LLVM-CHiMPS

Compilation Environment for FPGAs Using LLVM Compiler Infrastructure and CHiMPS Computational Model

Seung J. Lee[1], **David K. Raila**[2] and **Volodymyr V. Kindratenko**[1]

[1]Innovative Systems Lab., National Center for Supercomputing Applications
[2]Institute of Government and Public Affairs, University of Illinois at Urbana-Champaign

Reconfigurable Systems Summer Institute
July 8, 2008

CHiMPS

Compiling High level language to Massively Pipelined System

- **A computational model and architecture for FPGA computing by Xilinx, Inc.**

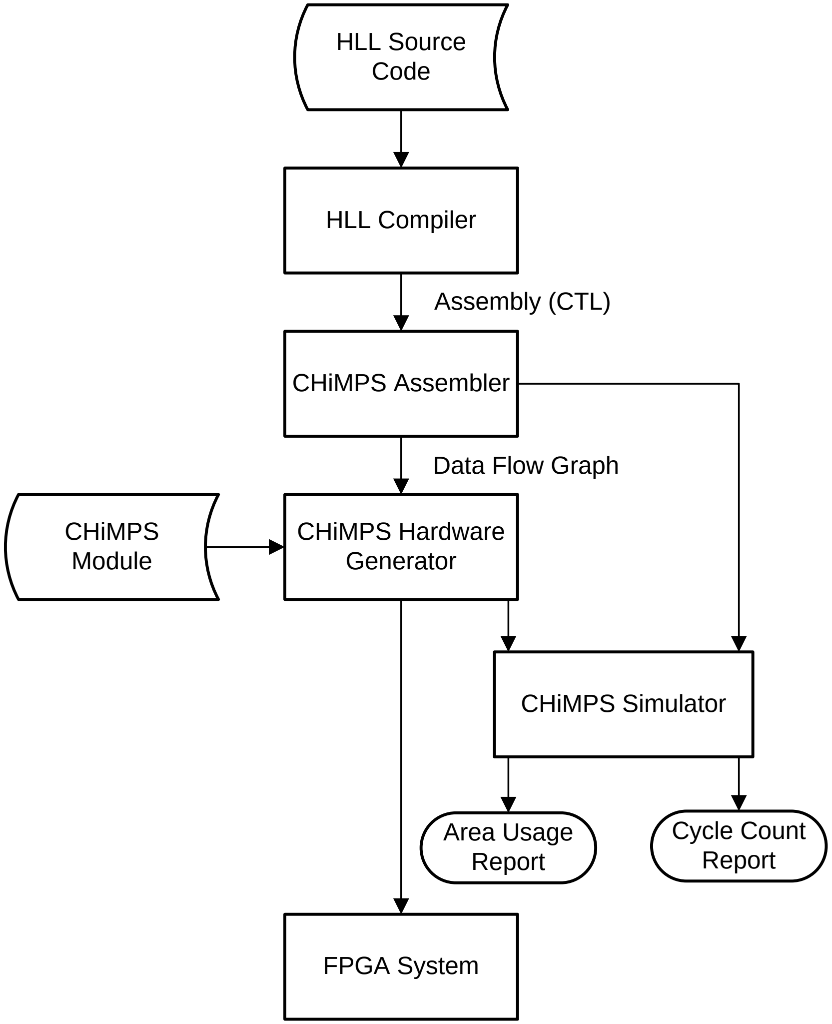
  - Standard software development model (ANSI C)
  Trade performance for convenience

  - Virtualized hardware architecture
  CHiMPS Target Language (CTL) instructions

  - Cycle accurate simulator

  - Runs on BEE2

CHiMPS compilation flow



Some known limitations in CHiMPS

- **Code restrictions**

```
char* foo (int select, char* brTid, char* brFid) {
        if (select)
                return brTid;
        return brFid;
}
```

A simple code fails in Xilinx CHiMPS compilation

Some known limitations in CHiMPS

- **Code restrictions**

```
for (i=0; i<n; i++) { }
```

```
for (i=0; i<=n; i++  )  { }
for (i=0; i<n  ; i+=2)  { }
for (i=1; i<n  ; i++  )  { }
```

(a) Supported and (b) Unsupported expressions of *for* loop

Some known limitations in CHiMPS

- **Poor optimization**

```
int foo() {
        int n;
        int k=0;
        for (n=0; n<10; n++)
                k+=n;
        return k;
}
```

```
Enter foo;
reg k, n
add 0;k
reg temp0:1
nfor l0;10;n
        add k,n;k
end l0
exit foo; k
```

(a) Source code and (b) CTL from Xilinx CHiMPS

LLVM

Low Level Virtual Machine

- **An open source compiler infrastructure developed in the University of Illinois (llvm.org)**

- Compilation strategy

Supports powerful global optimizations

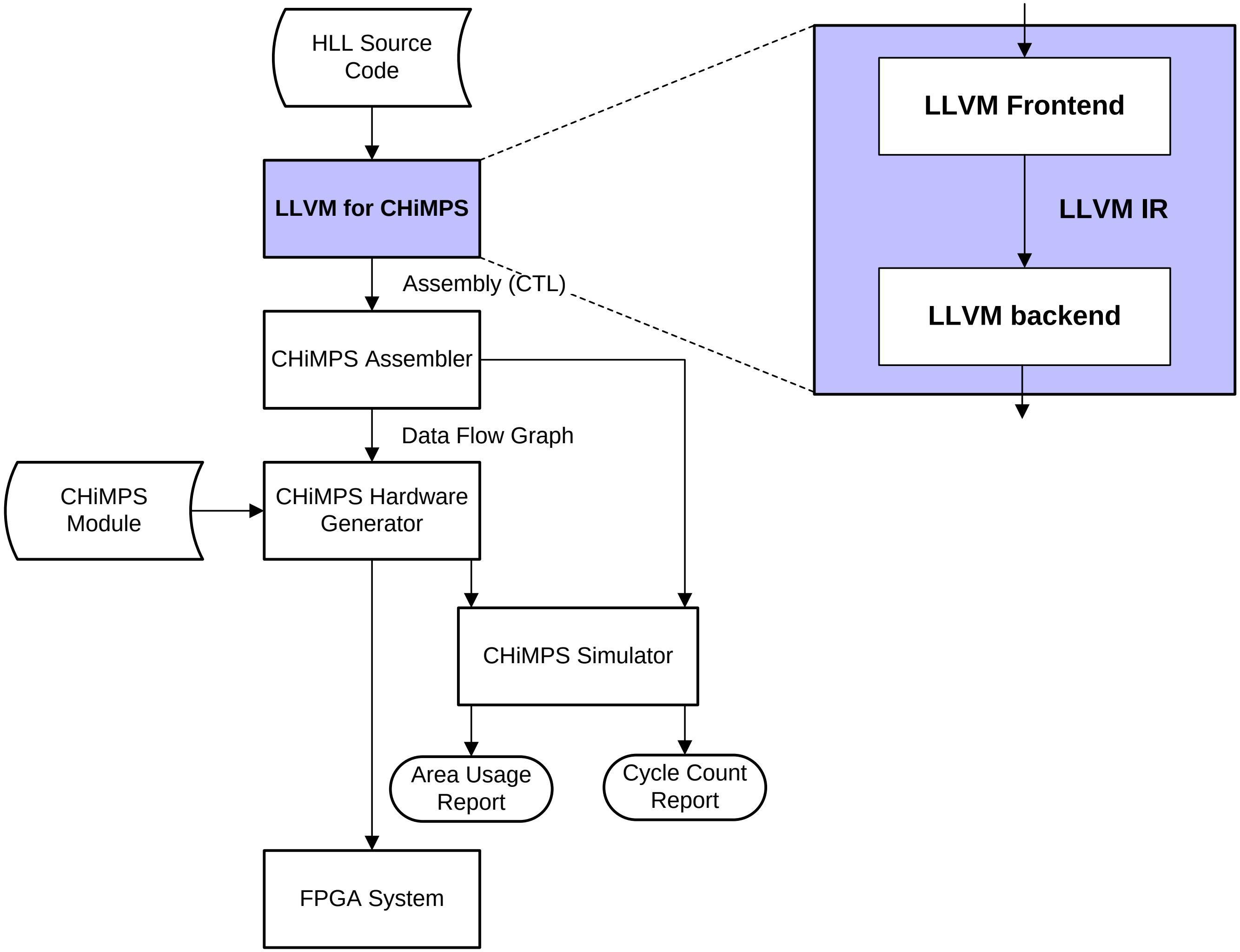
- Virtual Instruction set

Low-level intermediate representation (IR)
Static Single Assignment (SSA) form

- Compiler infrastructure

GCC-based C & C++ front-end and Clang
Modular & reusable components for building compilers
Static backends for most major architectures

LLVM in CHiMPS compilation flow



LLVM IR vs. CTL

- **LLVM IR**

  - Every LLVM backend for a specific target is based on

  - Low level SSA form virtual instruction set using simple RISC-like target independent instructions (atoms)

- **CTL**

  - Close resemblance to a traditional microprocessor instruction set

  - Defines low level and some high level instructions (atoms and some molecules)

Implementation of low level representations

- **Arithmetic and logical instructions**

Integer arithmetic
- *add, sub, multiply, divide, cmp*

Floating-point arithmetic
- *i2f, f2i, fadd, fsub, fmultiply, fdivide, fcmp*

Logical operations
- *and, or, xor, not, shl, shr*

Binary operations
- *add, sub, mul, udiv, sdiv, fdiv, urem, srem, frem*

Bitwise binary operations
- *shl, lshr, ashr, and, or, xor*

Other operations
- *icmp, fcmp, …*

Conversion operations
- *sitofp, fptosi, …*

(a) CTL and (b) LLVM IR

Implementation of low level representations

- **Memory access instructions**

  - memread, memwrite *- load, store, …*

  (a) CTL and (b) LLVM IR

- **CHiMPS pseudo instructions**

  - reg, enter, exit, assign, foreign, call

Code example

- **Source code**

```
void testmt(long s, double* a) {
        char h = mtrandinit(s);
        *a = s;
        *a *= mtrandint31(h);
        *a += mtrandint32(h);
        *a += mtrandreal1(h);
        *a += (mtrandreal1(h) / mtrandreal2(h));
        *a -= (*a * mtrandreal3(h) * mtrandres53(h));
}
```

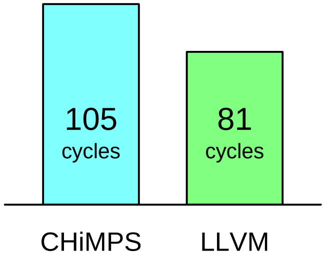
A fragment of Mersenne Twister code

Code example

- **CTL from CHiMPS and LLVM**

```
Enter testmt; s,a
reg h:8
reg temp0:64, temp1, temp2, temp3:64, temp4:64, temp5:64, temp6:u, temp7,
 temp8:64, temp9:64, temp10:64, temp11:u, temp12:64, temp13:64,
 temp14:64, temp15:64, temp16, temp17:64, temp18:64, temp19:64,
 temp20:64, temp21, temp22:64, temp23:64, temp24:64, temp25:64,
 temp26:64, temp27, temp28:64, temp29:64, temp30:64, temp31:64
call mtrandinit;s;h
i2f s;temp0
write a;temp0;8;;
call mtrandint31;h;temp1
add a;temp2
read 0;;8; temp2;;temp3
i2f temp1;temp4
fmultiply temp3,temp4;temp5
write temp2;temp5;8;;
call mtrandint32;h;temp6
add a;temp7
read 0;;8; temp7;;temp8
i2f temp6>>1;temp9
fmultiply 2.0,temp9;temp10
and temp6,1;temp11
i2f temp11;temp12
fadd temp10,temp12;temp13
fadd temp8,temp13;temp14
write temp7;temp14;8;;
call mtrandreal1;h;temp15
add a;temp16
read 0;;8; temp16;;temp17
fadd temp17,temp15;temp18
write temp16;temp18;8;;
call mtrandreal1;h;temp19
call mtrandreal2;h;temp20
add a;temp21
read 0;;8; temp21;;temp22
fdivide temp19,temp20;temp23
fadd temp22,temp23;temp24
write temp21;temp24;8;;
call mtrandreal3;h;temp25
call mtrandres53;h;temp26
add a;temp27
read 0;;8; temp27;;temp28
fmultiply temp28,temp25;temp29
fmultiply temp29,temp26;temp30
fsub temp28,temp30;temp31
write temp27;temp31;8;;
exit testmt
```

```
Enter testmt; s,a
reg tmp:32u, tmp_1:8, tmp1:64, tmp5, tmp5_1:64, tmp6:64, tmp12:32u,
tmp12_1:64, tmp13:64, tmp19:64, tmp20:64, tmp26:64, tmp29:64,
tmp30:64, tmp31:64, tmp39:64, tmp40:64, tmp43:64, tmp44:64, tmp45:64
add s;tmp
call mtrandinit; tmp;tmp_1
i2f s;tmp1
write a; tmp1;8;;
call mtrandint31; tmp_1;tmp5
i2f tmp5;tmp5_1
fmultiply tmp1, tmp5_1;tmp6
write a; tmp6;8;;
call mtrandint32; tmp_1;tmp12
i2f tmp12;tmp12_1
fadd tmp6, tmp12_1;tmp13
write a; tmp13;8;;
call mtrandreal1; tmp_1;tmp19
fadd tmp13, tmp19;tmp20
write a; tmp20;8;;
call mtrandreal1; tmp_1;tmp26
call mtrandreal2; tmp_1;tmp29
fdivide tmp26, tmp29;tmp30
fadd tmp20, tmp30;tmp31
write a; tmp31;8;;
call mtrandreal3; tmp_1;tmp39
fmultiply tmp31, tmp39;tmp40
call mtrandres53; tmp_1;tmp43
fmultiply tmp40, tmp43;tmp44
fsub tmp31, tmp44;tmp45
write a; tmp45;8;;
exit testmt
```



Implementation of high level representations

- **Instructions for conditional jump**

- demux, branch, unbranch, mux, switchass

- br, switch, select

(a) CTL and (b) LLVM IR

- **Conditional jumps in CHiMPS and LLVM**

```
int foo(int k, int j) {
    if (j < 300)
        k = 200 + j;
    return k;
}
```

```
Enter foo; k,j
reg temp0:1
cmp j,300;;temp0
demux m0;temp0;b1;b0
branch b0
    add j,200;k
unbranch b0
branch b1
unbranch b1
mux m0
exit foo; k
```

```
define i32 @foo(i32 %k, i32 %j) nounwind {
entry:
%tmp2 = icmp slt i32 %j, 300
br i1 %tmp2, label %bb, label %Return

bb:     ; preds = %entry
%tmp5 = add i32 %j, 200
ret i32 %tmp5

Return:
ret i32 %k
}
```

(a) Source code (b) CTL and (c) LLVM IR

Implementation of high level representations

- **Looping instructions**

- CTL defines *for* and *nfor* to support loops

High level instructions similar to *for* statement in C

- No explicit instructions for looping in LLVM IR

Loops represented by control flows among basic blocks

- Needs to detect loops in LLVM IR and dress them in high level to construct the high level loops in CTL

Not sequential because of back paths unlike Conditionals

- Possible to have 'improper regions' in LLVM IR

which makes the translation from LL to HL complex

Implementation of high level representations

- **Looping instructions**

  - CHiMPS does not support some statements:
  goto, *break*, *continue*, *switch/case* …
  which makes LLVM IR much simpler

  - Assumption of the reducible control flow graph
  Each retreating edge shown in the flow graph can be associated with a natural loop

Implementation of high level representations

- **Loops detection**

- Control Dependence Graph is helpful to detect loops

- Five steps to derive CDG from CFG (by Cytron et al.)

Control Flow Graph (CFG)
Reversed CFG (RCFG)
Dominator Tree for RCFG
Dominance Frontier for RCFG
Control Dependence Graph (CDG)

- LLVM supports a separate natural loop analysis pass

Implementation of high level representations

- **Destruction of SSA form for HL loops**

  - SSA is used for the efficient data flow representation and code analysis in LLVM IR

  Phi-nodes need to be replaced with proper copies

  - Briggs et al. proposed an approach to destruct the phi-nodes without lost-copy and swap problems

  - LLVM supports DemotePHI() to destruct phi-nodes since version 2.1

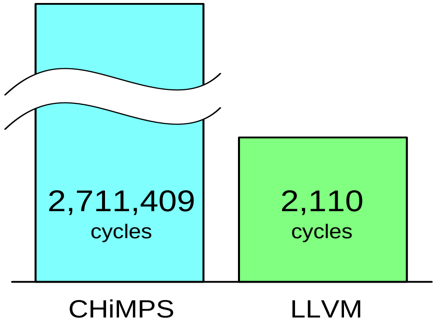
Code example

- **Source code and CTL from CHiMPS and LLVM**

```
int foo() {
  int i, j, k, n;
  int sum = 0;
  for (i=0; i<10; i++) {
    sum += i;
    for (j=0; j<10; j++) {
      sum += 2;
      for (k=0; k<10; k++) {
        sum += k;
        for (n=0; n<10; n++)
          sum += 3;
      }
    }
  }
  return sum;
}
```

```
Enter foo;
reg i, sum
add 0;sum
reg temp3:1
nfor l3;10;i
  reg j
  add sum,i;sum
  reg temp2:1
  nfor l2;10;j
    reg k
    add sum,2;sum
    reg temp1:1
    nfor l1;10;k
      reg n
      add sum,k;sum
      reg temp0:1
      nfor l0;10;n
        add sum,3;sum
      end l0
    end l1
  end l2
end l3
exit foo; sum
```

```
Enter foo;
reg phi_indvar9,phi_sum, indvar_next,
indvar9_rl, phi_sum_rl, tmp4, tmp5
add 0;phi_indvar9
add 0;phi_sum
nfor l0;10;indvar_next
  add phi_sum;phi_sum_rl
  add phi_indvar9;indvar9_rl
  add indvar9_rl, 3470;tmp4
  add phi_sum_rl, tmp4;tmp5
  add indvar9_rl, 1;indvar_next
  add indvar_next;phi_indvar9
  add tmp5;phi_sum
end l0
exit foo; tmp5
```



Implementation of high level representations

- **Limitations in optimization**

  - CTL code is generated at compile time

  No optimization by LLVM for a source code in which no such expressions can be optimized at compile time

  - LLVM does not have a chance to dynamically optimize the source code at run time

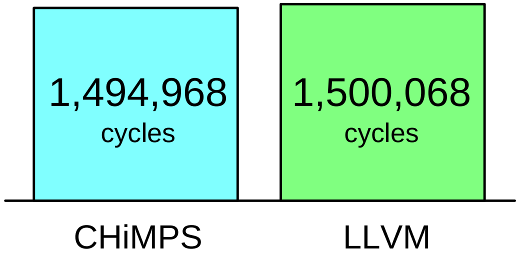
  - LLVM is not almighty

  Floating point math is still difficult to LLVM

Code example

- **Source code**

```
void matmul (long* a, long* b, long* c, long sz) {
  long i, j, k;
  for (i = 0; i < sz; i++)  {
    long offset = i * sz;
    long* row = a + offset;
    long* out = c + offset;
    for (j = 0; j < sz; j++)  {
      long* col = b + j;
      out[j] = 0;
      for (k = 0; k < sz; k++)
        out[j] += row[k] * col[k*sz];
    }
  }
}
```



(50 x 50 matrices)

Summary

- **LLVM backend for generation of CTL**
  - **Implementation of LL and HL representations**
  - **Reduced the number of cycles needed for execution**

- **Major difficulty in the implementation related to HL loops construction**
  - **Reducibility and normal loop assumptions were made**
  - **Analysis and transformation LLVM passes were used**

- **LLVM : A fast evolving open source project**
  - **Optimization functions are consistently being added**
  - **More chances to leverage optimizations and transformations for LLVM-CHiMPS in the future**

Acknowledgments

- **Innovative Systems Laboratory (ISL) at NCSA**
- **Xilinx Research Labs**
- **Prof. Vikram Adve and LLVM developers**

Thank you !!