LLVM コンパイラ基盤を用いたベクトル化コード生成についての検討

Consideration of Vectorized Code Generation Using LLVM Compiler Infrastructure

```
class RVInstVVMIQS<bits<5> funct5, bits<3> opv,
                   dag outs, dag ins,stringopcodestr, string argstr>
      : RVInst<outs, ins, opcodestr, argstr, [], InstFormatR> {
  bits<5> vs2;
  bits<5> vs1;
  bits<5> vd;
  bit vm;

  let Inst{31-27} = funct5;
  let Inst{26-25} = 0b10;
  let Inst{24-20} = vs2;
  let Inst{19-15} = vs1;
  let Inst{14-12} = opv;
  let Inst{11-7} = vd;
  let Opcode = 0b0001011;
}
```

図 1: 命令フォーマットの定義

1 はじめに

画像認識等のデータ並列性の高い処理の高速化には SIMD 命令による処理が効果的である．しかし，SIMD 命令による処理は同時演算数を変更する場合，同時に機械語コードを作り直す必要がある．そこで機械語コードの変更なしに同時並列演算数を変更することができる，データ並列処理のためのスケーラブルなベクトル処理機能を備えたベクトル拡張付き RISC-V を開発された [1]．

我々のベクトル拡張付き RISC-V は，ARM のベクトル拡張である ARM SVE[2] を参考に組込み向けに RISC-V[3] をベクトル拡張したものである．これにより，機械語コードが同時演算数に依存しないスケーラブルなベクトル拡張を実現したが，これに対応したコンパイラがない．

この問題に対して，既存の RISC-V コンパイラをベースとして，我々のベクトル拡張に対応した自動ベクトル化コンパイラを開発する手段を検討する．

2 LLVM コンパイラ基盤

コンパイラの開発にはコンパイラ基盤である LLVM[4] を用いる．コンパイラ基盤はコンパイラの各機能がモジュール化されており，機能の再利用が行いやすくなっている．そのため，LLVM で提供されている各種モジュールを利用することで、独自機能部分の実装に集中することができる．また，LLVM には自動ベクトル化機能が備わっており，入力ソースコードの繰り返し処理をベクトル化された LLVM IR に変換する．ベクトル化された LLVM IR では，処理対象の全データへ演算を行うためにベクトル演算を繰り返した後に余剰分のデータを逐次処理によって演算を行う．その LLVM IR からパターンマッチングにより各ターゲットへのベクトル命令への変換などが行われる．ベクトル拡張付き RISC-V は RISC-V を拡張しているため，LLVM の RISC-V コンパイラとしての機能を再利用してコンパイラの開発を行う．

なお，使用する LLVM のバージョンは 13.0.0 である．

LLVM はソースコードから中間表現である LLVM IR に変換を行うフロントエンド，LLVM IR からアセンブリコード生成を行うバックエンドからなる．本研究では独自命令生成のためにバックエンドに対して変更を加える．

LLVM のバックエンドでは LLVM におけるターゲットマシン情報記述のためのドメイン固有言語である TableGen を用いて記述される，レジスタ情報，命令フォーマットや命令のニーモニック等の定義に従って LLVM IR からアセンブリコードへの変換を行っている．

```
class VALUVVMIQS<bits<5> funct5, bits<3> opv, string opcodestr>
    : RVInstVVMIQS<funct5, opv, (outs VR:$vd),  (ins VR:$vs2, VR:$vs1),
                   opcodestr, "$vd, $vs2, $vs1"> {
}

multiclass VALU_IV_V_X_I_MIQS<string opcodestr, bits<3> opv> {
    def V : VALUVVMIQS<0b00001, opv, opcodestr # ".w">,
            Sched<[WriteVIALUV, ReadVIALUV, ReadVIALUV]>;
    [...]
    def I : VALUVI8MIQS<0b00101, opv, opcodestr # "i.w">,
            Sched<[WriteVIALUI, ReadVIALUV]>;
}

defm VADD_V : VALU_IV_V_X_I_MIQS<"vadd", 0b000>;
```

図 2: 命令の定義

```
    ⋮
vle32.v   v8, (t0)      #ベクトルロード
vle32.v   v16, (a3)     #ベクトルロード
vadd.w    v8, v16, v8   #ベクトル加算
vse32.v   v8, (a4)      #ベクトルストア
    ⋮
```

図 3: 生成されたアセンブリコード

3 LLVM における独自命令の生成機能の実装

ベクトル拡張付き RISC-V 命令の定義のために，図 1 のようなクラスを定義する．クラス RVInstVVMIQS はベクトル要素同士を演算する命令であるベクトル加算等の命令用のフォーマットとして定義する．各命令フォーマットは 32bit の命令フィールドを定義している基本クラス RVInst を継承する形で命令によって異なるフォーマットを定義する．

命令の定義は命令フォーマットのクラスを継承し，エンコードの値やニーモニックを指定して行う．命令の定義を図 2 に示す．図 2 の VALUVVMIQS クラスは命令定義の際に同じ定義の繰り返しを防ぐために定義する．同様の入出力レジスタを持つ命令はこのクラスをインスタンス化する際に命令の文字列と命令選択に用いるための 12-14 ビットの値の指定を行う．

配列加算のプログラムからアセンブリコードを生成した結果の一部を図 3 に示す．vadd.w が我々のベクトル拡張命令のベクトル加算命令である．しかし，ロード・ストア命令については RISC-V の V 拡張の命令のままである．

現在ベクトル拡張付き RISC-V のベクトル命令の内，ベクトル演算命令のプレディケートなし，即値による演算命令の実装が完了している．しかし，ロード・ストア命令等が未実装である．それらの命令ではプレディケートレジスタを用いており，新たにプレディケートレジスタの定義を必要とする．また，現在の LLVM IR はベクトル演算の繰り返しと逐次処理によるベクトル処理を行っている．そのため，我々のベクトル拡張のプレディケート付き命令を効果的に利用できない．

4 おわりに

本稿では，自動でベクトル化された独自のベクトル拡張付き RISC-V 命令アセンブリコードを得るための LLVM バックエンドの実装を行った．

今後の課題として，ベクトルロード・ストア命令等の未実装の命令生成の実現が挙げられる．現在実装済みの命令は命令の定義等で実装が可能であった．しかし，未実装の命令については現在のベクトル化された LLVM IR からの生成が困難であると考えられる．そのため，レジスタの定義に加え，LLVM IR の生成を行うフロントエンドの変更が必要である．

参考文献

[1] Yoshiki Kimura, et al.: "Proposal of Scalable Vector Instruction Set for Embedded RISC-V Processor," Proc. 2019 Seventh International Symposium on Computing and Networking Workshops (CANDARW), Vol.1, pp.435-439, 2019.

[2] Nigel Stephens, et al.: "The ARM Scalable Vector Extension," IEEE Micro, Vol.37, No.2, pp.26-39, 2017.

[3] Andrew Waterman, Krste Asanovi: "The RISC-V Instruction Set Manual Volume I: Unprivileged ISA, Document Version 2.2," 2017.

[4] Chris Lattner, Vikram Adve: "LLVM: A Compilation Frame-work for Lifelong Program Analysis Transformation," Proc. 2004 International Symposium on Code Generation and Optimization (CGO ' 04), pp.75-86, 2004.