ステージング型クエリコンパイラにおけるSIMD命令活用の提案と初期評価

根本　潤[†]　川島英之[†]　遠山元道[†]

1. はじめに

RDBMSにおける先進的なクエリエンジンの多くは、クエリ実行プランを逐次解釈するインタープリタではなく、クエリ実行プランから効率的な機械語をJust-In-Time(JIT)に生成・実行するクエリコンパイラとして実装されている。このようなクエリコンパイラの最新研究として、ステージング型のクエリコンパイラLB2がある[1)]。ステージング型のクエリコンパイラでは、部分評価を用いた最適化により、高級言語でインタープリタを実装するのと同等の開発効率で、高速なクエリ処理を実現する。一方、近年、CPUの命令レベルでのデータ並列化のため、クエリコンパイラにおいてSIMD命令を活用する研究も行われている[2)]。ただ、このようなハードウェアに特化したコード生成は、複雑かつ困難であるという課題がある[3)]。そこで、本研究では、ステージング型の高い生産性を維持しつつ、SIMD命令を活用した高性能なクエリ処理コードを自動生成するクエリコンパイラを提案する。本稿では、試作したクエリコンパイラの初期評価について報告する。

2. 関連研究

文献1)は、インタープリタとソースプログラムを入力として、より効率的でかつ同一の結果を出力するコンパイル済みプログラムに変換するという二村射影の考え方を参考に、ステージング型のクエリコンパイラLB2を提案している。LB2は、クエリ実行プランをインタープリタ的に解釈する過程(第1ステージ)で部分評価を行い、特化された効率的なCコードを生成することで、高速なクエリ処理(第2ステージ)が可能である。具体的には、LB2はライブラリベースの生成的プログラミングフレームワークであるLightweight Modular Staging (LMS)[4)]を利用しており、第2ステージで処理する式をLMSが提供する型コンストラクタRep[T]を用いて明示することでコード生成を実現している。

LB2では、SIMD命令のような最新ハードウェアを積極的に活用するような高速化手法については議論されていないが、文献2)では、Relax Operator Fusion(ROF)と呼ばれるクエリ処理モデルを導入することで、クエリコンパイラにおいてSIMD命令を活用する方法が提案されている。昨今のクエリコンパイラは、通常、Data-Centric(もしくはプッシュ型)と呼ばれるモデルを採用しており、効率化のため、クエリの各オペレータ間では極力タプルのマテリアライズ(バッファへの書き出し)を行わないようにするのが一般的だが、ROFでは、クエリ実行プランの中にあえてマテリアライズを行うポイントを設けることで、SIMD命令を活用可能にする。ただ、文献3)において言及されているように、Data-CentricなモデルにおけるSIMD活用コードの生成は複雑であり、その容易化は引き続き重要な研究課題である。

3. 提案システムと試作

提案するSIMD命令を活用したステージング型クエリコンパイラのシステムアーキテクチャを**図1**に示す。提案システムでは、クエリ実行プランを入力として、それを逐次的に解釈し、Cコードを生成する。そして、生成したCコードをJITコンパイルし、実行することで問合せ結果を応答する。コード生成にはLB2と同様にLMSを使用するが、ハードウェアに特化した機能はないため、LMSを拡張することでSIMD拡張命令セットであるAVX-512に対応したコード生成を可能にする。

試作では、クエリ実行プランのインタープリタとして、通常のスカラーデータのまま処理する選択、結合

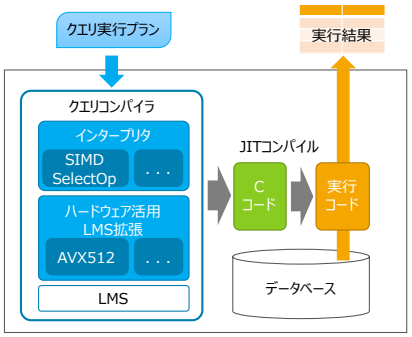

図1　提案システムアーキテクチャ

† 慶應義塾大学
Keio University

や集約といった各種オペレータの他、SIMD 命令で処理する選択オペレータを実装した。LMS を利用するため、クエリコンパイラ自体は Scala で実装した。SIMD 対応選択オペレータの疑似コードを以下に示す。

```
1  class SelectVectorOp(child: Op,
2                       predicates: Seq[Predicate]){
3    def exec = {
4      val buffer = new RecordBuffer(16)
5      val indexBuf = new Buffer[Int](16)
6      val indexVec = Vector16(15, 14, ..., 1, 0)
7      val foreachRecord = child.exec
8      (callback: RecordCallback) => {
9        foreachRecord { record =>
10         buffer.add(record)
11         if (buffer.len == 16) {
12           var mask: Mask16 = Mask16FromInt(0xffff)
13           predicates.foreach { pred =>
14             mask = mask
15               && evalVectorPredicate(pred, buffer)
16           }
17           indexVec.toBuffer(indexBuf, mask)
18           for (i <- 0 until mask.popcount) {
19             callback(Record(buffer(indexBuf(i))))
20           }
21           buffer.clean
22         }
23       }
24       // Scalar operations for the rest are omitted
25     } // Retruns (RecordCallback => Unit) function
26 } }
```

一旦意図的にマテリアライズを行ったのち (10 行目)、15 行目の evalVectorPredicate() において各述語を SIMD 命令で一括評価していく。そして、得られたビットマスクとインデックスベクターを用いて真となったタプルのインデックスをバッファに戻し、そのインデックスに該当するタプルを次のオペレータへと引き渡していく。なお、Vector16 や Mask16 は、拡張した LMS で定義した Rep[__m512i] 型や、Rep[__mmask16] 型の演算をカプセル化したクラスであり、こうした抽象化が可能なのがステージング型クエリコンパイラの特徴の 1 つである。

4. 初期評価

試作したクエリコンパイラを用いて簡易的な評価を行った。評価環境は、CPU が Intel Xeon Platinum 8176 2.10GHz、メモリが 1TB，OS が Ubuntu 18.04(Linux Kernel 4.15.0) であり、コンパイラは GCC 7.4.0 と Clan 6.0.0(最適化フラグはいずれも-O3) を使用した。ベンチマークとしては TPC-H(SF=1) を使用し、クエリは文献 2) において、SIMD 化の効果が低かった Q1 と効果が高かった Q14 を用いて評価した。結果を**図 2** に示す。

図 2 に示すように、Clang でコンパイルした場合、SIMD 化により Q1 が約 24%、Q14 が約 4%性能が低下するという結果であった。perf コマンドにより CPU 統計情報を確認したところ、Q14 では、分岐ミスは約半分に減少したものの、命令数が約 9%増加しており、バッファリングのペナルティが SIMD 化の効果を上回ってしまったことが原因であると推測できる。

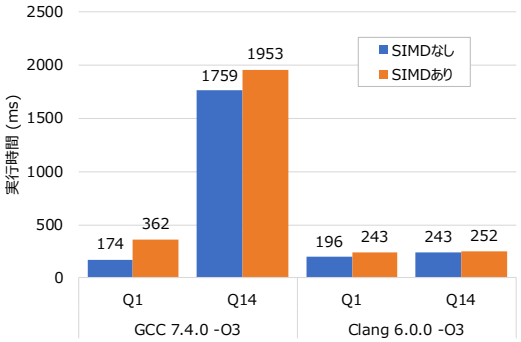

図 2 TPC-H 評価結果 (SF=1)

5. おわりに

本研究では、高性能で開発効率の高いクエリコンパイラの開発を目的として、SIMD 命令を活用したステージング型クエリコンパイラを提案した。試作と初期評価の結果、文献 2) や文献 3) で主張されている SIMD 化の効果を再現することはできなかった。

今後は、より大規模なデータセットでの評価や生成コードや実行コードの分析を通じて性能改善を試みる予定である。

謝辞

本研究は，NEDO の「実社会の事象をリアルタイム処理可能な次世代データ処理基盤技術の研究開発」の委託により実施したものである．

参考文献

1) R. Y. Tahboub, G. M. Essertel, and T. Rompf, “How to architect a query compiler, revisited,” in *Proceedings of the 2018 International Conference on Management of Data*, pp. 307–322, June 2018.
2) P.Menon, T.C. Mowry, and A.Pavlo, “Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last,” *Proc. VLDB Endow.*, vol. 11, pp. 1–13, Sept. 2017.
3) T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. Boncz, “Everything you always wanted to know about compiled and vectorized queries but were afraid to ask,” *Proc. VLDB Endow.*, vol. 11, pp. 2209–2222, Sept. 2018.
4) T. Rompf and M. Odersky, “Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls,” *Commun. ACM*, vol. 55, pp. 121–130, June 2012.