Uma extensão ao Clang para identificação de laços de redução*

Vítor Resing Plentz[1], **Edevaldo Santos**[1], **Gerson Geraldo H. Cavalheiro**[1],

[1]Universidade Federal de Pelotas
Centro de Desenvolvimento Tecnológico
Programa de Pós-Graduação em Computação

{vrplentz, edevaldo.santos, gerson.cavalheiro}@inf.ufpel.edu.br

Resumo. *Na atualidade a maioria dos programas são construídos de forma sequencial, utilizando para a solução de muitos problemas o emprego de estratégias de execução iterativas, com laços de repetição. É de notório saber que grande parte do custo de processamento se dá nestes laços, não aproveitando o poder das arquiteturas paralelas. Neste artigo é apresentada uma extensão ao compilador Clang para identificação de laços paralelizáveis do padrão Reduce.*

1. Introdução

A popularização de arquiteturas multiprocessadas, pela tecnologia multicore, tornou acessível o uso de recursos de processamento paralelo em sistemas computacionais de qualquer porte. A consequência imediata é o aumento da demanda por programas que possam tirar benefício imediato do poder computacional disponível. Em outras palavras, desenvolver programas paralelos é, atualmente, uma necessidade e não mais uma opção para busca de melhora de desempenho. No entanto, nem todos os programadores possuem habilidades para identificar código que possa ser paralelizado, nem as ferramentas de programação oferecem recursos suficientes para paralelização automática de código em nível de tarefa.

Observando este fato, considera-se o interesse por uma ferramenta de identificação de padrões em programas que representem porções paralelizáveis. Neste trabalho é proposto o desenvolvimento de estratégias de análise de código fonte para identificação de tais padrões de forma a apresenta-los ao programador para que este possa reprogramar seu trecho de código na forma de um algoritmo paralelo na ferramenta de programação de sua escolha.

O caso de estudo desenvolvido neste trabalho considera o alto custo computacional gerado pelo uso de comandos iterativos (laços) nos programas e que tal recurso algorítmico ocorre com alta frequência em aplicações de diferentes domínios [Alba and Kaeli 2001, Bahi et al. 2007]. O contexto de desenvolvimento, neste momento, está restrito à identificação de padrões iterativos refletindo um modelo de execução de redução (*reduce*).

Foi desenvolvida uma extensão (*plugin*) ao compilador Clang para identificação de trechos de códigos iterativos com o padrão *reduce*. O restante deste artigo documenta o trabalho realizado e o *background* necessário para o mesmo.

*Financiado pela FAPERGS/CNPq PRONEX 16/2551-0000.

2. Background

Nesta seção são apresentados os conceitos necessários e as ferramentas escolhidas para este trabalho.

2.1. Algoritmo iterativo *reduce*

O padrão *reduce* é baseado em operações lógicas/matemáticas não associativas. A entrada neste padrão é dada por um *array*, uma variável de saída, um conjunto de operações e um espaço de iteração. Uma estrutura iterativa sobre o espaço de iteração executa o conjunto de operações apresentado sobre cada posição do *array*, armazenando o resultado de cada iteração, de forma acumulativa, sobre a variável de saída. Um exemplo de algoritmo de redução é a obtenção da soma dos valores de todos os elementos de um *array*.

2.2. Ferramental disponível com Clang

Clang [Cla] provê um front-end para compilação de linguagens C-like (C, C++, Objective C/C++, OpenCL) para a LLVM (Low Level Virtual Machine). Basicamente o Clang gera uma representação intermediária que é utilizada pela LLVM. Uma vez que o programa passa pelo Clang o mesmo já está apto para a LLVM em qualquer máquina. A opção por Clang é justificada neste trabalho por possuir possuir facilidades para o desenvolvimento de plugins customizados, usufruindo da estrutura e navegação de *AST's* (Abstract Syntax Tree), que facilitam a identificação de estruturas em um determinado código.

2.2.1. LibTooling - *Matcher* e *MatchFinder*

Entre as bibliotecas que o Clang possui suporte, se encontra o LibTooling. Esta biblioteca oferece primitivas para manipular a AST do código compilado pelo Clang. Com uso desta biblioteca, é possível realizar alterações no código fonte.

Esta biblioteca aplica o conceito de *matcher*. Um *matcher* representa uma expressão para busca por elementos específicos na AST. Os matchers podem ser de três tipos: Node Matchers, Narrowing Matchers e Traversal Matchers.

- **Node Matchers:** Podem ser definidos como o core das expressões, especificam o tipo do nó a ser buscado.
- **Narrowing Matchers:** São utilizados como suporte para os Node Matchers, correspondendo aos atributos do nodo.
- **Traversal Matchers:** Esse tipo Matcher específica a relação de um determinado nodo com outros nodos.

A classe MatchFinder::MatchCallback é utilizada em conjunto com os matchers, com essa classe é possível definir ações sobre um determinado nodo de uma AST que corresponde ao Matcher definido.

3. Metodologia

Neste trabalho foram definidos um *matcher* que corresponde ao padrão *reduce* e um MatchFinder que responde ao *matcher*.

3.1. *Matcher*

O *matcher* definido é representado pela Figura 1, e segue as seguintes características do padrão:

- Ser uma operação binária não associativa;
- Possuir como operandos um elemento de array e uma váriavel de saída;
- A operação binária deve estar dentro de um laço;
- A expressão não deve possuir mais de um operador (exceto operador de atribuição);
- O acesso ao array não deve possuir nenhuma operação.

```
57  class Finder : public MatchFinder::MatchCallback {
58  public :
59    virtual void run(const MatchFinder::MatchResult &Result) {
60        ASTContext *Context = Result.Context;
61        const BinaryOperator *BO = Result.Nodes.getNodeAs<BinaryOperator>("reduce");
62        // We do not want to convert header files!
63        if(!BO || !Context->getSourceManager().isWrittenInMainFile(BO->getOperatorLoc()))
64          return;
65        const VarDecl *IncVar = Result.Nodes.getNodeAs<VarDecl>("incVarName");
66        const VarDecl *CondVar = Result.Nodes.getNodeAs<VarDecl>("condVarName");
67        const VarDecl *InitVar = Result.Nodes.getNodeAs<VarDecl>("initVarName");
68        const VarDecl *ArrayIndex = Result.Nodes.getNodeAs<VarDecl>("arrayIndex");
69        const VarDecl *AccOperator = Result.Nodes.getNodeAs<VarDecl>("accOperator");
70        const VarDecl *Accumulator = Result.Nodes.getNodeAs<VarDecl>("accumulator");
71
72        if(!areSameVariable(IncVar, CondVar) || !areSameVariable(IncVar, InitVar)){
73          return;
74        }else if(!areSameVariable(IncVar, ArrayIndex)){
75          return;
76        }else if(!areSameVariable(AccOperator, Accumulator)){
77          return;
78        }
79      //  BO->dump();
80        llvm::outs() << "Potential Reduce pattern, in the next line you'll find the file and the suspect line.\n";
81        BO->getExprLoc().dump(Context->getSourceManager());
82        llvm::outs() << "\n";
83      }
84  };
```

Figure 1. Reduce Matcher implementado no plugin

Verificação O trabalho de verificar se as variáveis utilizadas no acesso do array (índice do array) e a váriavel que é iterada no laço são a mesma é realizado no MatchFinder, como mostrado na Figura 2. No caso do padrão ser reconhecido o usuário é notificado.

```
class Finder : public MatchFinder::MatchCallback {
public :
  virtual void run(const MatchFinder::MatchResult &Result) {
      ASTContext *Context = Result.Context;
      const BinaryOperator *BO = Result.Nodes.getNodeAs<BinaryOperator>("reduce");
      // We do not want to convert header files!
      if(!BO || !Context->getSourceManager().isWrittenInMainFile(BO->getOperatorLoc()))
        return;
      const VarDecl *IncVar = Result.Nodes.getNodeAs<VarDecl>("incVarName");
      const VarDecl *CondVar = Result.Nodes.getNodeAs<VarDecl>("condVarName");
      const VarDecl *InitVar = Result.Nodes.getNodeAs<VarDecl>("initVarName");
      const VarDecl *ArrayIndex = Result.Nodes.getNodeAs<VarDecl>("arrayIndex");
      const VarDecl *AccOperator = Result.Nodes.getNodeAs<VarDecl>("accOperator");
      const VarDecl *Accumulator = Result.Nodes.getNodeAs<VarDecl>("accumulator");

      if(!areSameVariable(IncVar, CondVar) || !areSameVariable(IncVar, InitVar)){
        return;
      }else if(!areSameVariable(IncVar, ArrayIndex)){
        return;
      }else if(!areSameVariable(AccOperator, Accumulator)){
        return;
      }
    //  BO->dump();
      llvm::outs() << "Potential Reduce pattern, in the next line you'll find the file and the suspect line.\n";
      BO->getExprLoc().dump(Context->getSourceManager());
      llvm::outs() << "\n";
    }
};
```

Figure 2. MatchFinder implementado no plugin

4. Resultados Obtidos

Com a utilização do plugin criado, dado um código de entrada (Figura 3) nas linguagens suportadas pelo Clang que possuem algum trecho de código com o padrão *reduce*, a saída do plugin (Figura 4) exibe no terminal a linha onde o padrão foi identificado e que pode ser paralelizado.

```
1  #include <iostream>
2
3  #define ARRAYSIZE 10
4
5
6  int main(){
7      int array[ARRAYSIZE] = {0, 1 , 2, 3, 4, 5, 6, 7, 8, 9};
8      int sum;
9      for(int i = 0; i < ARRAYSIZE; i++){
10         sum = sum + array[i]; //caracteriza o padrão reduce
11     }
12     for(int i = 0; i < ARRAYSIZE; i++){
13         sum = sum + array[i+1]; //dependência de dados
14     }
15     int j = 0;
16     for(int i = 0; i < ARRAYSIZE; i++){
17         sum = sum + array[j]; //não caracteriza o padrão reduce definido
18     }
19
20     for(int i = 0; i < ARRAYSIZE; i++){
21         sum = sum / array[i]; //operação associatiava
22     }
23
24     for(int i = 0; i < ARRAYSIZE; i++){
25         sum  = sum + sum; // não envolve o vetor
26     }
27
28     for(int i = 0; i < ARRAYSIZE; i++){
29         sum = sum + array[i] + sum; //não caracteriza o padrão reduce definido
30     }
31     return 0;
32 }
```

Figure 3. Exemplo de teste com código em C++ realizado no plugin

```
vplentz@vplentz-Inspiron-7460:~/clang-llvm/build$ ./bin/reduction tests/test_red.cpp --
Potential Reduce pattern, in the next line you'll find the file and the suspect line.
/home/vplentz/clang-llvm/build/tests/test_red.cpp:10:13
```

Figure 4. Saída do plugin, utilizando o arquivo de teste exibido acima.

5. Conclusão

Programar de forma sequencial apesar de muitas vezes ficar em desvantagem quanto ao desempenho da programação paralela, também possui características boas, como a fácil legibilidade do código e a agilidade para a criação do mesmo. Com o tipo de ferramenta proposta neste artigo, é possível aproximar os dois estilos de programação, desta forma o programador pode programar sequencialmente, usufruir de seus benefícios e após essa etapa, utilizar o plugin para melhorar o desempenho da aplicação.

References

Clang: a C language family frontend for LLVM. `https://clang.llvm.org/`. Acessado em: 07/01/2018.

Alba, M. d. R. and Kaeli, D. R. (2001). Runtime predictability of loops. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 91–98.

Bahi, J. M., Contassot-Vivier, S., and Couturier, R. (2007). *Parallel Iterative Algorithms: From Sequential to Grid Computing (Chapman & Hall/Crc Numerical Analy & Scient Comp. Series)*. Chapman & Hall/CRC.