_malloc (1024);
code = genSubImages(ptr); // Code generation
code (in1, in2, res, 32); // Code call
```

Build and run an application

```
RunDemo.py -a cxram -i CxRAM-ImageDiff
Namespace(arch=['cxram'], clean=False, debug=False, inputfile=['CxRAM-Image
-->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
```

Main program

```
#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_SLLI_RRI_I_32_1(r3,r1,i0) /* SL */ \
do { \
        riscv_G32(((0x0 & 0x7f) << 25)|((i0 & 0x1f) << 20)|((r1 & 0x1f) <<
} while(0)

#define RV32I_SLL_RRR_I_32_1(r3,r1,r2) /* SL */ \
do { \
        riscv_G32(((0x0 & 0x7f) << 25)|((r2 & 0x1f) << 20)|((r1 & 0x1f) <<
} while(0)

#define RV32I_ADDI_RRI_I_32_1(r1,r0,i0) /* ADD */ \
```

Main program

```
void riscv_genSL_3(h2_sValue_t P0, h2_sValue_t P1, h2_sValue_t P2)
{

    if ((P0.arith == 'i') && (P0.wLen <= 32) && (P0.vLen == 1) && P0.ValOrRe
    {
        RV32I_SLLI_RRI_I_32_1(P0.regNro, P1.regNro, P2.valueImm);
    }

    else if ((P0.arith == 'i') && (P0.wLen <= 32) && (P0.vLen == 1) && P0.Va
    {
        RV32I_SLL_RRR_I_32_1(P0.regNro, P1.regNro, P2.regNro);
    }

    else
    {
        printf("Warning, generation of SL is not possible with this arg
        h2_codeGenerationOK = false;
    }
```

Compilette

```
pifiii genSubImages(h2_insn_t * ptr)
{
/* Code Generation of 51 instructions */
/* Symbol table :*/
        /*VarName = { ValOrLen, arith, vectorLen, wordLen, regNo, Value} */
        h2_sValue_t a = {REGISTER, 'i', 1, 32, 10, 0};
        h2_sValue_t b = {REGISTER, 'i', 1, 32, 11, 0};
        h2_sValue_t res = {REGISTER, 'i', 1, 32, 12, 0};
        h2_sValue_t len = {REGISTER, 'i', 1, 32, 13, 0};
        h2_sValue_t h2_outputVarName = {REGISTER, 'i', 1, 32, 10, 0};
        h2_sValue_t i = {REGISTER, 'i', 1, 32, 5, 0};
        h2_sValue_t h2_00000009 = {REGISTER, 'i', 1, 32, 6, 0};
        h2_sValue_t h2_00000012 = {REGISTER, 'i', 1, 32, 6, 0};
```

Main program

```
riscv_genLABEL(h2_prologue_00000000);
riscv_genMV_2(i, (h2_sValue_t) {VALUE, 'i', 1, 32, 0, 0});
riscv_genLABEL(h2_begin_00000001);
riscv_genBGE_3(i, len, (h2_sValue_t) {VALUE, 'i', 1, 32, 0, (int)((
riscv_genLUI_2(h2_00000046, (h2_sValue_t) {VALUE, 'i', 1, 32, 0, 53
riscv_genADD_3(h2_00000050, res, i);
riscv_genMV_2(h2_00000051, (h2_sValue_t) {VALUE, 'i', 1, 32, 0, 2})
riscv_genSL_3(h2_00000051, h2_00000050, h2_00000051);
riscv_genOR_3(h2_00000051, h2_00000046, h2_00000051);
riscv_genADD_3(h2_00000055, a, i);
riscv_genADD_3(h2_00000059, b, i);
riscv_genMV_2(h2_00000060, (h2_sValue_t) {VALUE, 'i', 1, 32, 0, 16}
riscv_genSL_3(h2_00000060, h2_00000059, h2_00000060);
riscv_genOR_3(h2_00000060, h2_00000055, h2_00000060);
riscv_genW_3(h2_00000051, h2_00000060, (h2_sValue_t) {VALUE, 'i', 1
riscv_genMV_2(h2_00000065, (h2_sValue_t) {VALUE, 'i', 1, 32, 0, 1})
riscv_genADD_3(i, i, h2_00000065);
riscv_genBA_2((h2_sValue_t) {REGISTER, 'i', 1, 32, 0, 0}, (h2_sValue
```

CxRAM Usage

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

Dataset

CxRAM-Status : Software Tools

Code generation

- HybroGen compiler first public release `https://github.com/CEA-LIST/HybroGen`
- Emulator, based on QEMU : not (yet) open source
- Full development chain :
  - Cross compiler (gcc)
  - Cross source level debugger (gdb)
- 5 CsRAM instruction set variants support

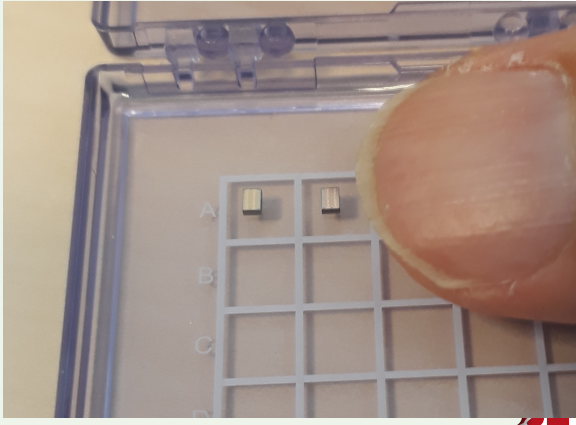
High level Applications

- 

OpenCV support (image filtering)
- 
Tensor flow lite
- Working on :
  - Cryptography

Chip design evolution

- 1 chip built, characterized : CSRAM part only, (photo)
- Result published : "A 35.6TOPS/W/mm^2 3-Stage Pipelined Computational SRAM with Adjustable Form Factor for Highly Data-Centric Applications"
- 1 chip built, under testing : CSRAM + RISCV
- Ongoing work on new instruction set



Informations

- `https://hpcharles.wordpress.com/`

PhD position

- Long description
  - Micro compilation
  - Ternary Neural Network
  - Using near memory computing
- Possible other PhD positions