# My personnal view on
# - Computing in/near Memory
# - Low Level Programming model
# - Compiler Evolution

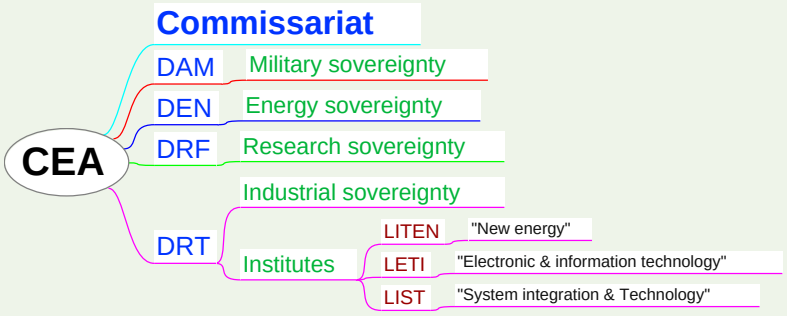
InterFLOP https://www.interflop.fr/

Henri-Pierre CHARLES

CEA DSCIN department / Grenoble

2024-06-12

# Introduction : Présentation-CEA



# Programming Model : Arithmetic Expression

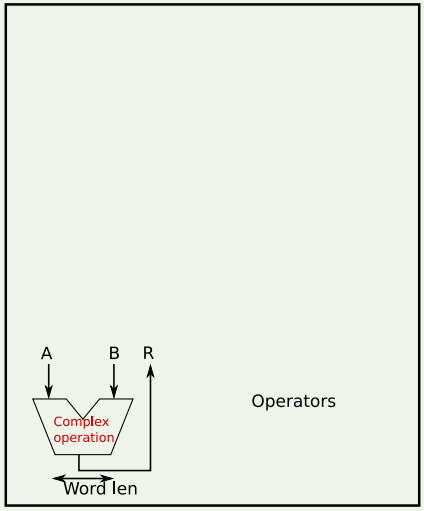
## Simple example

- 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

# Programming Model : Basic Operator




**Designer**

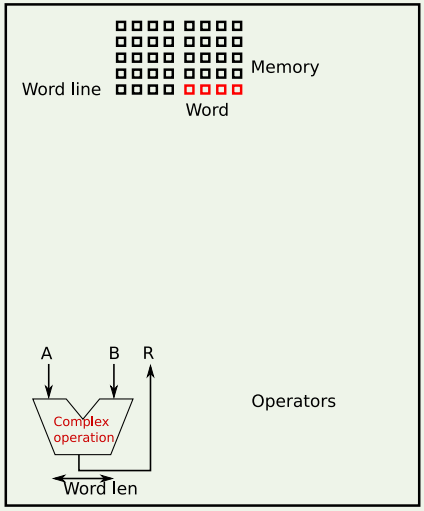
- Minimize transistor count / surface
- Minimize energy (technology)
- Minimize delay (design, pipeline)

**Compiler writer**

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

Arithmetic logic unit

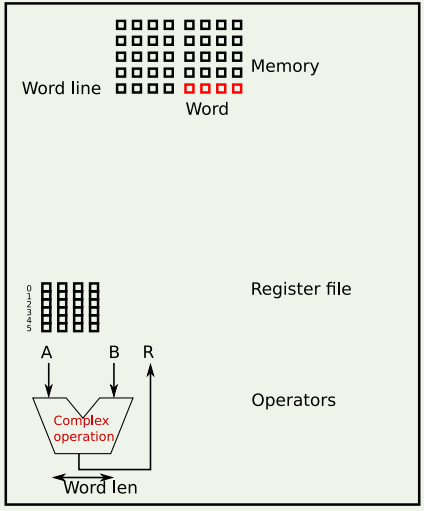
# Programming Model : Memory for data



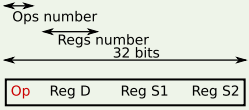

## Illustration

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

# Programming Model : Instruction

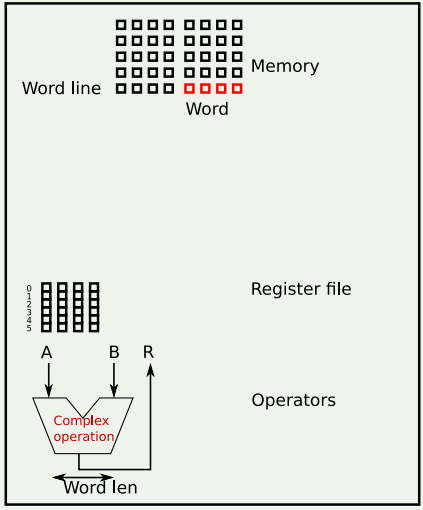



## Illustration




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

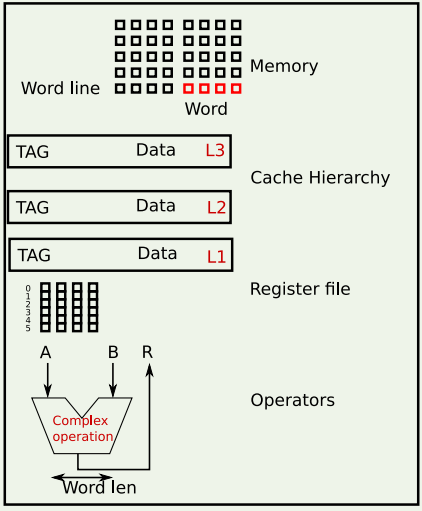
# Programming Model : Registers




## 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`

# ProgrammingModel : Memory hierarchy




## Behavior

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

# ProgrammingModel : Instruction Decode Algorithm

## 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

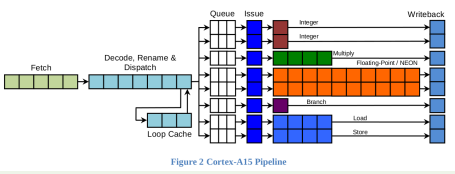



Figure 2 Cortex-A15 Pipeline

## LITTLE

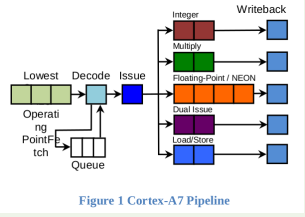



Figure 1 Cortex-A7 Pipeline

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

# HWParallelismLevel uArch

## What every programmer should know about performance

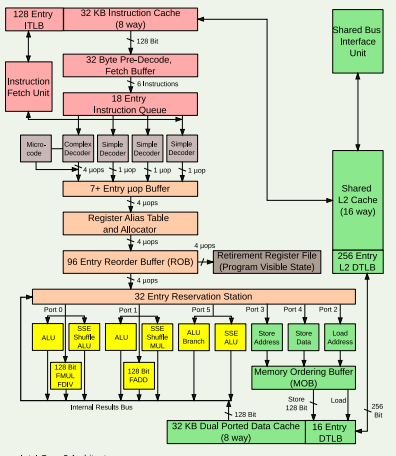
- Hidden micro architecture
- Memory hierarchy

## Intel Example

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

Intel Micro architecture

## Intel Core2 micro arch




Intel Core 2 Architecture

# Models : C Language for Architecture

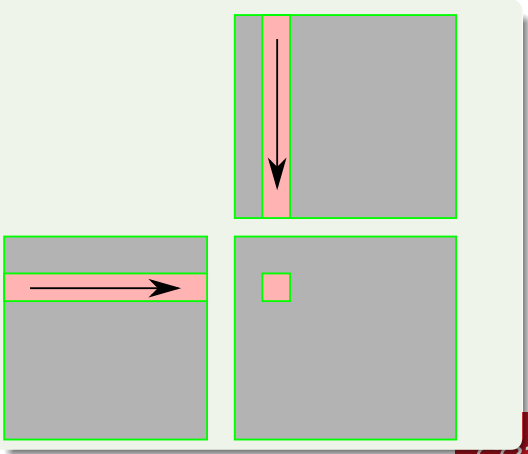
## Matrix multiply (sketch)

```
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];
```

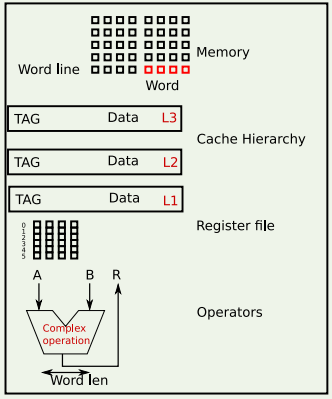
Learn to program = learn to serialize / schedule on defined hardware !



Other "interesting examples"

# Introduction : CISC-versus-RISC

## RISC




## CISC

"Complex" versus "Reduced" has no meaning.

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

# Programming Model : Model-and-Compiler

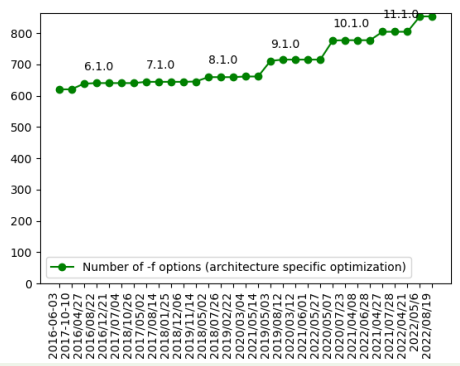
## Compiler life (gcc)

- 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



# GPU / CUDA

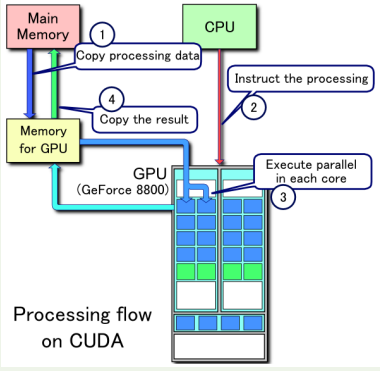
## Characteritics

- 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, float
{
  const int i = threadIdx.x;
  dest[i] = a[i] * b[i];
}
""")
```

## Python code

```
multiply_them = mod.get_function("multiply_them")

a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

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

print dest-a*b
```

# GPU-CUDA : Low Level

## Programming language

- Kernel extraction approach (or embed into scripts)
- Heterogeneous compilation, native for host
  - Compilation to IR (PTX), with JIT compilation
  - Compilation to binary
- Deal with code divergence

## Compilation

- Use API
- Use kernel extractor (vncc, ...)

## Low level

- PTX “assembly language”
- Crucial importance of data / processor mapping
- Need to pay for memory copy

# Examples : UpMEM


## Characteritics

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

## UpMEM

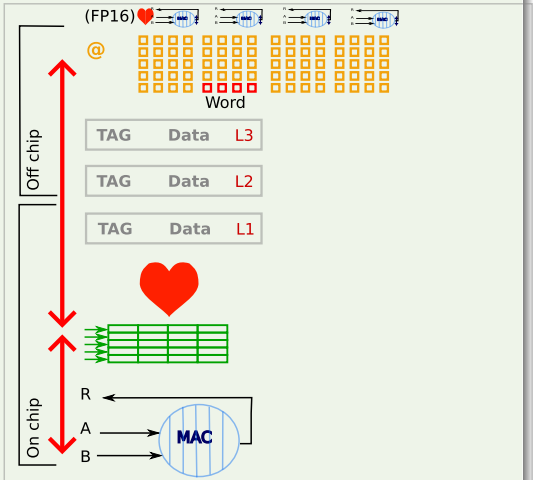
- How to deal with DRAM latency (Hyperthreading)
- How to deal with interleaved memory (Programmer concern)

## Low level

- OpenMP based programming thread model
- Heterogeneous ISA

# Examples : Samsung Aquabolt-XL HBM2-PIM

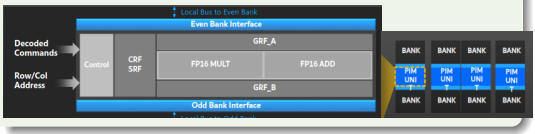
## CPU / PIM




## Argumentation

- 9 instructions
- micro programs with control flow
- usage via “PIM aware software stack”
- no direct access to compiler

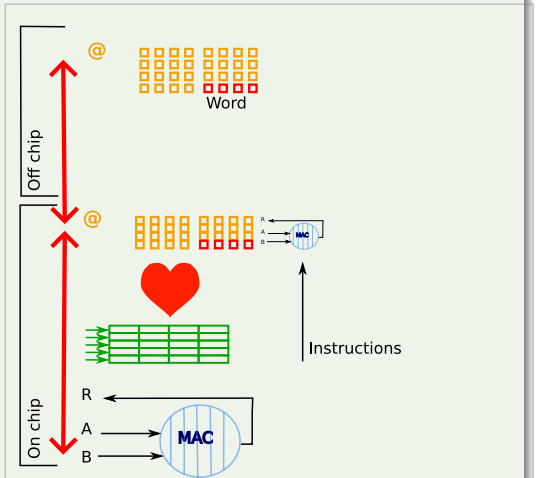
## Illustration




Hot Chips Jin Kim 2021 slides

# Examples : CSRAM (Computational SRAM)

## Architecture




## Programmer view

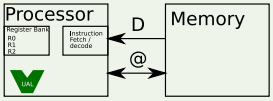
- Two source code, but
- Single program flow
- Non Von Neumann model : CPU send instructions to CSRAM
- DSL approach, which express
  - Heterogeneous computation (DONE)
  - Memory hierarchy (ONGOING)

## Compiler

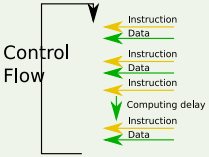
- `https://github.com/CEA-LIST/HybroGen`
- Functionnal emulator (based on QEMU):

# Inverted Von Neumann Programming Model

## Choosen Programming model




Bus transactions Code

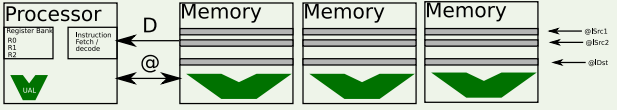



ld r0 = @A

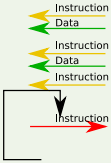
ld r1 = @B

add r2 = r0, r1

st r2 = @C




Bus transactions Code




ld r0 = @Src1

ld r1 = @Src2

ld r2 = @Dst

add @Dst = @Src1, @Src2

increment line adresses

## Why ?

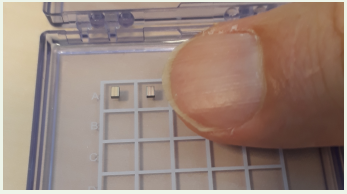
- Allows scalability :
  - Any vector size
  - Any tile number
  - Any system configuration : near or far IMC
- Works with any processor

# CxRAM-Status : Circuit

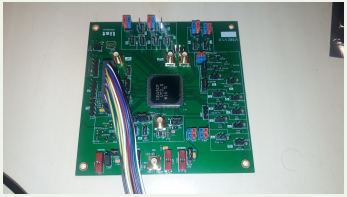
## Chip design evolution

- 1 chip built, characterized : CSRAM part only, (photo)
- Result published : "A 35.6TOPS/W/mm² 3-Stage Pipelined Computational SRAM with Adjustable Form Factor for Highly Data-Centric Applications" 2020
- 1 chip built, under testing / characterization : CSRAM + RISCV
- Ongoing work on new instruction set variants

## IMPACT circuit (2019)



## RISCV and CSRAM under testing (2023)



# IMC Conclusion : Important points

## Facts

- Computation is free (mostly)
- The challenge is to avoid data movement
- Using analogous computation has huge gains, but need DACs

## Technical points

- Data layout become the important point
- DRAM use data interleaving ...
- What are the available arithmetics ?

## Industrial points

- There is no standard :
  - Programming language
  - Data movement expression
- What's the driving application ?

# Compiler : CompilerSupport

## "My accelerator has compiler support !"

Many definitions :

- Write in control register + busy wait
- Use "asm intrinsic"
- Use high level function call
- Use well "defined arithmetics"
- Integrated in high level IR
- Scheduling outside compiler

## Examples

- MPI : "data movment arithmetic" ?
- BLAS : function call versus tensor flow lite loop nest
- saturated arithmetic + RGB arithemtic for pixels
- controlled precision for FP arithmetics

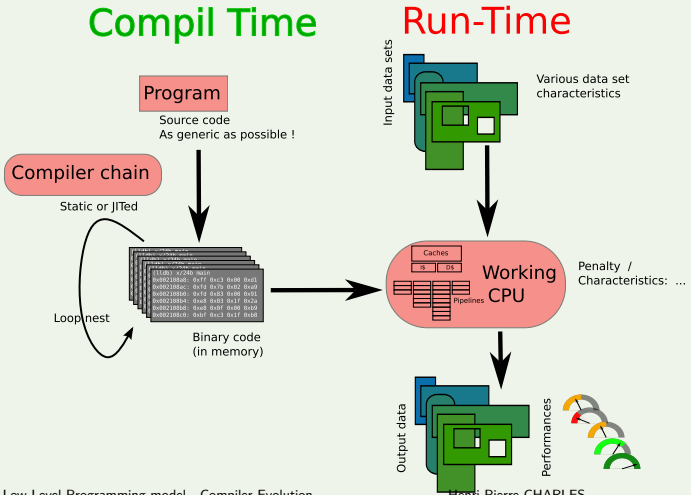
Amdahl law's is forever !

# Motivation : Static Compiler and notion of "*Compilette*"

## Static Compiler

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

## Static compilation



# Motivation : Static Compiler and notion of "*Compilette*"

## Static Compiler

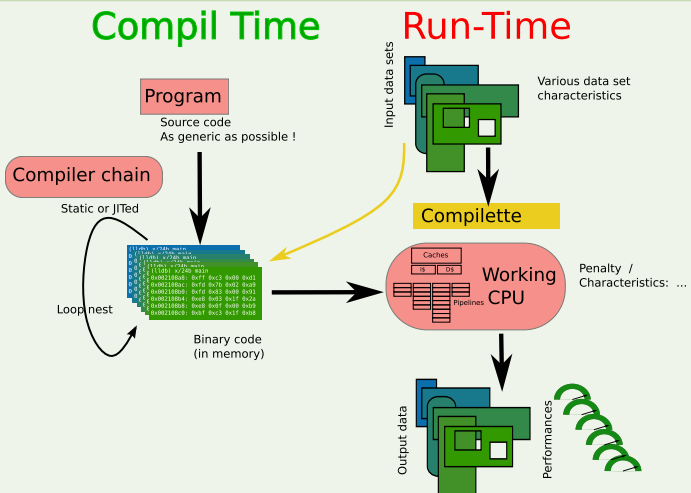
- Run once
- Does not know data set characteristics
- Slow compilation (even with JIT)

## Compilette improve performance

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

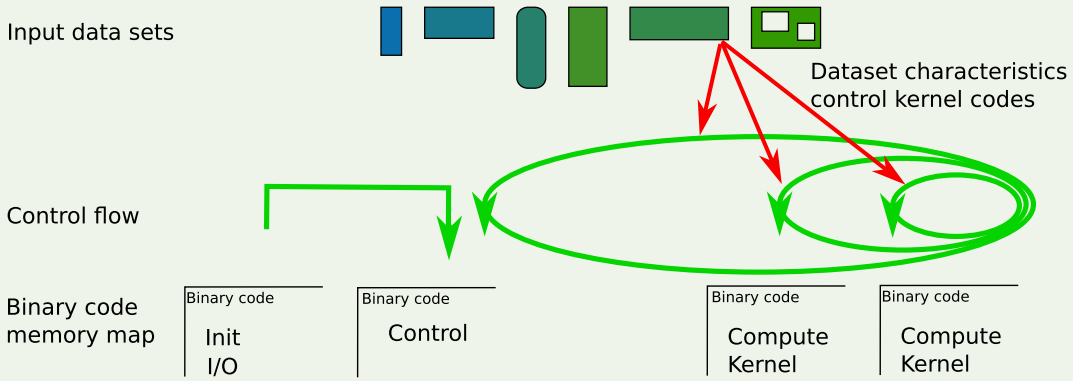
How to implement / program that ?

## Static compilation with Dynamic adaptation



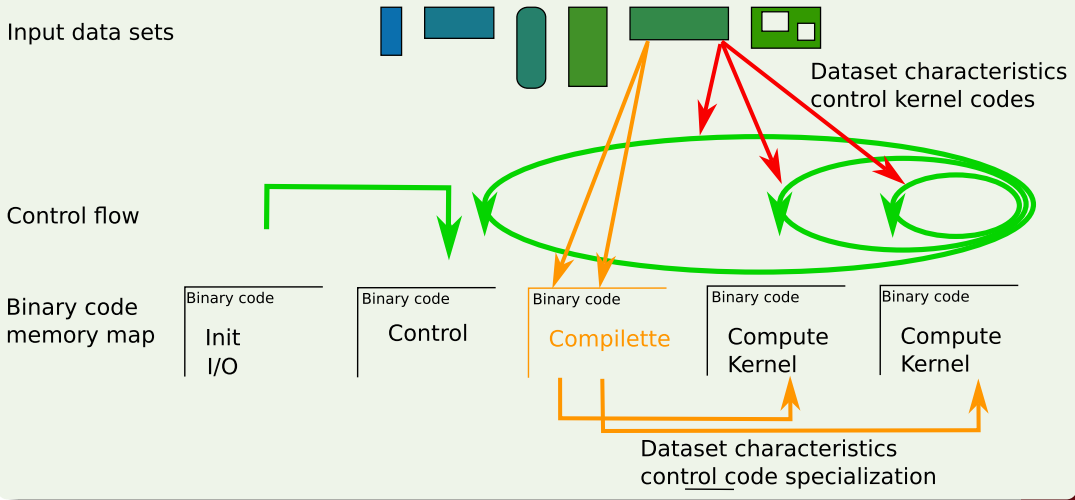
# Compilette principle : "Working Example"

## Control flow in application



# Compilette principle : "Working Example"

## Control flow in application with dynamic adaptation



# List of Code Generation Scenarios

## Compilation scenarios

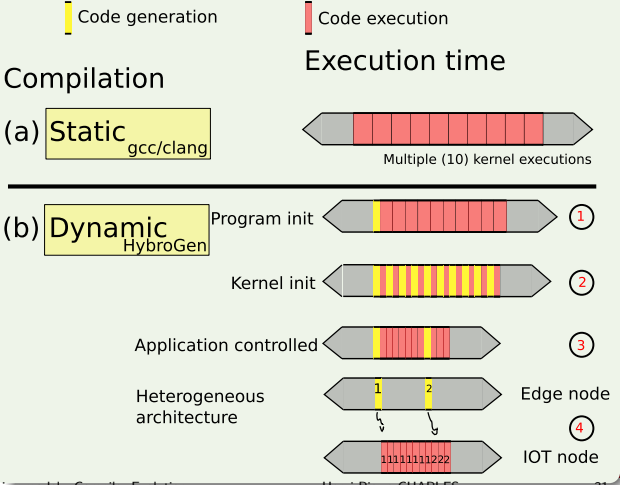
- (a) Static compilation
- (b) Dynamic adaptation
  1. Program initalization
  2. Kernel initialization
  3. Application controlled
  4. Heterogeneous architecture (multi-isa support)

## Target list

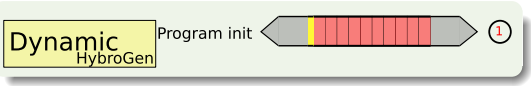
- RISCV
- RISCV + CSRAM
- POWER 8
- AARCH64
- Others

All scenarios examples works on all platforms

## Illustration



# Scenario #1 : Application Initialization




## Generate code at initialization & reuse

- Binary code generated at initialization time
- Demonstrator based on Celcius to Farenheit conversion

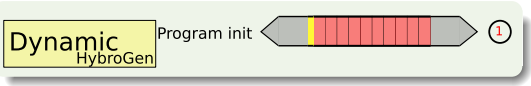
## Optimization source

- Optimization based on initialization time knownledge
  - Input parameters
  - Data set characteristics

## gcc -O3 version

```
0000000000402d34 <CelciusToFarenheit>:
 402d34: add w0, w0, w0, lsl #3
 402d38: mov w1, #0x6667    // #26215
 402d3c: movk w1, #0x6666, lsl #16
 402d40: smull x1, w0, w1
 402d44: asr x1, x1, #33
 402d48: sub w0, w1, w0, asr #31
 402d4c: add w0, w0, #0x20
 402d50: ret
```

# Scenario #1 : Application Initialization




## Generate code at initialization & reuse

- Binary code generated at initialization time
- Demonstrator based on Celcius to Farenheit conversion

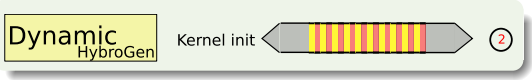
## Optimization source

- Optimization based on initialization time knownledge
  - Input parameters
  - Data set characteristics

## Compilette version

```
(gdb) x/20i fC2F
0x4162a0: mov w10, #0x9
0x4162a4: mul w10, w0, w10
0x4162a8: mov w11, #0x5
0x4162ac: sdiv w11, w10, w11
0x4162b0: add x0, x11, #0x20
0x4162b4: ret
```

# Scenario #2 : Kernel Initialization




## Value injection / dynamic polymorphism

```
#[  int 32 1 mult (int[] 32 1 a,
                   int   32 1 len)  {
    int 32 1 r, i;
    for (i = 0; i < len; i = i + 1) {
        // b value will be included
        //in code generation
        a[i] = a[i] * #(b);
      }
  }  ]#
```

## Optimization source

- Optimization based on kernel input parameters

## POWER Binary codes specialization on 42

```
(gdb) x/20i fPtr
   0x10021710:  li       r16,10
   0x10021714:  li       r15,0
   0x10021718:  cmpw     r15,r16
   0x1002171c:  bge      0x10021744
   0x10021720:  rotlwi   r17,r15,2
   0x10021724:  add      r17,r3,r17
   0x10021728:  rotlwi   r18,r15,2
   0x1002172c:  add      r18,r3,r18
   0x10021730:  lwz      r18,0(r18)
   0x10021734:  mulli    r18,r18,42
%    Specialized value        ----^
   0x10021738:  stw      r18,0(r17)
   0x1002173c:  addi     r15,r15,1
   0x10021740:  b        0x10021718
   0x10021744:  blr
   0x10021748:  .long 0x0
```

# Scenario #2 : Kernel Initialization


## Value injection / dynamic polymorphism

```
#[  int 32 1 mult (int[] 32 1 a,
                   int   32 1 len)  {
    int 32 1 r, i;
    for (i = 0; i < len; i = i + 1) {
        // b value will be included
        //in code generation
        a[i] = a[i] * #(b);
      }
  }  ]#
```

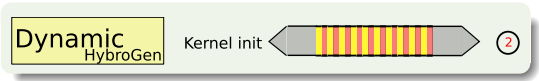
## Optimization source

- Optimization based on kernel input parameters

## POWER Binary codes specialization on 0

```
(gdb) x/20i fPtr
   0x10021710:  li       r16,10
   0x10021714:  li       r15,0
   0x10021718:  cmpw     r15,r16
   0x1002171c:  bge      0x10021744
   0x10021720:  rotlwi   r17,r15,2
   0x10021724:  add      r17,r3,r17
   0x10021728:  rotlwi   r18,r15,2
   0x1002172c:  add      r18,r3,r18
   0x10021730:  lwz      r18,0(r18)
   0x10021734:  li       r18,0
% Specialized value 0    ----^
   0x10021738:  stw      r18,0(r17)
   0x1002173c:  addi     r15,r15,1
   0x10021740:  b        0x10021718
```

# Scenario #2 : Kernel Initialization




## Value injection / dynamic polymorphism

```
#[  int 32 1 mult (int[] 32 1 a,
                   int   32 1 len)  {
    int 32 1 r, i;
    for (i = 0; i < len; i = i + 1) {
        // b value will be included
        //in code generation
        a[i] = a[i] * #(b);
      }
  }  ]#
```

## Optimization source

- Optimization based on kernel input parameters

## POWER Binary codes specialization on 512

```
(gdb) x/20i fPtr
   0x10021710:  li       r16,10
   0x10021714:  li       r15,0
   0x10021718:  cmpw     r15,r16
   0x1002171c:  bge      0x10021744
   0x10021720:  rotlwi   r17,r15,2
   0x10021724:  add      r17,r3,r17
   0x10021728:  rotlwi   r18,r15,2
   0x1002172c:  add      r18,r3,r18
   0x10021730:  lwz      r18,0(r18)
   0x10021734:  rotlwi   r18,r18,9
% Specialized value 512      ----^
% (shift left instead of mul)
   0x10021738:  stw      r18,0(r17)
   0x1002173c:  addi     r15,r15,1
   0x10021740:  b        0x10021718
   0x10021744:  blr
```

# Scenario #3 : Application Controlled


## Example Transprecision Newton square root

Declaration **flt #(FloatWidth) 1** allow to specialize variable word width (and related operators) at run time

```
#[
flt #(FloatWidth) 1 iterate(
              flt #(FloatWidth) 1 u,
              flt #(FloatWidth) 1 val,
              flt #(FloatWidth) 1 div )
{
    flt #(FloatWidth) 1 r, tmp1, tmp2;
    tmp1 = val / u;
    tmp2 = u + tmp1;
    return tmp2 / div;
} ]#
```

## Optimization source

- Dynamic transprecision
- Dynamic word size adaptation depending on programmer threshold (precision)

## Transprecision algorithm

- Generate kernel for 32 bits `float`
- Starts iteration with 32 bits `float`
- Until A threshold
- Regenerate with 64 bits `double`
- Until convergence

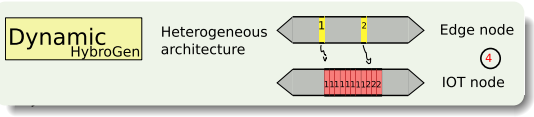
# Scenario #4 : Heterogeneous Architectures

## Image Difference

```
int 32 1 subImage(int[] 16 8 a,
  int[] 16 8 b, int[] 16 8 res, int 32 1 l
// a, b, res are array of csram lines
  {
    int 32 1 i; // RISC-V register
    // Control done on RISC-V
    for (i = 0; i < len; i = i + 1)
    { // Workload done on C-SRAM
      res[i] = a[i] - b[i];
    }
  }
  return 0;
```

## Optimization source

- Correct usage of specialized accelerator
- Code interleaving (RISCV, CSRAM)




## Computational SRAM

- RISCV for the control part
- C-SRAM for the workload

## Code generation principle

- Specialized CSRAM compiler plugin to
  - Recognize CSRAM instruction
  - Generate RISCV instructions

## Other demonstrations

- TensorFlowLite convolution adaptation for CSRAM

# HybroGen Internals

## HybroGen description

- DB for instructions characteristics (postgresql)
- Lexer / Parser (ANTLR)
- Intermediate representation (in house)
- Classic optimization (reg allocaltion, constant expression)
- Plugin for target specific platforms
- Code Generator Generator

## Running platforms

- Self hosted platform + OS
- QEMU
- Bare metal platform

## Compilation Steps

- HybroGen
  - Parse & build IR
  - Transform pass
    - Arch specific plugin
    - Constant optimizer
    - Window optimizer
    - Register allocation
  - C code generator generation
- C cross compiler (gcc)
- Execution with run time code specialization

## Resulting code

- No run-time complex IR to manage
- No run-time tools (no assembly, no VM)

# Scientific Methodology

## Rationale

- To have a binary code genarator "simple" but powerfull enough to make experiments on compilation scenarios
- Code generation toolbox for
  - innovative compilation scenarios (4 preceding examples)
  - Code generation for "in house" CEA processor accelerator
  - Improve programmability of silicon based accelerators

## Metrics

- Code efficiency : execution time and energy : real or based on model
- Speed of code generation time : few cycle per generated instruction
- Code generator size ($< Kb$)

# Conclusion / Future Works

## Main message :

- Static compiler are
  - complex to evolve
  - tied with input programming language & classic computer architecture
- Dynamic binary code adaptation is the way to go
- Let's prepare new tools & new scenarios

## Release

- Two release per year (when ready)
- V4.0 done or soon
- Improve public demonstrators
- Open source licence (CECILL-B i.e. French BSD like)
- Stay as small as possible ($\simeq$11KLOC of python)

## Future works

- Working on programmable accelerators
- Generalize plugin support. E.G
  - Support for CEA VRP (variable precision accelerator) : Generalize the transprecision support
  - Support code generation for future CSRAM instruction set
- New code specialization scenarios : trigonometric math function, BLAS, ...
- Application level optimizations

## Access / Contact

- HybroGen on GitHub
- Mail HP Charles