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

Seung J. Lee[1], David K. Raila[2], Volodymyr V. Kindratenko[1]
1) National Center for Supercomputing Applications (NCSA)
2) University of Illinois at Urbana-Champaign (UIUC)
lee225@uiuc.edu, raila@illinois.edu, kindr@ncsa.uiuc.edu

Abstract

CHiMPS (Compiling High level language to Massively Pipelined System) system, developed by Xilinx is gaining popularity due to its convenient computational model and architecture for field programmable gate array computing. The CHiMPS system utilizes CHiMPS target language as an intermediate representation to bridge between the high level language and the data flow architecture generated from it. However, currently the CHiMPS frontend does not provide many commonly used optimizations and has some use restrictions. In this paper we present an alternative compiler environment based on low level virtual machine compiler environment extended to generate CHiMPS target language code for the CHiMPS architecture. Our implementation provides good support for global optimizations and analysis and overcomes many limitations of the original Xilinx CHiMPS compiler. Simulation results from codes based on this approach show to outperform those obtained with the original CHiMPS compiler.

1. Introduction

1.1. CHiMPS

Recently, systems have been developed that employ Field Programmable Gate Arrays (FPGAs) as application accelerators. In order to execute applications on such systems, the application programmer extracts computationally intensive kernel from the application, reorganizes data structures to accommodate memory subsystem, and rewrites the kernel in the highly specialized environment for FPGA execution. The CHiMPS (Compiling High level language to Massively Pipelined System) system [1], developed by Xilinx, employs a unique approach by imposing a computational model on the FPGA subsystem, treating it as a virtualized hardware architecture that is convenient for high level code compilation. The CHiMPS compiler transforms high level language (HLL) codes, currently written in ANSI C, to CHiMPS target language (CTL), which is combined with runtime support for the FPGA runtime environment and assembled using the CHiMPS assembler and FPGA design tools to a hardware data flow implementation [2]. The instructions defined in CTL have a close resemblance to a traditional microprocessor instruction set which is convenient for compiling, optimizing, and should be familiar to programmers [3, 4]. This effectively bridges the gap between hardware design and traditional software development and only imposes minor costs related to runtime overhead support to virtualize the architecture.

1.2. LLVM

Low level virtual machine (LLVM) compiler [5], originally developed at the University of Illinois, is now an open source compiler project that is aimed at supporting global optimizations. It has many attractive features for program analysis and optimization. LLVM has a GCC based C and C++ frontend as well as its own frontend, Clang, that offers modular and reusable components for building compilers that are target-independent and easy to use. This reduces the time and effort to build a particular compiler. Static backends already implemented in LLVM including X86, X86-64, PowerPC 32/64, ARM and others share those components. A notable feature of LLVM is its low level static single assignment (SSA) form virtual instruction set which is a low-level intermediate representation (IR) that uses RISC-like instructions. The IR makes LLVM versatile and flexible as well as language-independent and is amenable data flow representations.

1.3. LLVM in CHiMPS compilation flow

The current LCC based frontend of CHiMPS compiler [1] is known to have some serious drawbacks as shown in Section 2. In the work presented in this paper, LLVM is employed to improve these shortcomings and to investigate the ability to do high level program transformations that better support the CHiMPS architecture. Figure 1 illustrates the compilation flows of the original Xilinx CHiMPS. The LLVM backend part is aimed at substituting for CHiMPS HLL compiler. In the following sections, the implementation details are discussed.

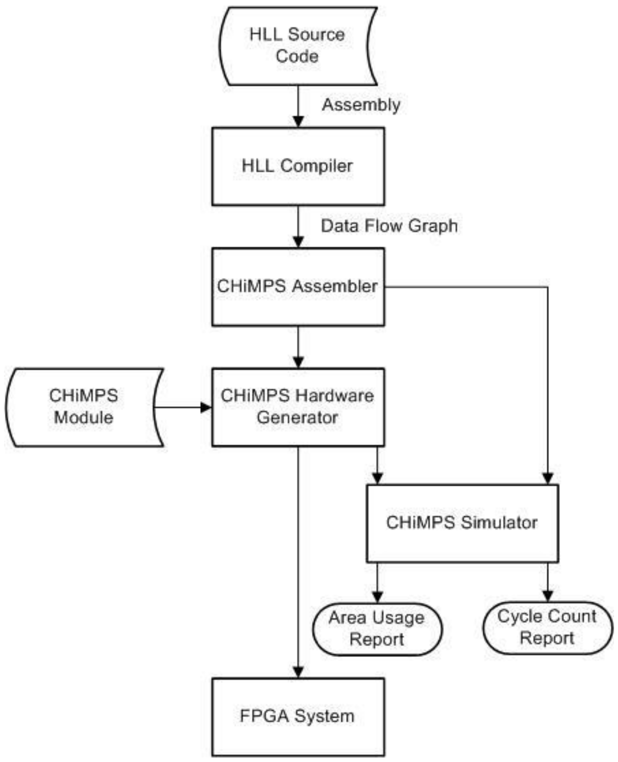

Figure 1. Compilation flow of Xilinx CHiMPS compiler.

1.4. Related work

Several research and commercial HLL to FPGA compilers are currently available [19-27]. Although their implementations differ significantly, the general framework of compilation strategies is the same: high level source code is translated into a low-level hardware language, such as VHDL or Verilog.

The idea of the frontend optimization strategy using LLVM has been proven by the Trident compiler [19]. Trident's framework is similar to LLVM-CHiMPS compilation except that Trident compiler is used instead of CHiMPS backend. Also in our LLVM-CHiMPS compiler a more aggressive set of optimization techniques is implemented in the LLVM frontend whereas in the Trident's approach many optimizations are pushed down to the Trident's backend.

ROCCC compiler [20] is another example of a compiler that translates HLL code into highly parallel and optimized circuits. Instead of using LLVM frontend and CHiMPS backend, the ROCCC compiler is built on the SUIF2 and Machine-SUIF [29, 30] platforms, with the SUIF2 platform being used to process optimizations while the Machine-SUIF platform is used to further specialize for the hardware and generate VHDL.

On the commercial side, several HLL compilers are available including Mitrion-C [26], Impulse-C [25], DIME-C [27], and MAP-C [28]. These compilers, however, are proprietary and support only a few reconfigurable computing systems.

In our LLVM-CHiMPS approach, we take advantage of the optimization techniques implemented in the open-source LLVM compiler and of the FPGA hardware knowledge applied to the backend CHiMPS compiler developed by Xilinx.

2. Examples of known limitations in Xilinx CHiMPS compiler

Original Xilinx CHiMPS compiler has some limitations that make it inconvenient and difficult for programming. For example, legal high level expressions in HLL code sometimes need to be paraphrased for CHiMPS compilation. Consider the ANSI C code shown in Figure 2.

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

Figure 2. Simple code that fails in Xilinx CHiMPS compiler

```
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++)  { }
```

Figure 3. (a) Supported and (b) unsupported expressions of for loop construct

Source code

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

  return k;
}
```

CTL from Xilinx CHiMPS compiler

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

LLVM IR

```
define i32 @foo() nounwind  {
entry:
      ret i32 45
}
```

Figure 4. XIlinx CHiMPS compiler does not employ common optimization techniques.

This simple code fails because two *return* statements are used in a single function, and although this is legal in ANSI C, the Xilinx CHiMPS frontend does not support multiple returns.

Figure 3a shows the only form of *for* statement that is fully supported by the Xilinx CHiMPS compiler. All other uses of *for* statement, such as shown in Figure 3b are not guaranteed to produce correct code [16].

A very significant weakness of the Xilinx CHiMPS compiler frontend is that no optimizations are carried out during compilation [17]. Figure 4 depicts an example for a simple summation from 0 to 9 that yields 45, and is not optimized at all by the CHiMPS frontend. The CTL code emitted is almost directly related to the source code, although this kind of a simple high level code is commonly optimized at the compilation time, as shown in the LLVM IR, the right column in Figure 4.

These serious drawbacks in CHiMPS can be improved using LLVM as a frontend which offers more stable compilation with numerous global optimizations and the potential for significant program transformations.

3. From LLVM to CHiMPS

As shown in Figure 1, our approach is to use LLVM as the frontend to generate CTL code instead of the Xilinx CHiMPS frontend. This section describes how LLVM is modified to meet this goal. LLVM has its own virtual instruction set which is a low-level IR that uses RISC-like target independent instructions. Every LLVM backend for a specific target is based on this LLVM IR.

In the following sections, overall comparison of instructions at two different levels is exposed. While LLVM instructions are low level, CHiMPS target language defines low level, as well as some high level instructions such as *for* and *nfor*, which are similar to *for* statement in C language. The difference between these two levels is critical in this implementation of LLVM-CHiMPS, and so it needs to be discussed in more detail. Readers are directed to refer to the LLVM and CHiMPS documentation [2, 3, 5] for more detailed syntax, semantics and examples of the instructions for CHiMPS and LLVM.

3.1. Implementation of low level representations in CHiMPS

LLVM is intended to support global optimization which is one of the motivations for using it in this work. Additionally, LLVM’s simple RISC-like instructions are quite similar to CHiMPS instructions which are also similar to traditional microprocessor instructions, so there is a good connection between these models and a good starting point for LLVM-CHiMPS.

- ▫ 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, …*

Figure 5. Arithmetic and logical instructions in (a) CHiMPS and (b) LLVM

- ▫ Pseudo-instructions

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

Figure 6. CHiMPS pseudo-instructions

- ▫ Standard memory access instructions

  - *memread, memwrite*

- ▫ Memory access and addressing operations

  - *load, store, …*

Figure 7. Memory access instructions in (a) CHiMPS and (b) LLVM

CHiMPS also has common instructions for arithmetic and logical operations. Figure 5 enumerates those instructions and the counterparts in the current LLVM IR (version 2.2). As shown in Figure 5, there is a great deal of similarity between the LLVM IR and CTL instructions, although classification of them is slightly different. Thus, LLVM IR can be readily translated into CHiMPS counterparts.

Pseudo-instructions in CHiMPS shown in Figure 6 are easily handled because such instructions as *reg*, *enter* and *exit* are nothing more than declaration of registers and notification of start and end of a function. Since version 2.0 of LLVM, integers have had signless types such as *i8*, *i16* and *i32* instead of *ubyte*, *sbyte*, *short* and so on. This bit width information can be used to describe registers in CTL whose width is not the default 32 bits. When copying the registers to a new size, specification of a bit width may be necessary, where it is useful. The other instructions for function call are easily associated with *call* instruction in LLVM IR. The memory access instructions also show close resemblance as shown in Figure 7.

The LLVM backend that we implemented benefits from the similarities between the two intermediate languages for emission of CTL. Example source code on the left in Figure 8 is a fragment of Mersenne Twister code by Makoto et al. [18]. Figure 8 shows the resulting code from Xilinx CHiMPS (middle column) and our LLVM-CHiMPS (right column) compilers. Comparison of the number of cycles counted from the CHiMPS simulator for each of CTL from LLVM and CHiMPS shows promising results: 81 cycles are spent for CTL from LLVM-CHiMPS compared to 105 cycles for Xilinx CHiMPS, which demonstrates the effectiveness of our LLVM optimizations. As shown in the figure, a good optimization is done for the high level 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));
}
```

Source code

```
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
```

CTL from Xilinx CHiMPS

```
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
```

CTL from LLVM

Figure 8. CTL from CHiMPS and LLVM

▫ Conditionals	▫ Branching instructions
- *demux, branch, unbranch, mux, switchass*	- *br, switch, select, …*

Figure 9. Instructions for conditional jump in (a) Xilinx CHiMPS and (b) LLVM

Source code

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

CTL

```
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
```

LLVM IR

```
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
}
```

Figure 10. Conditional jumps in CHiMPS and LLVM

3.2. Implementation of high level representations in CHiMPS

CTL also defines higher level control flow instructions than those shown in the previous section. These constructs support conditionals and looping instructions. Conditional instructions can be represented in a relatively easy way with the current LLVM IR grammar but the others require more support.

3.2.1. Conditional instructions

Branch and conditionals such as *if* in C must behave sequentially. In this case, instructions for conditional jump are necessary as shown in Figure 9 [2]. Implementation of these instructions in the LLVM backend for CHiMPS is slightly different from those in the previous section. In LLVM, the high level source code is broken down during compilation into atomic units called basic blocks that are used for global optimization purposes. Therefore, the conditional jump is represented as the control flow among basic blocks in LLVM IR.

Consider a simple code that does summation based on a condition shown in Figure 10. A CTL representation shown in the middle is similar to the high level source code, whereas the LLVM code shown on the right is fragmented into three basic blocks. At the *entry* block, the ensuing control flow is determined to fall into the *bb* block only when the condition is satisfied at the line of *br i1 %tmp2, label %bb, label %Return* line where the *br* instruction is used to make bridges among basic blocks. This is because of the characteristic of LLVM which breaks the source code for optimization purpose during compilation, as mentioned above.

Careful observations after several compilations of HLL source codes using LLVM reveal that comparison instructions such as *icmp* and branch instructions such as *br* are kept together in a basic block when a conditional instruction is used in the source code, as shown on the right in Figure 10. Because of this consistency, LLVM-CHiMPS can translate this code into the conditional representation in CTL even though CTL does not have the notion of a basic block. Each branched basic block in LLVM IR is considered a single branch in CTL. A branch location identified by a line that begins with a *br* instruction can also be simply converted into a branching line starts with *demux* in CTL.

```
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;
}
```

Source code

```
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
```

CTL from *Xilinx* CHiMPS

```
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
```

CTL from LLVM

Figure 11. CTL from Xilinx CHiMPS and LLVM-CHiMPS

LLVM IR also offers a simple conditional instruction, *select*, which is intended for selection of a value based on a condition without branching. This instruction has an analogue in CTL, *switchass*, so it is easily translated.

3.2.2. Looping instructions

The major issue in implementing the LLVM backend for CTL generation is related to high level looping in CTL. CTL defines high level instructions such as *for* and *nfor* to support loops which are similar to *for* statement in C, while there are no explicit instructions for looping in LLVM because LLVM is a low-level representation with loops represented by the control flow among basic blocks like the conditional jump case in Section 3.2.1. For this reason, in order to construct the high level loops in CTL, it is necessary to detect loops in LLVM IR and reorganize to dress them in HL. However, the way loops are handled is quite different from conditional jump because back path is employed to bring the control flow back to a loop entry point. Furthermore, certain patterns in the control flow may yield unstructuredness in a loop, which is also known as ‘improper regions’. This unstructuredness is usually caused by multiple entry strongly connected components. In this case, jumps into the middle of loops are found so the loop header does not dominate some of nodes in the loop [9]. Therefore, these improper regions in a loop need to be dealt with cautiously. Although unstructuredness in a loop is not frequent, it is sometimes found so the translation from a low level to a high level may be complex. There have been a number of efforts to construct or restructure effective loops from low level analysis with Control Flow Graph (CFG) and others [6-14].

However, most computational kernels that LLVM-CHiMPS is intended for do not require statements such as *goto*, *break*, *continue*, or *switch/case*, and are not supported by CHiMPS as of release Alpha 0.12 [15]. It seems reasonable to consider *for* loops, which leaves the control flow always reducible. If we expect only reducible flow graphs, each retreating edge shown in the flow graph can be associated with a natural loop because all retreating edges are back edges [6]. A natural loop is defined to have a single entry node which is the loop header. In order to detect these kinds of loops, adopting the notion of Control Dependence Graph (CDG) is quite helpful. Cytron et al. [14] used five steps to derive CDG from CFG by employing the concept of reversed CFG, dominator tree and dominance frontier. Although this derivation of CDG is not hard, it was already implemented in LLVM as a separate natural loop analysis pass.

Another issue for reconstruction of loops is related to the fact that LLVM uses an SSA based representation. SSA is gaining popularity because of its efficiency in representing data flow in a code, which also expedites analysis and optimization of a program [13, 14]. However, SSA is just an intermediate representation for compilation purpose so the phi-nodes need to be replaced with properly copied instructions for the construction of high level loops. Although Cytron et al. [14] considered replacing phi-nodes with reasonably placed copy instructions, the implementation suffered from lost-copy and swap problems [13]. For these reasons, Briggs et al. [13] suggests an alternative approach to properly destruct the phi-functions. This behavior was implemented in LLVM as a single function which is DemotePHI() since version 2.1.

As introduced above, loops are also represented by the control flow among basic blocks in LLVM IR. Therefore, a comparison instruction such as *icmp* and branch instructions such as *br* are also observed together in a basic block such as conditional jump. It may seem to be difficult to identify if a basic block is related to loops or one-way conditional jump. However, it is easily detectable with a loop analysis pass which informs us which basic block is associated with which loop.

In this study, the LLVM analysis and transformation passes introduced above are used to construct the high level looping representation in CTL. Using the LLVM passes, it is possible to translate high-level CTL from LLVM IR also with optimizations.

As an example, consider a code sample given in Figure 11 that yields 34745 as the result. CTL from LLVM-CHiMPS requires 2,110 cycles to execute in simulator and that from Xilinx CHiMPS requires 2,711,409 cycles which is more than 1,000 times as many. All of the inner loops in the source code are highly optimized by LLVM, thus resulting in better overall performance.

However, there is limit to optimization using LLVM for CTL because CTL is initially intended to be generated at compile time by CHiMPS so LLVM does not have a chance to dynamically optimize the source code at run time. The optimization shown in Figure 11 is constant expression evaluation also known as constant folding, so evaluated constant expressions at compile time are simply replaced. On the other hand, Figure 12 shows a source code for matrix multiplication in which no such expressions can be easily optimized at compile time. Accordingly, CTL generated from LLVM is not optimized as well. For the multiplication of two 50-by-50 matrices, CTL from LLVM uses 1,500,068 cycles in the simulator while that from Xilinx CHiMPS requires only 1,494,968 cycles. Now that LLVM IR uses SSA based representation, it needs some copy instructions in the revised codes for CTL, while demoting the phi-nodes, which made the number of simulation cycles a little bit higher than that from Xilinx CHiMPS. Nonetheless it is evident that CTL from LLVM has more chances to have improved performance through optimizations at compile time, so this approach is still promising.

```
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];
      }
   }
}
```

Figure 12. Source code for matrix multiplication

4. Conclusions and future work

We have described our implementation of an LLVM backend for generation of CTL. We also have shown a few examples where global optimizations performed by LLVM during compilation time greatly reduced the number of cycles needed to execute the loop. The major difficulty in this implementation was related to a few instructions, such as *for*, with high level characteristics in CTL. For such instructions, one-to-one correspondence between CTL and LLVM IR could not be found and therefore analysis and transformation LLVM passes were used based on reducibility and normal loop assumptions. LLVM is a fast evolving open source project contributed by many developers. For this reason, many optimization functions are consistently being added and there appear more chances to leverage optimizations and transformations for LLVM-CHiMPS in the future.

The method discussed in this paper is based on LLVM IR. Every instruction in CTL is translated from the current LLVM IR grammar. This means that we need to consider a new translation whenever the LLVM IR grammar changes, which is an ongoing process. When LLVM 2.0 was released, many changes were introduced such as representation of data types. LLVM also has backends for various target machines so these backends generate optimized assembly codes from LLVM IR. A de-compilation tool [7, 8] can be implemented to emit CTL code after analyzing one of these optimized machine assembly codes. This may be advantageous because the generation of CTL can be independent of the version update of LLVM IR.

In this paper, only simulation was carried out. In our future work, we will consider application kernels on the real hardware.

5. Acknowledgments

The authors would like to thank the Innovative Systems Laboratory (ISL) at the National Center for Supercomputing Applications (NCSA), the Xilinx Research Labs and Prof. Vikram Adve at the University of Illinois for their comments and support.

6. References

[1] P. Sundararajan, D. Bennett, and J. Mason, Performance estimation for FPGA acceleration, in *Proc. Workshop on Tools and Compilers for Hardware Acceleration (TCHA)*, 2006.
[2] D. Bennett, CHiMPS Target Language (CTL) reference manual, Xilinx Research Labs, 2006.
[3] CHiMPS Tutorial, Xilinx, Inc., 2007.
[4] D. Bennett, An FPGA-oriented target language for HLL compilation, *Proc. Reconfigurable Systems Summer Institute*, 2006.
[5] The LLVM Compiler Infrastructure Project, http://llvm.org/
[6] A. Aho, M. Lam, R. Sethi, and J. Ullman, *Compilers Principles, Techniques, and Tools*, Addison Wesley, 2006, ISBN 0321486811.
[7] C. Cifuentes, A Structuring Algorithm for Decompilation, *Proc. The XIX Conferencia Latinoamericana de Informatica*, 1993, pp. 267-276.
[8] C. Cifuentes, D. Simon, and A. Fraboulet, Assembly to high-level language translation, *Proc. The International Conference on Software Maintenance*, IEEE-CS Press, 1998, pp. 228-237.
[9] S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, 1997, ISBN 1558603204
[10] P. Havlak, Nesting of reducible and irreducible loops, *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 4, 1997, pp. 557-567.
[11] J. Janssen, and H. Corporaal, Making graphs reducible with controlled node splitting, *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 6, 1997, pp. 1031-1052.
[12] F. Mueller, and D. Whalley, Avoiding unconditional jumps by code replication, *SIGPLAN Not.*, vol. 27, no. 7, 1992, pp. 322-330.
[13] P. Briggs, K. Cooper, T. Harvey, and L. Simpson, Practical improvements to the construction and destruction of static single assignment form, *Softw. Pract. Exper.*, vol. 28, no. 8, 1998, pp. 859-881.
[14] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck, Efficiently computing static single assignment form and the control dependence graph, *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, 1991, pp. 451-490.
[15] Xilinx, Inc. Release Notes of CHiMPS Tool Kit Alpha 0.12, 2007.
[16] D. Bennett, Software Architect, Xilinx Research Labs, Personal Communication
[17] J. Mason, Software Architect, Xilinx Research Labs, Personal Communication
[18] T. Nishimura, and M. Matsumoto, A C-program for MT19937, Keio University, Japan, 2002.
[19] J. Tripp, K. Peterson, C. Ahrens, J. Poznanovic, and M. Gokhale, Trident: an FPGA compiler framework for floating-point algorithms, *International Conference on Field Programmable Logic and Applications*, 2005, pp. 317-322
[20] Z. Guo, W. Najjar, and B. Buyukkurt, Efficient hardware code generation for FPGAs, *ACM Transactions on Architecture and Code Optimization*, vol. 5, no. 1, 2008 pp. 6:1-26
[21] G. Genest, R. Chamberlain, and R. Bruce, Programming an FPGA-based Super Computer Using a C-to-VHDL Compiler: DIME-C, *Second NASA/ESA Conference on Adaptive Hardware and Systems*, 2007, pp. 280-286
[22] I. Pentinmakial, FPGA C Compiler, http://fpgac.sourceforge.net/
[23] D. Gallowayet al., Transmogrifier C, http://www.eecg.toronto.edu/EECG/RESEARCH/tmcc/tmcc/

[24] M. Gokhale et al., Streams-C: Stream-Oriented C Programming for FPGAs, http://www.streams-c.lanl.gov/team.shtml
[25] Impulse C, Impulse Accelerated Technologies, Inc., http://www.impulsec.com/
[26] Mitrion-C, Mitrionics, Inc., http://www.mitrionics.com/
[27] DIME-C, Nallatech Inc., http://www.nallatech.com/[28] MAP-C, SRC Computers Inc., http://www.srccomp.com/
[29] SUIF Compiler System. http://suif.stanford.edu, 2004
[30] Machine-SUIF. http://www.eecs.harvard.edu/ hube/research/machsuif.html, 2004