Vol.2019-HPC-170 No.16
2019/7/25

LLVMを用いたベクトルプロセッサ向けコンパイラ

石坂 一久[1,a)]　井手口 裕太[1,b)]　大野 善之[1,c)]

概要：ベクトル長をソフトウェアから設定可能であるベクトルプロセッサ向けの LLVM の拡張について，SX-Aurora TSUBASA の Vector Eingine 向けの実装を通して説明する．このような可変長ベクトル命令のサポートの課題として，固定長ベクトル命令では起こらない，ベクトル長がベクトルレジスタの要素数よりも短い際のベクトルレジスタの部分的な更新への対応がある．そこで，我々はベクトル命令向け Intrinsics を対象として，Intrinsic 関数の引数にベクトル長と”pass through” を導入し，さらにコンパイラが自動でベクトル長設定命令を生成する方式を開発した．本方式を SX-Aurora TUSBASA 向け LLVM へ実装し，既存のコンパイラの大規模な変更を伴わず，従来の Intrinsics のセマンティクスを維持した可変長ベクトル対応が可能であることを確認した．

キーワード：ベクトルプロセッサ、SX-Aurora TSUBASA、コンパイラ、LLVM

1. はじめに

ベクトル演算は同じ演算を複数のデータに対して一度に行う演算であり，プロセッサの性能向上のため，HPC 向けプロセッサから組み込み向けのプロセッサまで幅広く利用されている．例えば，Intel の x86 プロセッサは，MMX 命令セット拡張で 64bit 幅の SIMD 命令を導入し，その後 128bit の SSE，256bit の AVX，512bit の AVX512 と強化している．また，最近では ARM の Scalable Vector Extension(SVE) [1] や RISC-V Vector Extension [2] が，いずれもまだ実チップは一般に利用可能ではないが，HPC 向けにベクトル命令を導入しようとしている．

これらの最近のベクトル命令は，クラシカルな SIMD 命令と異なり，最大ベクトル長がプロセッサの実装依存であることや，ベクトル命令が実際に演算を行うベクトル長（Active Vector Length や Explicit Vector Length と呼ばれる．本稿では単にベクトル長と呼ぶ）をソフトウェアから設定できるという特徴を持つ．本稿では後者の特徴を可変ベクトル長と呼ぶこととする．

このように，従来の SIMD 命令とは異なる特徴を持つため，コンパイラには新たな対応が求められる．OSS の代表的なコンパイラである LLVM では，ARM SVE, RISC-V Vector への対応が 2016 年頃より議論され始めているが [3, 4]，中間言語 (IR) の拡張や従来の SIMD 向け自動ベクトル化機能の拡張については未だ明らかとなってないことも多く，開発者コミュニティの間で議論が行われている．

筆者らは，NEC のベクトル型コンピューターである SX-Aurora TSUBASA（以下 SX-Aurora）向けに，LLVM ベースのコンパイラを開発している [6]．NEC の SX シリーズのベクトルプロセッサは数十年に渡り前述した可変長のベクトル演算をサポートしており，我々の知る限り現時点で実プロセッサが利用可能な唯一の可変長ベクトル演算をサポートしたプロセッサである．我々の目的の一つは，このようなプロセッサ向けの OSS コンパイラを提供することとで，コンパイラ開発コミュニティでの議論が進むことに貢献することである [*1]．また，本 LLVM は可変長ベクトルサポートの第一歩として，ベクトル命令用の Intrinsic をサポートしている．熟練開発者向けに Intrinsic（組み込み関数）を提供することも我々の目的の一つである．

本稿では、SX-Aurora 向けの LLVM について，特にその特徴である可変長ベクトル向けの Intrinsic について説明する．まず第二章では，SX-Aurora TSUBASA についてベクトル命令を中心に説明する．第 3 章で SX-Aurora 向け LLVM の実装する可変長ベクトル向け Intrinsic の設計と実装について述べ，第 4 章ではその活用事例を簡単に紹介する．

[1] NEC データサイエンス研究所
Nakahara-ku, Kawasaki, Kanagawa 211–8666, Japan
[a)] k-ishizaka@ay.jp.nec.com
[b)] y-ideguchi@cj.jp.nec.com
[c)] y-ohno@ji.jp.nec.com

[*1] SX-Aurora 向けの LLVM のソースコードは github で公開されている．`https://github.com/sx-aurora-dev/llvm`

2. SX-Aurora TSUBASA

SX-Aurora は PCIe バスに接続されるアクセラレータカードである Vector Engine （以下 VE）上にベクトルプロセッサを搭載している．ホストプロセッサは Intel Xeon プロセッサである．VE の諸元を表 1 に示すが，VE は計算性能に対して高いメモリバンド幅を備えていることを特徴とし，B/F が比較的大きいワークロードを対象としていることがわかる．図 1 に VE の外観を示す．

表 1　Vector Engine (Type 10B) の諸元

計算性能 (単精度)	4.30 TFlops
計算性能 (倍精度)	2.15 TFlops
コア数	8
周波数	1.4 GHz
メモリサイズ	48 GB
メモリ帯域	1.2 TB/s

図 1　Vector Engine の外観

2.1 VE のベクトル命令セット

ベクトル命令の最大ベクトル長は MVL（Maximum Vector Length）レジスタで定義されるが，VE では 256 である．VE は 64bit データを 256 要素格納できるベクトルレジスタを 64 本備えている．またマスク命令に利用する 256bit のベクトルマスクレジスタを 15 本，スカラレジスタを 64 本を持っている．

2.1.1 ベクトル長

ベクトル命令のベクトル長（実際に演算が行われる要素数）は VL（Vector Length）レジスタで指定する．例えば，VL の値が 128 の時は，ベクトルレジスタ中の先頭 128 要素のみに演算が行われる．VL に値を設定するには LVL（Load Vector Length）命令を使う．以下に LVL 命令のニーモニックを示す．ここで，%sy はスカラレジスタ，I は即値を示す．すなわち，LVL の引数にはスカラレジスタまたは即値を指定できる．

```
lvl {%sy | I}
```

2.1.2 VE のベクトル命令の概要

ベクトルレジスタには，倍精度浮動小数点，単精度浮動小数点，整数などの任意の型のデータを格納でき，格納されているデータに応じてベクトル命令を使い分ける．例えば，浮動小数点の足し算の場合は，倍精度の場合は vfadd.d 命令，単精度の vfadd.s 命令であり，命令の suffix が要素のデータ長を示している．これらのニーモニックを以下に示す．

```
vfadd.d %vx, {%vy | %sy}, %vz[, %vm] // double
vfadd.s %vx, {%vy | %sy}, %vz[, %vm] // float
```

ここで，%vx，%vy，%vz はベクトルレジスタを示す．第二引数の {%vy | %sy} は，ベクトルレジスタもしくはスカラレジスタを指定することができることを示している．スカラレジスタが指定された場合は，スカラレジスタの値がベクトルレジスタのすべての要素に足される．最後の省略可能の引数の%vm は，ベクトルマスクレジスタである．

2.1.3 マスク演算

ベクトルマスクレジスタが指定された時は，対応するビットが 1 の要素のみ演算が行われる．対応するビットが 0 の要素については，出力レジスタの値は更新されない．

2.1.4 Packed 命令

また，VE のベクトル命令は packed 命令をサポートしている．ベクトルレジスタの一要素は 64bit であるが，ここに 32bit の値を 2 つ可能し，packed 命令を利用することで，512 個のデータに対して演算を行うことができる．単精度浮動小数点の足し算の packed 命令を以下に示す．

```
pvfadd %vx, {%vy | %sy}, %vz[, %vm]
```

packed 命令もマスク演算が可能である．packed 命令の場合は，512bit のマスクが必要なため，2 つの連続するベクトルマスクレジスタをペアで利用し，命令中には若い方のベクトルマスクレジスタを指定する．

2.2 可変長ベクトル命令による実装例

図 2 の DAXPY を行うプログラムのベクトル命令による実装例を図 3 に示す．このベクトルコードを分かりやすく説明するため，C 言語風の擬似コードで記述した例を図 4 に示す．

図 2　DAXPY のソースコード

```
1 void
2 daxpy(double* x, double* y, double a, int n)
3 {
4   for (int i = 0; i < n; ++i)
5     y[i] = a * x[i] + y[i];
6 }
```

擬似コードに示されているように，ベクトル命令を利用した場合は，一命令で最大 256 要素の計算が可能であるので，ループは 256 飛びとなる．図 4 の 6 行目はベクトル長を計算しており，ループの残り回転数と最大ベクトル長（256）の小さい方を選んでいる．これにより，ループ長がベクトル長で割り切れない時も，あまりループを作る必要

図 3　DAXPY のベクトル実装例

```
1   lea %s35,256                // s35 = 256
2   or %s36,0,%s3               // s36 = n
3 .loop:
4   mins.w.zx %s37,%s36,%s35    // s37 = min(s36,
      s35)
5   lvl %s37                    // VL = s37
6   vld %v0,8,%s0               // load x
7   vld %v1,8,%s1               // load y
8   vfmad.d %v0,%v0,%s2,%v1     // y = a * x + y
9   vst %v0,8,%s1               // store y
10  lea %s0,2048(%s0)           // x += 256
11  lea %s1,2048(%s1)           // y += 256
12  lea %s34,256(%s34)          // i += 256
13  lea %s36,-256(%s36)         // s36 = n - i
14  brlt.w %s34,%s3,.loop
```

図 4　DAXPY の擬似ベクトルコード

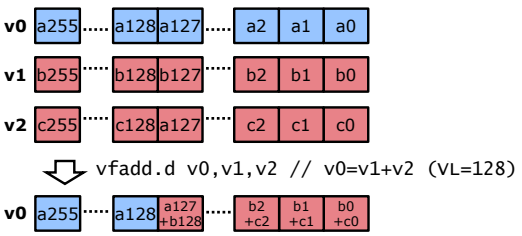
```
1 void
2 daxpy(double* x, double* y, double a, int n)
3 {
4   // MVL = 256
5   for (int i = 0; i < n; i += MVL) {
6     int VL = min(n - i, MVL);
7     y[i:i+VL-1]
8       = a * x[i:i+VL-1] + y[i:i+VL-1];
9   }
10 }
```

がない．例えば，ループ長が 1200 回転の場合を考えると，ベクトル長が 256 に固定の場合は，1024 回転を 4 回のベクトル命令で実行できるのに対して，176 回転のあまりをスカラ命令で実行する必要があり，結果としてベクトル化の効果を大幅に削減することになる．これに対して，ベクトル長が変更可能な場合は，あまりも 1 回のベクトル命令で実行することが可能である．

2.3　VE のベクトル命令の特徴

VE のベクトル命令の特徴として，図 5 に示すように，ベクトルレジスタの [VL:MVL-1] の要素は，値が更新されないことがある．例えば RISC-V Vector では，このような要素には 0 が代入される．

図 5　ベクトル長が短い場合のベクトル演算



これは，配列の全要素の和を求めるようなリダクション計算で有用である．リダクション計算のベクトル化では，図 6 の擬似コードに示すように，ベクトル要素数分の部分和をベクトルレジスタ上で計算し，最後にその和を求めるという方法が取られる．図では tmp が部分和用のベクトルレジスタを表している．あまりループがある場合は，最後の部分和の計算では，ベクトル長が MVL より小さくなるが，その場合も 0 クリアされないため，部分和を維持することができる．0 クリアや値が不定となる場合は，あまりループだけ別に計算するなどの追加のステップが必要となる．

図 6　配列のリダクション

```
1 tmp[0:MVL-1] = 0.0;
2 for (i = 0; i < n; i += MVL) {
3   int VL = min(n - i, MVL);
4   tmp[0:VL-1] += x[i:i+VL-1];
5 }
6 s = 0.0;
7 for (i = 0; i < MVL; ++i)
8   s += tmp[i];
```

3.　LLVM の VE 対応

SX-Aurora 向けの LLVM として，LLVM へ VE バックエンドの追加を行った．これにより，LLVM は VE 用のコードを生成するクロスコンパイラとして動作することが可能である．VE バックエンドでは，基本的なスカラ命令の実装に加えて，ベクトル命令向けの Intrinsics を実装している．冒頭で呼べたように可変長ベクトルに対する自動ベクトル化機能の実装は，現在開発者コミュニティで IR の拡張を含めて議論が行われている．このため今回の VE 対応では，自動ベクトル化はサポートしていない．

以下では VE のベクトル命令用の Intrinsics について，その課題と解決策について述べる．

3.1　可変長ベクトル命令用 Intrinsics の設計

LLVM はベクトル型拡張をサポートしており，VE 用 Intrinsics でもこれを用いて，以下のように 256 要素（64*256=16384bit）のベクトル型を定義している．本来は最大ベクトル長は MVL で定義されるため，コンパイル時には変数であるべきであるが，SX シリーズは数世代に渡り MVL として 256 を用いているため，現在の実装では，最大ベクトル長は固定としている．

```
typedef double __vr
  __attribute__((__vector_size__(2048)));
```

可変長ベクトル向けの Intrinsics として，まず LLVM の x86 バックエンドがサポートする固定長の SIMD 命令向けの Intrinsics に相当する Intrinsics 対して，LVL 命令（ベク

トル長設定命令）を追加した場合を考える．このような Intrinsics を VE Intrinsics と呼ぶことにする．VE Intrinsics による DAXPY の実装例を図 7 に示す．ほぼ図 4 と一対一に対応した形で記述されており，図 3 に示すアセンブリコードとも対応している．なお，Intrinsic 関数名の suffix は，戻り値と引数を表す．例えば”_vss” は，戻り値がベクトルで，スカラの引数が 2 つあることを示す．

図 7 VE Intrinsics による DAXPY の実装例

```
1 for (i = 0; i < n; i+=VL) {
2   int vl = min(n-i, VL);
3   _ve_lvl(vl);
4   __vr vx = _ve_vld_vss(8, &x[i]);
5   __vr vy = _ve_vld_vss(8, &y[i]);
6   vy = _ve_vfmadd_vvsv(vx, a, vy);
7   _ve_vst_vss(vy, 8, &y[i]);
8 }
```

この Intrinsics の利点としては，アセンブリコードとの一致性が高く，命令セットに詳しいユーザーにとって親しみやすい点が挙げられる．

3.1.1 Intrinsics 設計の課題

Intrinsics は熟練開発者がアセンブリコードを書くのを効率化するという狙いがあるため，この利点は重視されるべき点であるが，一方でこの VE Intrinsics には，ベクトル長の変更への対応が難しいという課題がある．

その理由として，2 つの点があることがわかっている．これらを図 8 を例に説明する．このコードは，まずベクトル長を 256 に設定し，vx，vy，vz へデータをロードし，その後ベクトル長を 128 に設定して，vz へ vx と vy の和を代入している．最後に再びベクトル長を 256 に設定し，vz をストアしている．すなわち，z の前半 128 要素のみ x と y の和に更新されることを想定している．

図 8 VE Intrinsics でのベクトル長変更の例

```
1 __vr vx, vy, vz;
2 _ve_lvl(256);
3 vx = _ve_vld_vss(8, px);
4 vy = _ve_vld_vss(8, py);
5 vz = _ve_vld_vss(8, pz);
6 _ve_lvl(128);
7 vz = _ve_vfaddd_vvv(vx, vy);
8 _ve_lvl(256);
9 _ve_vst_vss(vz, 8, pz);
```

ひとつ目の理由は，Intrinsic 関数_ve_lvl で設定したベクトル長が，実際のベクトル命令で想定通りに利用されない場合があることである．これは，図のコードを見ればわかるように，ソースコード上では，_ve_lvl とベクトル命令（_ve_vld_vss や_ve_vfaddd_vvv など）に対応する Intrinsic 関数の間に依存関係がない．このためコンパイラは，これらの関数を移動させることができ，その結果ベクトル長が正しく設定されない可能性がある．これに対する解決策としては，ソースコード上に現れない暗黙の依存を導入することや，すべての Intrinsic 関数の属性として「副作用あり」と設定することで，volatile のように関数の移動を禁止するという方法があるが，LLVM への実装コストや性能への悪影響などが懸念される．

ふたつ目の理由は，ベクトル長が最大ベクトル長より短い場合に，結果不正が起る可能性があることである．現在の LLVM では，各行でのベクトル型変数への代入は，全要素への代入として扱われるため，前半 128 要素のみ代入を意図した 7 行目の代入でも，全要素が代入されるとして扱われる．このため，5 行目で vz へのロードから，7 行目の vz への依存がはられず，5 行目はデットコードして扱われ，結果として z の後半 128 要素の値は不定となる．

これを修正する方法として，ベクトル型変数への代入のセマンティクスを変更し，コンパイラの依存解析系を対応させる方法があるが，修正規模が非常に大きくなることが予想される．

3.2 ベクトル長明示型 Intrinsics

そこで，我々は可変ベクトル長向けの IR の拡張提案 [5] に Inspire され，新たな Intrinsics を設計することにした．この Intrinsics は，

- ベクトル長は，個々の Intrinsic 関数の引数として指定する．
- MVL よりも短い時に，ベクトル長以降の要素（[VL+1:MVL]）に代入する値（pass through）も，Intrinsics の引数として指定する

という 2 つの特徴を持つ．この Intrinsics を本稿では VEL Intrinsics と呼ぶ．

これらの特徴は，前節で述べた 2 つの課題に対応する．ひとつ目の特徴は，Intrinsic 関数毎にベクトル長を明示することで，VL を設定する Intrinsics （_ve_lvl）自体を不要とし，対応関係が崩れる課題を解決する．ふたつ目の特徴は，引数に pass through を導入することで，VL が MVL より短い場合でも，Intrinsic 関数は出力ベクトル型変数の全要素を定義することが可能となる．これにより，ベクトル型変数へ代入のセマンティクスを変更することなく，命令間の依存関係を正しく設定することが可能となり，ふたつ目の課題を解決する．

前記のプログラムを VEL Intrinsics で記述した例を図 9 に示す．Intrinsic 関数の_vel_vfaddd_vvvl は，3 番目の引数として，pass through 値を指定している．ここに vz を指定することで，後半 128 要素の値を保たれるようにすることができる．なお，pass through 値が不要な場合を考え

て，各 Intrinsic 関数には，pass through 引数がある関数とない関数の二種類を用意している．

図 9　VEL Intrinsics でのベクトル長変更の例

```
1 __vr vx, vy, vz;
2 vx = _vel_vld_vssl(8, vx, 256);
3 vy = _vel_vld_vssl(8, vy, 256);
4 vz = _vel_vld_vssl(8, vz, 256);
5 vz = _vel_vfaddd_vvvvl(vx, vy, vz, 128);
6 _vel_vst_vssl(vz, 8, pz, 256);
```

3.3　ベクトル長設定命令の生成

VEL Intrinsics では，ベクトル長を設定する LVL 命令に対応する Intrinsic 関数はない．このため，プログラムを正しく動かすためには，コンパイラで LVL 命令を生成する必要がある．そこで，VE バックエンドでは LVL 命令を生成する新たなパス LVLGen を導入した．最もシンプルな LVL 命令生成のアルゴリズムは，すべてのベクトル命令の前に LVL 命令を挿入することであるが，性能への影響が懸念されるため，LVL 命令数を削減するためのアルゴリズムを採用した．

LVLGen のアルゴリズムを図 1 に示す．本アルゴリズムでは，Basic Block（BB）単位で連続するベクトル命令が同じベクトル長かどうかを判断し，ベクトル長が同じと判断できない場合だけ，LVL 命令を生成する．

Algorithm 1 LVLGen: Generation of LVL instruction

```
for all Basic Block BB in a function do
  CurrentVL ← None
  for all Instruction I in a BB do
    if I is Vector Instruction then
      L ← vector length of I
      if CurrentVL is None or CurrentVL is not L then
        generate a LVL instruction to load L to VL register
      end if
      CurrentVL ← L
    end if
  end for
end for
```

このアルゴリズムでは，BB 間で無駄な LVL 命令が生成される可能性があるが，多くの場合は，Intrinsics で記述するようなループでは，ループボディはひとつの BB で構成されることが多いため，このようなシンプルなアルゴリズムで性能上は問題ないと想定している．

3.4　マスク付き packed 命令への対応

前述したように 512 要素への演算を行う packed 命令では，二つの連続するベクトルマスクレジスタをペアで利用する．しかしながら，Intrinsics では物理レジスタの番号を指定することができない．そこで，VE バックエンドでは，マスクレジスタに対応する 256b のベクトル型（__vm）に加えて，512b の新たなベクトル型（__vm512）を定義し，これに対してベクトルマスクのペアを割り当てるようにしている．

また，ベクトルマスクレジスタに対する AND や OR などの演算が行う命令があるが，ISA 上はベクトルマスクレジスタをペアで扱うための命令はなく，二つのベクトルマスクレジスタに対して同じ演算を行う必要がある．Intrinsics では，仮想的な__vm512 型の変数に対して演算を行う Intrinsics を定義し，コンパイラで 2 つの命令に展開している．これにより，Intrinsics でもマスク付きの packed 命令を使うことを可能としている．

3.5　Intrinsics の実装

LLVM では，Intrinsics を実装するには，多数のファイルを変更する必要があり，その手順が煩雑である．そこで，今回の Intrinsics の実装では，ベクトル命令を定義する DSL を開発し，その DSL から LLVM のソースコードを自動生成するようにした．

例えば，倍精度浮動小数点の足し算命令は以下のように定義される．

```
Def(0x82,           # opcode
    ''VFAD'',       # instruction name
    ''d'',          # sub opcode (double)
    ''vfadd.d'',    # asm
    [[VX(T_f64), VY(T_f64), VZ(T_f64)]], #
    args
    ''{1} = {2} + {3}'')  # description
```

この定義から，Intrinsics の関数名や LLVM 内部での命令名などを自動で決定し，LLVM での命令定義，Intrinsics の定義，Clang 用の Builtin 関数の定義，ベックエンドでの命令生成ルールの定義などを生成する．加えて，”description” を利用して，テストコードの生成やマニュアルの作成も行う．本 DSL は python 内の言語内 DSL として実装してあり，python の文法・機能を利用して，命令定義を記述することが可能である．

3.6　動作確認

最初に述べたベクトル長の変更に対応していない VE Intrinsics と，次に述べた対応している VEL Intrinics を実装し，動作確認を行った．

図 8 の VE Intrinsics で実装されたコードのコンパイル結果を図 10 に示す（主要部分のみ示している）．前述したように，vz へのロードはデッドコードと判断されコンパイラの最適化により消されている．vfadd.d 命令の結果は px が格納されている v0 に格納され（行 8），これが vst 命令によってストアされている（行 11）．結果として，

pz[128:255] には px[128:255] が格納されており，想定とおりのコードになっていない．

図 10 図 8 のコンパイル結果

```
 1 lea %s34, 256
 2 lvl %s34
 3 stl %s34, -24(,%s9)     // spill VL
 4 vld %v0,8,%s0           // load px to v0
 5 vld %v1,8,%s1           // load py to v1
 6 lea %s34, 128
 7 lvl %s34                // VL = 128
 8 vfadd.d %v0,%v0,%v1     // v0 = v0 + v1
 9 ldl.sx %s34, -24(,%s9)  // restore VL
10 lvl %s34                // VL = 256
11 vst %v0,8,%s2           // store v0 to pz
```

一方，図 9 に示す VEL Intrinsics で実装されたコードのコンパイル結果を図 11 に示す．この場合は，vfadd.d 命令の結果は pz がロードされている v2 に格納され（行 8），それが pz にストアされており（行 10），想定通りの動作をするコードが生成されている．

図 11 図 9 のコンパイル結果

```
 1 lea %s34, 256
 2 lvl %s34              // VL = 256
 3 vld %v0,8,%s0         // load px to v0
 4 vld %v1,8,%s1         // load py to v1
 5 vld %v2,8,%s2         // load pz to v2
 6 lea %s35, 128
 7 lvl %s35              // VL = 128
 8 vfadd.d %v2,%v0,%v1   // v2 = v0 + v1
 9 lvl %s34              // VL = 256
10 vst %v2,8,%s2         // store v2 to pz
```

4. LLMV-VE の活用

筆者らは，ソフトウェアを VE へ移植する際の最適化に開発した LLVM を利用している．そのようなアプリケーションには以下のようなものが含まれる．これらのアプリの詳細については文献 [7,8] などを参考にされたい．

- VE 対応 TensorFlow が利用する ML カーネル [7]
- VE 用 Deep Learning ライブラリ veDNN [7]
- 3D Localization などで使われる画像処理カーネル [8]
- pword2vec（word2vec の実装）
- RabbitCT（CT 画像再構成）

5. おわりに

ARM SVE や RISC-V Vector 拡張などの登場により，LLVM では可変長ベクトル命令への対応が盛んに議論されている．SX-Aurora は，筆者らの知る限り可変長ベクトル命令を持つ現在唯一の市販コンピューターである．本稿では筆者らが開発・公開している SX-Aurora 向けの LLVM について説明した．本 LLVM では可変長ベクトル向けの Intrinsics を実装しており，可変長ベクトルに触れることができる．本コンパイラと実プロセッサを活用することで，コンパイラの研究開発が進むことを期待している．

参考文献

[1] Stephan, N. et al., The ARM Scalable Vector Extension, IEEE MICRO VOL. 37, ISSUE. 2, MARCH - APRIL 2017
[2] RISC-V ”V” Vector Extension, https://riscv.github.io/documents/riscv-v-spec/riscv-v-spec.pdf
[3] Huter, G., [llvm-dev] [RFC] Supporting ARM's SVE in LLVM, Nov., 2016, http://lists.llvm.org/pipermail/llvm-dev/2016-November/106819.html
[4] Kruppe, R., Adventures with RISC-V Vectors and LLVM, EuroLLVM Developers' Meeting, Apr, 2019.
[5] Moll, S., RFC: Prototype & Roadmap for vector predication in LLVM, Jan., 2019, https://reviews.llvm.org/D57504
[6] Focht, E., [llvm-dev] [RFC] NEC SX-Aurora VE backend, Apr, 2019, http://lists.llvm.org/pipermail/llvm-dev/2019-April/131580.html
[7] 大野 善之 他, SX-Aurora TSUBASA の Deep Learning への適用, SWoPP, Jul., 2019.
[8] 井手口 裕太 他, ベクトルプロセッサを用いた三次元位置追跡, SWoPP, Jul., 2019.

付　　録

A.1 略語

LVL ベクトル長レジスタに値を設定する命令
MVL 最大ベクトル長
VE Vector Engine
VL 実効ベクトル長（実際に演算を行う要素数）

A.2 本稿に登場するベクトル命令

vfadd.d 倍精度浮動小数点の加算
vfadd.s 単精度浮動小数点の加算
vfmad.d 倍精度浮動小数点の FMA
pvfadd 単精度浮動小数点の packed 加算 (512 要素演算)
vld ベクトルロード
vst ベクトルストア

A.3 VE のレジスタ

%sn スカラレジスタ (n=0-63)
%vn ベクトルレジスタ (n=0-63)
%vm$n$ ベクトルマスクレジスタ (n=0-15)