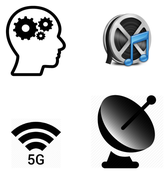
Designing Domain-Specific Heterogeneous Manycores from Dataflow Programs

Süleyman Savas - School of Information Technology, Halmstad University, Sweden

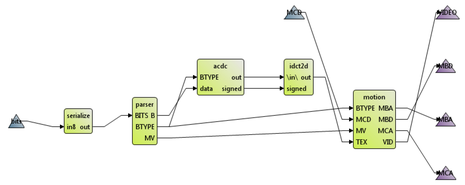
Motivation/Problem



- Machine Learning
- Audio/Video Processing
- Wireless Communication
- Radar Signal Processing

Common properties:
- Massive data stream
- Continuous processing
- Chain of tasks
- Communication

Main requirements:
- High performance
- Low power



Typical structure of an application

- Heterogeneous structure

Solution

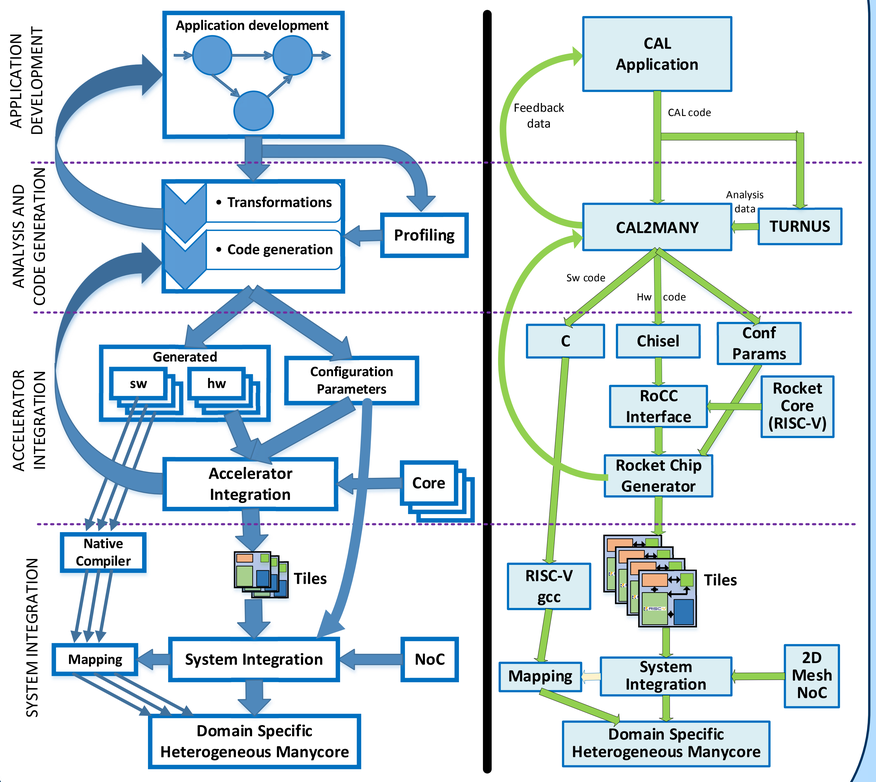
Manycore architectures with cores specialized on certain tasks through instruction extension.

- Efficient for certain domain
- Can perform general purpose processing
- How to design?

Summary of the Design Flow

- Developing the application in a language with support for parallelism
  - CAL actor language
- Analyzing the application to identify the hot-spots
  - TURNUS framework (Number of operations, execution count etc..)
- Generating hardware/software co-design
  - Hardware for the hot-spot
  - Software code for the rest of the application
  - Cal2Many framework (C + Chisel)
- Integrating the custom hardware to simple core and creating a tile
  - The software code on the simple core
  - Hot-spot on the custom hardware
  - Rocket core with RoCC interface
- Connecting several tiles with a network-on-chip
  - 2D mesh NoC

Design Flow



a) Generic design flow

b) Realization of the design flow

Results

QR decomposition
- 4x performance increase with accelerator
- With automatic code generation
  - 4% performance loss
  - 10-15 % increase in LUT and FF usage in FPGA

Autofocus criterion calculation
- 3x performance increase with accelerator
- With automatic code generation
  - No performance loss
  - No increase in FPGA resource usage

Conclusions

- Specialized cores provide higher performance
- Automation facilitates design of architectures and consequently exploration of design space

Application Development

```
__acc__calculate_boundary : action ==>
guard row_counter < COL_SIZE
var
    float a,
    float b,
    float r_tmp,
    int index
do
    index := row_counter * ROW_SIZE + col_counter;
    a     := r[index] * r[index];
    b     := x_in[col_counter] * x_in[col_counter];
    SquareRoot(a + b);
    r_tmp     := SquareRoot_ret;
    c         := r[index] / r_tmp;
    s         := x_in[col_counter] / r_tmp;
    r[index] := r_tmp;

    col_counter := col_counter + 1;
end
```

Profiling

		Overall		
Actor	Action	Firings		Weight
instance	calculate_boundary	256	8.60%	94722.47	64.66%
instance	calculate_inner	1920	64.52%	46080.00	31.46%
instance	read_x_in	256	8.60%	2560.00	1.75%
instance	read_x_in_not_done	256	8.60%	1792.00	1.22%
instance	col_done	256	8.60%	1280.00	0.87%
instance	row_done	16	0.54%	32.00	0.02%
instance	read_x_in_done	16	0.54%	16.00	0.01%

Code Generation

```
__acc__calculate_boundary : action ==>
guard row_counter < COL_SIZE
var
    float a,
    float b,
    float r_tmp,
    int index
do
    index := row_counter * ROW_SIZE + col_counter;
    a     := r[index] * r[index];
    b     := x_in[col_counter] * x_in[col_counter];
    SquareRoot(a + b);
    r_tmp     := SquareRoot_ret;
    c         := r[index] / r_tmp;
    s         := x_in[col_counter] / r_tmp;
    r[index] := r_tmp;

    col_counter := col_counter + 1;
end
```

CAL

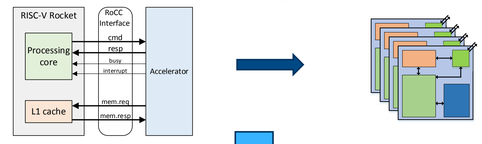
```
static void singleQRD___acc__calculate_boundary ( ) {
    .
    .
    //Copy two floating point numbers into 64 bit registers
    .
    .
    //Output registers - each will hold two outputs
    .
    .
    //Time to call the custom instructions
    ROCC_INSTRUCTION_NO_BLOCK(XCUSTOM_ACC, outputReg0, acc_input0,
                              acc_input1, FUNCT_IN1 );
    ROCC_INSTRUCTION(XCUSTOM_ACC, outputReg1, acc_input2, 0, FUNCT_FIRE );
    .
    .
    //Save the results
    .
    .
    //Set the outputs
}
```

C + custom instruction macros

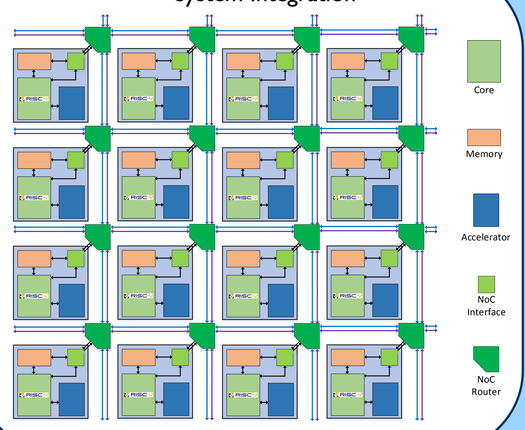
```
val multiplier0 = Module(new FPMult(32))
a_v1 := multiplier0.io.out
multiplier0.io.in1 := r_v0
multiplier0.io.in2 := r_v0

val multiplier1 = Module(new FPMult(32))
b_v1 := multiplier1.io.out
multiplier1.io.in1 := x_in_v0
multiplier1.io.in2 := x_in_v0

val sqrt0 = Module(new fpSqrt())

val adder0 = Module(new FPAdd(32))
sqrt0.io.in := adder0.io.out
adder0.io.in1 := a_v1
adder0.io.in2 := b_v1
val SquareRoot_ret_v0 = sqrt0.io.out
r_tmp_v1 := SquareRoot_ret_v0
```

Chisel

Accelerator Integration



System Integration



hh.se

suleyman.savas@hh.se

HALMSTAD UNIVERSITY