コンパイラ共通基盤COINSのLLVM向け拡張

酒井 宏城[1,a)]　澄川 靖信[1,b)]　滝本 宗宏[1,c)]

概要：本論文ではコンパイラ共通基盤 COINS の中間表現を LLVM の中間表現に変換するモジュールの実装を示す．本変換モジュールによって，COINS における既存の実装を活かしながら，さらに高品質なコード生成や，GPGPU 向けコード生成といった応用が可能になる．本変換モジュールは COINS の低水準中間表現における最適化モジュールの 1 つとして実現したものであり，その性能を示すために，SPEC ベンチマークによる評価を行った．

キーワード：COINS，LLVM，コンパイラ共通基盤，コンパイラ，中間表現，コード最適化

An Extension to COINS for LLVM

Abstract: We propose a transformation module that transforms the intermediate representation of COINS compiler infrastructure into one of LLVM. The transformation module enables COINS to emit better code or the code for GPGPU with the capability of LLVM. We have implemented the module as an optimization module for the low-level intermediate representation on COINS, the effectiveness of which is shown through some experiments on SPEC benchmarks.

1. はじめに

コンピュータではより高速・高効率・低消費電力な処理を実現するため，従来の CPU による処理に加え，様々なハードウェアアーキテクチャ的な工夫が検討・実装されている．具体的な例としては以下のようなものがある．

- マルチコア・メニーコア
- 汎用演算における GPU の活用 (GPGPU)
- SIMD 命令の活用
- FPGA・リコンフィギャラブルシステム

このようなハードウェアを効率よく利用するためにはコンパイラの対応が不可欠である．特に，近年，コンパイラ共通基盤の 1 つである低水準仮想機械 (LLVM: Low-Level Virtual Machine)[1] に対応するバックエンドを実装することでハードウェアの活用を進める動きが盛んになっている．例えば，NVIDIA 社の GPU で用いられる中間表現 PTX を生成するバックエンドモジュールが LLVM には統合された．この統合によって，LLVM の中間表現を経由して GPGPU 向けコードを生成することができる．

[1] 東京理科大学
Tokyo University of Science
[a)] h-sakai@cs.is.noda.tus.ac.jp
[b)] yas@cs.is.noda.tus.ac.jp
[c)] mune@cs.is.noda.tus.ac.jp

一方，他のコンパイラ共通基盤の 1 つである COINS [2] は，最適化器の実装を容易とする中間表現として，低水準中間表現（low-level intermediate representation, LIR）[3] を備えており，LIR 上で多くの最適化器が実現されている．

本研究では，LLVM の中間表現を出力する LIR2LLVM を実装した． 本研究の実現によって，LIR の簡便さを保ちながら，LLVM のコンポーネント群・バックエンド群を利用できる．

以降の構成は次のとおりである．第 2 節で本稿の理解のために必要な前提知識を説明する．第 3 節で提案手法である LIR2LLVM の詳細を述べ，第 4 節で実験結果とその評価を述べる．第 5 節で関連研究を述べた後，第 6 節で今後の課題を述べ，第 7 節でまとめる．

2. 前提知識

コンパイラ共通基盤とは，コンパイラ開発に必要な，字句解析器や構文解析器，最適化器といった共通のモジュールを提供するものである．コンパイラ・インフラストラクチャを用いることによって，コンパイラ開発者は，コンパイラを最初から実装することなく，必要なモジュールを実装するだけで済むようになる．

コンパイラの多くは基本ブロックを節点とする制御フローグラフ（control flow graph，以降 CFG と呼ぶ）によっ

て，制御の流れを表し，基本ブロック内の各命令を中間表現によって表す．

中間表現は，**3 番地コード** [4] という形式で表されることが多い．3 番地コードは，1 命令がせいぜい 3 つの番地と 1 つの演算子を含む次のいずれかの形式をとる．

- $x = y\ op\ z$
- $x = op\ y$
- $x = y$
- `goto` L
- `if` x `goto` L

それぞれが 2 項演算の代入，単項演算の代入，コピー代入，無条件ジャンプ，条件付きジャンプを表す．ただし，op は単項演算子あるいは 2 項演算子であり，x, y, z は変数あるいは定数であり，L は特定の 3 番地命令に付与されたラベルを表す．

さらに，3 番地コードは**静的単一代入**（static single assignment, SSA）形式 [5] をとる場合もあり，本研究が生成する LLVM の中間表現としても採用されている．SSA 形式は，各変数に対して定義が 1 つとなるように変換されたプログラム形式であり，変数の定義ごとにバージョン付けを行うことで実現するのが一般的である．SSA 形式のプログラムでは，通常形式のプログラムで同一であった変数が，別々の変数として表現されることがあり，複数の定義が同一の CFG 節に到達する場合，それらを引数とする仮想関数である ϕ 関数を挿入しなければならない．ϕ 関数を挿入する節は，定義を含む節点が初めて支配できなくなる節点（以降，支配辺境（dominance frontier）と呼ぶ）と，支配辺境のさらなる支配辺境（以降，反復支配辺境（iterated dominance frontier）と呼ぶ）である．ここで，節点 m が節点 n を支配するとは，開始節から節点 n へ到達する任意の実行経路に節点 m が存在するということである．

LLVM の中間表現では，ϕ 関数を記述するのではなく，ϕ 関数が必要な変数をメモリで管理する方法を用いている．ϕ 関数を明示するためにはオプションを指定する必要がある．

2.1 COINS

COINS はコンパイラ共通基盤の 1 つである．COINS は，高水準中間表現（high-level intermediate representation, HIR）と LIR の 2 つの中間表現をもつ．

COINS は次の手順によって目的コードを生成する．

(1) 字句解析，構文解析を行い，原始言語から HIR および記号表を生成する
(2) HIR 最適化を行う
(3) HIR から LIR への変換を行う
(4) LIR 最適化を行う
(5) 目的言語のコード生成を行う

これらは典型的な手順であるが，例えば HIR から C 言語への変換を行うといった，標準的な手順から外れたモジュールを実装することも容易である．

図 1 LIR の構造
Fig. 1 Structure of LIR module

表 1 LIR のデータ型
Table 1 Types of LIR

型名	サイズ (bit)	意味
In	n	整数型
Fn	n	浮動小数点数型
An	n	集成体型

2.1.1 LIR

図 1 に LIR の階層構造を示す．LIR に変換されたプログラムは **L モジュール**と呼ばれるコンパイル単位に分割される．L モジュールは，モジュール名，大域変数に関する記号表，関数定義である **L 関数**，**L データ**と呼ばれるデータ部分を内部に持つ．L 関数は局所変数の記号表と **L シーケンス**と呼ばれる命令列からなり，各命令は **L 式**と呼ばれる．

L 式における LIR のデータ型は **L 型**と呼ばれ，**表 1** に示す 3 つの型からなる．表 1 の n はビットサイズを表す自然数である．例えば `I32` 型は 32 ビット整数型，`F64` は 64 ビット浮動小数点数型をそれぞれ表す．A 型は C 言語でいう構造体や共用体といった，複数の型が合成された型に割り当てられる．

L 式はさらに，L 型を含んでいるかや副作用を含むかどうかによって文法上の種類がある．[*1]LIR2LLVM に関連した種類について，例を**表 2** に示す．表中の t は L 型を表す．

LIR では，ロード命令とストア命令をそれぞれ，オペランドがメモリアドレス番地である MEM 式と代入文を用いて実現している．例えば，アドレス `addr` の値を仮想レジスタ `x` にロードするとき，`(SET (REG x) (MEM addr))` という文が生成される．なお，ここでは説明を容易にするために型情報は省略している．同様に，仮想レジスタ `x` をアドレス `addr` へストアするとき，`(SET (MEM addr) (REG x))` という文が生成される．

[*1] 詳細な文法定義は [3] を参照されたい．

表 2 L 式の一例
Table 2 A part of kinds of L-expression

種類	フォーマット例	例の意味
Const 式	(INTCONST t z)	整数定数 z
Addr 式	(STATIC t s)	シンボル s で表されるメモリアドレス
Reg 式	(REG t s)	シンボル s で表されるレジスタの値
Pure 式	(ADD t x y)	x と y の加算
Mem 式	(MEM t x)	アドレス x に格納されている値
Set 式	(SET t (REG t s) x)	レジスタ s に値 x を代入する
Jump 式	(JUMP l)	ラベル l への無条件ジャンプ
Call 式	(CALL addr ($x_1, \cdots, x_n$) ($y_1, \cdots, y_m$))	アドレス addr に多値関数があると仮定して引数 $x_1, \cdots, x_n$ で関数呼び出しを行い結果を $y_1, \cdots, y_m$ に代入する

2.2 LLVM

LLVM はプログラムのコンパイル時，リンク時，実行時など，プログラムのライフサイクルにおけるさまざまな機会を捉えて最適化を行うための仮想機械のプロジェクトとして開始されたコンパイラ共通基盤の 1 つである．LLVM のサブプロジェクトとして **Clang** があり，LLVM 向けの C コンパイラを提供している．

LLVM の中間表現は **LLVM IR** と呼ばれ，多くの型が定義されている．本稿では，LIR2LLVM に関連する型を示す．

- 基本型
  - 整数型
  - 浮動小数点型
  - void 型
  - label 型
- 派生型
  - ポインタ型
  - 配列型
  - 構造体型

LIR と同様に，整数型についてはビットサイズを指定できる．すなわち in は n ビット整数型を表す．また，型 t の要素が n 個並んだ配列型は [$n \times t$] で表され，型 t へのポインタ型を t*と書く．

LLVM IR には，多くの命令が用意されているが，型と同様に単純な LIR からの変換として考慮しなければならない命令は比較的限定される．ここでは [6] および LLVM リファレンス中から，本研究に必要な LLVM IR 命令の例を**表 3** に示す．ただし，t は LLVM の型を，x,y,p,result は変数ないしリテラルを表す．

LLVM IR は**図 2** に示すように，モジュールと呼ばれる単位に分割される．モジュール内にはグローバル定数，変数の宣言，および外部宣言，関数定義，メタデータの定義を含む．

LLVM IR は SSA 形式を採用しているが，メモリへの読み書きを用いることで実質的に変数（値の書き換え）を実現することができる．次のような C 言語プログラム片があるとする．

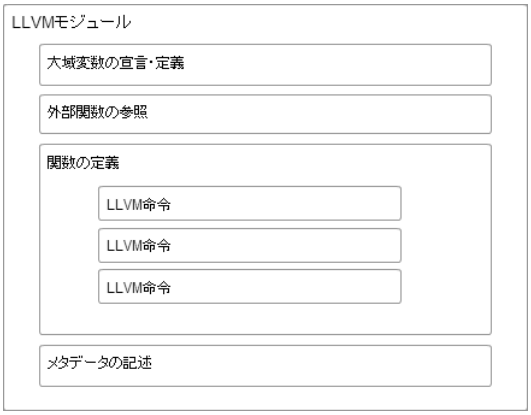

図 2 LLVM IR の構造
Fig. 2 Structure of LLVM IR

```
int x = 1;
x = x + 3;
```

Clang では対応部分が以下のような LLVM IR コードに変換される．

```
%x = alloca i32, align 4
store i32 1, i32* %x, align 4
%0 = load i32* %x, align 4
%add = add nsw i32 %0, 3
store i32 %add, i32* %x, align 4
```

このプログラム片では，alloca 命令で確保した領域に対しての読み書きを load/store 命令で行うことで，変数としての振舞いを実現している．

3. LIR2LLVM

本節では，提案手法である LIR2LLVM について説明する．

LIR と LLVM IR はどちらも低水準な中間表現なので，各命令は比較的似ている．また，モジュール構造も類似しており，L モジュール・L 関数はそれぞれ LLVM IR のモジュール・関数に対応づけられる．

しかしながら，次の 2 つの問題がある．

表 3 LLVM IR 命令の一部
Table 3 A part of kinds of instruction in LLVM IR

種類	フォーマット例	例の意味
終端命令	`ret` *t* `x`	値 x を返し，関数の呼び出し元へ制御を移す.
二項演算子	`result = add` *t* `x y`	x と y を加算し，その結果を result に代入する
メモリアクセス演算子	`result = load` *t*`* p`	アドレス p から値を読み出し，result に代入する
	`store` *t* `x` *t*`* p`	値 x をアドレス p が指す領域に保存する
	`result = alloca` *t*	*t* 型に必要な領域を確保し，領域へのポインタを result に代入する
型変換演算子	`result = inttoptr` *t* `x to` *t'*	整数 x をポインタ *t'* 型に変換したものを result に代入する
	`result = ptrtoint` *t* `x to` *t'*	ポインタ x を整数 *t'* 型に変換したものを result に代入する

- L 型から LLVM IR の型への変換
- L 式から LLVM IR の命令への変換

以降でこれらの問題について詳しく述べる.

3.1 L 型から LLVM IR の型への変換

L 型，および LLVM IR の型は，共に整数型，浮動小数点数型を持つので対応付けができる．また，L 型の A 型を LLVM IR の構造体型に対応付けることもできる.

しかし，この単純な方針では問題が生じる場合がある．次の C コード

```
char *str = "str";
```

は，LIR では

```
(SET I64 (MEM I64 (FRAME I64 "str.1"))
         (STATIC I64 "string.4"))
```

のようになる．LIR ではポインタ型が無いので，変数 str は I64 型で表現されている．[*2] しかしながら，LLVM において文字列リテラルは配列型で表されるので，str は [4 x i8]*型となってしまい，型が合わなくなる.

本手法では，この問題に対して，LIR のアドレスを表す STATIC 式，FRAME 式を，LLVM IR の ptrtoint 命令を用いて変換する．したがって上述の C プログラムは，LIR を経由して次の LLVM IR に対応づける.

```
%str.1.addr = alloca i64
%0 = ptrtoint  [4 x i8]* @string.4 to i64
store i64 %0, i64* %str.1.addr
```

3.2 L 関数の変換

多くの L 関数は LLVM IR の関数にそのまま変換できる．しかしながら，いくつかの関数において，次の 3 つの問題がある.

- コマンドライン引数の情報を保持する main 関数の仮引数 argc と argv の型の整合性が取れていない．LIR では，それぞれの型が I32，I64 であるのに対し，LLVM IR では i32，i8**である.
- 規格 C では，戻り値の型が int 型である main 関数において，return 文が省略された場合，関数は 0 を返すと規定されているが，LIR では何も返さないことになっている.
- 入力プログラムにおいて，非 void 関数で return 文が省略されている場合，LIR では戻り値の型情報が記録されない.

LIR2LLVM では，最初の問題を main 関数だけを特別扱いし，仮引数がある場合にはその型を i32，i8**とすることによって対応した.

同様に，2 つ目の問題についても，main 関数を特別扱いし，関数本体の末尾に ret i32 0 を挿入することで対応した.

3 つ目の問題は，LIR で型情報が得られないことから，LIR の変換元である HIR の記号表を参照することによって型情報を補完している.

3.3 L 式から LLVM IR 命令への変換

多くの L 式は，LLVM IR に単純に変換できる．しかしながら，次の 3 つの問題が存在する.

- LIR には符号反転を行う NEG 式が存在するが，LLVM IR には存在しない．LIR2LLVM は，NEG 式を 0 からオペランドを減算するように変換する．すなわち (NEG x) は sub *t* 0，%x と変換する.
- LIR では 2 項演算子を整数型と浮動小数点型の区別をすることなく同じ演算子を使用しているが，LLVM では型ごとに異なる演算子を使用している．例えば，LLVM IR では加算演算子として，整数型用の add と浮動小数点数型用の fadd がある．したがって LIR2LLVM は L 型に応じて適切な LLVM IR 命令を選択する.
- LIR には論理否定を行う BNOT 式が存在するが，LLVM IR には存在しない．LIR2LLVM は BNOT 式を-1 との排他的論理和を取るように変換する．すなわち (BNOT x) は xor *t* -1，%x と変換する.

ただし，*t* は L 型から変換された LLVM IR の型である.

[*2] 理解を容易にするため，ターゲット環境として 64 ビット環境を仮定している．以降の例について，特に断りがない場合も同様である.

3.4 変数の変換

本節では，大域変数および局所変数の変換方法について記述する．

3.4.1 局所変数の変換

Clang が生成する LLVM IR では，入力プログラムの自動変数の値を `alloca` 命令によって確保し，`load`/`store` 命令によって値の読み書きを行うのが標準である．

LIR2LLVM も同様の方法で変数領域を管理する．しかしながら，

```
int x, y;
x = 1;
y = x;
```

のようなプログラムは次の LIR で表現される．

```
(SET I32 (MEM I32 (FRAME I64 "x.1"))
         (INTCONST I32 1))
(SET I32 (MEM I32 (FRAME I64 "y.2"))
         (MEM I32 (FRAME I64 "x.1")))
```

LIR2LLVM では，変数をメモリで管理するために，`alloca` 命令を用いて領域を確保する．ここで `alloca` 命令の戻り値は，確保した領域へのポインタ型である．LIR2LLVM による局所変数の扱いをまとめると次の通りである．

- 関数の最初に `alloca` 命令を挿入し領域の確保を行う．
- 変数が使用されるとき，`alloca` 命令で確保した領域をオペランドとする `load` 命令を挿入する．
- 変数への代入がある時，`alloca` 命令で確保した領域をオペランドとする `store` 命令を挿入する．

最終的に，先程の例で示した C プログラムは，次の LLVM IR に変換される．

```
%x.1.addr = alloca i32
%y.2.addr = alloca i32
store i32 1, i32* %x.1.addr
%0 = load i32* %x.1.addr
store i32 %0, i32* %y.2.addr
```

4. 評価と考察

4.1 評価

SPEC CPU2000 に含まれるベンチマークのうち，LIR2LLVM で動作が確認できた 5 つのベンチマークプログラム (crafty, bzip2, parser, gzip, mcf) によって実行性能を評価した．

評価環境は**表 4** に示した通りである．

各処理系の最適化オプションを default（最適化なし）から-O2 まで変化させながらベンチマークスコアを測定した．COINS の default をベースラインとしてスコアを相対値で表したものが**図 3** である．

LIR2LLVM は LIR から LLVM IR を出力するが，これを Clang に渡すことで機械語の生成を行っている．すなわち，LIR2LLVM では，COINS と Clang のそれぞれについて最適化オプションを選択できるが，評価の際には双方に同じオプションを指定するようにした．

表 4 評価環境の構成
Table 4 Evaluation environment

項目	
CPU	Intel Xeon E5-1660
メモリ	64GB
補助記憶装置	2TB HDD
COINS	1.4.6
LLVM & Clang	3.2

LIR2LLVM を COINS と比較したところ，全てのプログラムにおいて，-O1 オプション，-O2 オプションどちらの場合でも，LIR2LLVM が高い実行効率を示した．特に crafty においては，-O2 オプションのもとで約 49.8%の向上が見られた．

LIR2LLVM を Clang と比較した場合，mcf においてはほぼ差が見られなかった．crafty, bzip2, parser, gzip においては，実行効率の低下が見られた．特に parser においては，-O1 オプションで約 21.8%，-O2 オプションで約 20.2%の実行効率の低下が見られた．

4.2 考察

LIR2LLVM は，評価の限りにおいて COINS よりも高速なコードを生成した．LIR2LLVM は冗長なコード生成を行うにもかかわらず，ベンチマークスコアの傾向は Clang と似ていることが分かる．これは LLVM による最適化の効果だと考えられ，冗長なコード生成という欠点がある程度抑えられていると評価できる．

しかしながら，LIR2LLVM によるオーバーヘッドと思われるスコア低下も見られることから，単純に LIR 経由による変換が優れているとは言い切れない．

5. 関連研究

LLVM は様々な環境をターゲットとした処理系の実装が行われている．前述した PTX バックエンドは，LLVM IR から GPGPU 向けコード生成を行うものである．それ以外の試みとしては，LLVM IR を入力としてハードウェア記述言語を生成する Trident[7] がある．

GPGPU 向けコードをより効率的に生成する手法に関しては数多くの研究例・実装例がある．特に，他の中間表現を経由して GPGPU 向けコードを生成する研究としては Ikra[8] がある．Ikra 処理系は原始言語として Ruby を選択しており，目的言語として CUDA を利用することで最終的に GPGPU 向けコードを生成することができる．

COINS においても，HIR および LIR から C 言語を生成するモジュール (HIR2C および LIR2C) が実装されている．

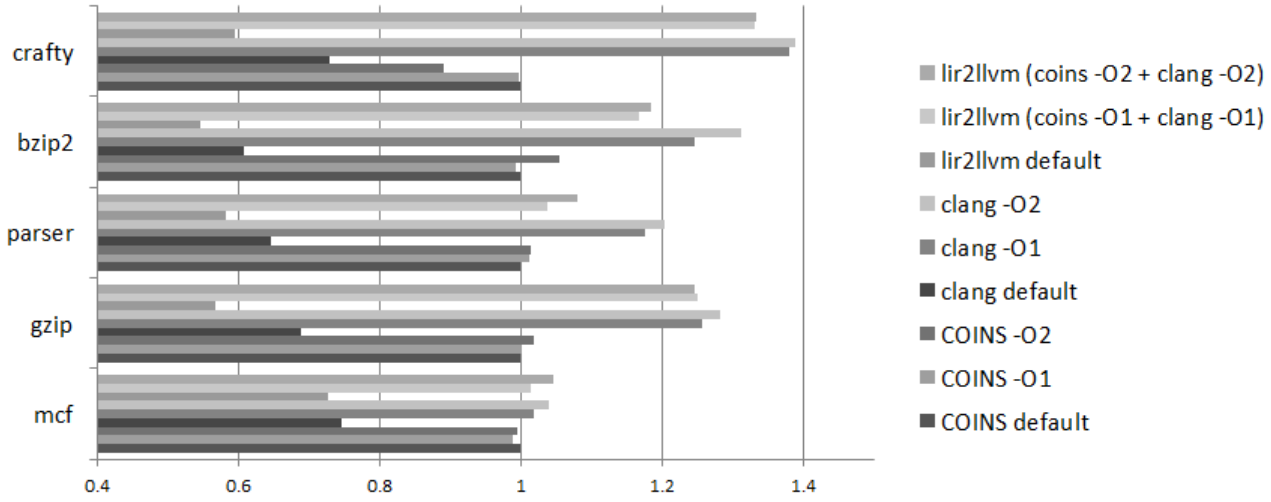

図 3 評価結果

Fig. 3 Results of benchmark(higher is better)

6. 今後の課題

今後の課題として，次のようなことが考えられる．

- より効率のよいコード生成を行うために，2 つの処理系をつなげた場合のコンパイルオプションの組み合わせについて，最適なものを調査する．
- ベンチマークス イートの一部のベンチマークプログラムでしか評価を行えて いないことから，変換モジュールの実装．をさらに検討する．
- アライメントなど の付加情報によって，生成されるコードの品質が向上 する可能性があるので，LLVM IR コードに，より詳細な型やアライ メントの情報を伝達できるようにする．

最終的に，LIR から LLVM IR を経由して GPGPU 向けのコードの生成することを目指す．

7. まとめ

本研究では LIR を経由して COINS と LLVM を接続する LIR2LLVM を実装し，SPEC ベンチマークによる評価を行った．結果として，生成したコードは Clang に近い傾向を示すことが分かった．

参考文献

[1] Lattner, C. and Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, Washington, DC, USA, IEEE Computer Society, pp. 75– (online), available from ⟨http://dl.acm.org/citation.cfm?id=977395.977673⟩ (2004).

[2] 中田育男，佐々政孝，滝本宗宏，渡邊 坦：コンパイラの基盤技術と実践 - コンパイラ・インフラストラクチャ COINS を用いて，朝倉書店 (2008).

[3] COINS プロジェクト：COINS プロジェクト LIR 仕様書，COINS コンパイラ・インフラストラクチャ協会（オンライン），入手先 ⟨http://coins-compiler.sourceforge.jp/spec/lir.pdf⟩（参照 2013/11/20）.

[4] A.V. エイホ，R. セシィ，J.D. ウルマン，M.S. ラム：コンパイラ - 原理・技法・ツール (Information & Computing)，サイエンス社，第 2 edition (2009).

[5] Appel, A. W.: 最新コンパイラ構成技法，翔泳社 (2009).

[6] 柏木餅子， 風薬：きつねさんでもわかる LLVM コンパイラを自作するためのガイドブック，インプレスジャパン (2013).

[7] Tripp, J. L., Gokhale, M. B. and Peterson, K. D.: Trident: From High-Level Language to Hardware Circuitry, *Computer*, Vol. 40, No. 3, pp. 28–37 (online), DOI: 10.1109/MC.2007.107 (2007).

[8] Masuhara, H. and Nishiguchi, Y.: A Data-parallel Extension to Ruby for GPGPU: Toward a Framework for Implementing Domain-specific Optimizations, *Proceedings of the 9th ECOOP Workshop on Reflection, AOP, and Meta-Data for Software Evolution*, RAM-SE '12, New York, NY, USA, ACM, pp. 3–6 (online), DOI: 10.1145/2237887.2237888 (2012).