AVALIAÇÃO DE COMPILAÇÃO VETORIZADA

Nilzete O. Álvares
Depto. de Estatística e Informática
Univ. Fed. de Goiás - Campus 2 - Goiânia - Goiás - 74000

Leila M. R. Eizirik e Claudio L. Amorim
Programa de Sistemas e Computação - COPPE/UFRJ
Cx. Postal 68511 - Ilha do Fundão - Rio de Janeiro - RJ - 21945

RESUMO

As idéias e algoritmos básicos propostos por Fischer e Donegan são avaliados através da implementação de um compilador. Os resultados experimentais obtidos confirmam a hipótese de que a compilação é maciçamente vetorizável e revelam que apenas dezessete instruções vetoriais são responsáveis por 90% da vetorização. Neste trabalho descrevemos o processo de compilação vetorizada e a metodologia de avaliação. Os resultados obtidos são analisados e conclusões são extraídas.

ABSTRACT

The ideas and basic algorithms which have been proposed by Fischer and Donegan to vectorize an arithmetic expression compiler are evaluated through its implementation. Our experimental results confirm the hypothesis that the compiler process is massively vectorized. Furthermore we find that only seventeen vector instructions are responsible for 90% of the vectorization. In this work we describe the process of vectorization and the evaluation methodology used. The results are analysed and conclusions are drawn.

1. Introdução.

O desempenho efetivo das máquinas vetoriais "pipelined" tais como CRAY-XMP, FUJITSU VP-200 e CYBER 205 depende do grau de vetorização do código gerado [1]. Praticamente todas as pesquisas têm sido orientadas para compiladores Fortran vetorizadores [2] . Tipicamente, esses compiladores analisam laços Fortran de um programa contendo seqüências de instruções escalares para convertê-las em seqüências equivalentes de instruções vetoriais e escalares, procurando atingir o máximo de vetorização.

Outro aspecto igualmente importante, porém pouco investigado é o de se explorar a vetorização do processo de compilação. Isso porque uma parte significativa do tempo dessas máquinas é gasta na compilação de programas. Estima-se que as referidas máquinas usem 50% de seu tempo na compilação.

Neste trabalho descrevemos a implementação de um compilador para expressões aritméticas de forma a avaliar as idéias e algoritmos básicos propostos por Fischer [3] e Donegan [4] para vetorizar a compilação.

Na seção 2 discutimos os principais algoritmos e suas implementações. Na seção seguinte, a metodologia de avaliação e os resultados obtidos são apresentados. Finalmente, na seção 4 conclusões são obtidas.

2. Descrição do Compilador.

O compilador desenvolvido reconhece comandos de atribuição do Fortran tais que o lado esquerdo deve ser um identificador simples e o lado direito uma expressão aritmética válida em Fortran. Não é permitido o uso de funções nem de "arrays" de mais de uma dimensão.

O compilador para expressões aritméticas foi implementado utilizando os algoritmos de Donegan para análise léxica e os algoritmos de Fischer para análise sintática, semântica e geração de código.

O método de análise sintática é baseado numa função de precedência dos operadores. Para construção desta função de precedência Fischer utiliza um tipo de gramática chamada AIG (Arithmetic Infix Grammar) definida a seguir:

Seja $OP_0, \ldots, OP_n$ um conjunto de classes disjuntas de operadores tais que os operadores Op_i têm prioridade maior do que os operadores de $Op_{i-1}, i = 1, \ldots, n$. Para cada classe de operadores OP_i, define-se:

$$KIND(i) = \begin{cases} BIN & \text{, se os operadores de } OP_i \text{ forem binários.} \\ UN & \text{, se os operadores de } OP_i \text{ forem unários.} \end{cases}$$

$$ASSOC(i) = \begin{cases} -1 & \text{, se os operadores de } OP_i \text{ forem associativos à esquerda.} \\ 1 & \text{, se os operadores de } OP_i \text{ forem associativos à direita.} \end{cases}$$

A classe OP_0 contém um único operador pré-definido #. # é binário, associativo à esquerda e é usado como separador.

Uma gramática AIG, segundo a definição usada por Fisher, é uma gramática livre do contexto $G = (V, V_t, P, S)$ onde:

$$V_t = \{id, (,)\} \cup OP_0 \cup \ldots \cup OP_n,$$

$$V = V_t \cup \{S, N_0, \ldots, N_{n+1}\}$$

e o conjunto de produções P é dado pela união dos conjuntos:

1. $\{S \rightarrow \#N_0\#\}$
2. $\{N_i \rightarrow N_{i+1} \mid 0 \leq i \leq n\}$
3. $\{N_{n+1} \rightarrow id, N_{n+1} \rightarrow (N_1)\}$
4. Para $0 \leq i \leq n$

   Se $KIND(i) = UN$ então
   $\{N_i \rightarrow \Theta N_i \mid \Theta \in OP_i\}$
   senão
   se $ASSOC(i) = -1$ então
   $\{N_i \rightarrow N_i \Theta N_{i+1} \mid \Theta \in OP_i\}$
   senão
   $\{N_i \rightarrow N_{i+1} \Theta N_i \mid \Theta \in OP_i\}$

A linguagem reconhecida pelo compilador implementado é dada por uma gramática AIG com algumas modificações para inclusão do comando de atribuição como um operador especial.

As classes dos operadores são as seguintes:

	OP	KIND	ASSOC
0	#	BIN	−1
1	=	BIN	1
2	+,−	BIN	−1
3	*,/	BIN	−1
4	ident, neg	UN	1
5	**	BIN	1
6	index	BIN	1

onde os operadores **ident** e **neg** correspondem internamente aos operadores unários +, −. O operador index é usado na referência a "arrays", isto é, A(B) é transformado em A index B durante a análise léxica.

As regras de produção são as seguintes:

$S \rightarrow \#N_0\#$ $\quad$ $N_7 \rightarrow id$
$N_0 \rightarrow N_1$ $\quad$ $N_7 \rightarrow (N_2)$
$N_1 \rightarrow id = N_2$ $\quad$ $N_0 \rightarrow N_0 \# N_1$
$N_2 \rightarrow N_3$ $\quad$ $N_2 \rightarrow N_2 + N_3 \mid N_2 - N_3$
$N_3 \rightarrow N_4$ $\quad$ $N_3 \rightarrow N_3 * N_4 \mid N_3 / N_4$
$N_4 \rightarrow N_5$ $\quad$ $N_4 \rightarrow ident\ N_4 \mid neg\ N_4$
$N_5 \rightarrow N_6$ $\quad$ $N_5 \rightarrow N_6 ** N_5$
$N_6 \rightarrow N_7$ $\quad$ $N_6 \rightarrow N_7 index N_6$

2.1. Análise Léxica.

O módulo LÉXICO executa a análise léxica vetorizada dos comandos aritméticos A entrada é o vetor de caracteres VECTOR contendo um ou mais comandos de atribuição de expressões aritméticas separadas pelo delimitador #. A saída é o vetor INPUT com os códigos dos "tokens" identificados.
Inicialmente é verificada a validade dos caracteres que compõem VECTOR e os caracteres brancos são retirados. A seguir obtém-se um novo vetor VECTOR1 cujos elementos são a concatenação dos caracteres de VECTOR que formam cada "token". Finalmente, constrói-se o vetor de inteiros INPUT que contém os códigos para identificadores e constantes nas posições correspondentes aos mesmos em VECTOR1.

2.2. Análise Sintática.

A análise sintática é realizada em duas partes:

- a primeira testa se a entrada é sintaticamente válida (módulo BEM-FORMADA).
- a segunda constrói a árvore de sintaxe reduzida correspondente à entrada (módulo PARSER).

2.2.1. Módulo BEM-FORMADA.

O módulo BEM-FORMADA testa se a cadeia de "tokens" fornecida pelo módulo léxico é uma sentença derivável na gramática AIG definida anteriormente.

A entrada para o módulo é o vetor INPUT obtido do módulo LÉXICO e a saída é um valor booleano (1 se for bem formada e 0 caso contrário).

Para verificar se a cadeia de "tokens" de INPUT é válida são testadas as seis condições abaixo:

1. O primeiro e o último elementos de INPUT devem ser o delimitador #.
2. Toda e qualquer subcadeia da entrada deve ser tal que o número de fecha parênteses deve ser no máximo igual ao número de abre parênteses.
3. Entre dois delimitadores # consecutivos existe exatamente um operador de atribuição "=" e o número de abre parênteses deve ser igual ao número de fecha parênteses.
4. Para quaisquer dois elementos consecutivos o_1 e o_2 do vetor INPUT:
   - $o_2 \in FOLLOW(o_1)$
   - se o_2 é unário e o_1 é um operador qualquer então $CLASS(o_1) \leq CLASS(o_2)$

2.2.2. Módulo PARSER.

Para gerar a árvore de sintaxe reduzida, o módulo PARSER utiliza uma função de precedência PREC. O valor de PREC é calculado para cada operador, identificador e constante que ocorre na entrada. A função PREC define a ordem de aplicação dos operadores e depende dos seguintes fatores:

- Nível de aninhamento de parênteses
- Prioridade do operador
- Associatividade do operador
- Posição no vetor de entrada

Seja IDOPS o vetor obtido de INPUT retirando os parênteses. IDOPS é constituído dos identificadores, constantes e operadores que ocorrem na entrada.

Definimos um vetor de inteiros associado à IDOPS:

$$PREC(IDOPS) = \\ = POS(IDOPS) * ASSOC(IDOPS) + \\ +2 * (q+1) * CLASS(IDOPS) + \\ +2 * (q+1) * (q+2) * NEST(IDOPS)$$

onde:

$POS(IDOPS) = 1,\ldots,q$; é o vetor das posições de cada elemento de IDOPS.

$CLASS(IDOPS)$ é o vetor com a classe correspondente a cada elemento de IDOPS.

$CLASS(\Theta) = j \Leftrightarrow \Theta \in OP_j$

$CLASS(id) = n+1, CLASS(\text{"("}) = CLASS(\text{")"}) = n+2$, n é o número de classes de operadores.

$NEST(IDOPS)$ é o vetor com o nível de aninhamento dos elementos de IDOPS em relação aos parênteses.

O módulo PARSER recebe como entrada o vetor INPUT e retorna três vetores EXPROOTS, LSUBTREE e RSUBTREE que representam a árvore de sintaxe abstrata.

Inicialmente, o módulo BEM-FORMADA é chamado para verificar a validade da entrada. No caso da entrada ser válida cria-se o vetor PREC conforme definido acima.

A seguir constrói-se o vetor EXPROOTS constituído das raízes das árvores que representam cada comando de atribuição. Se a raiz do i-ésimo comando for IDOPS(j) então $EXPROOTS(i) = j$.

Utilizando o vetor de precedência PREC, cria-se o vetor LSUBTREE constituído das raízes das sub-árvores à esquerda. Se i é tal que $IDOPS(i) \neq \#$ então LSUBTREE(i) recebe o índice da raiz da sub-árvore esquerda de IDOPS(i) ou 0 se não existir esta sub-árvore. Analogamente obtém-se RSUBTREE com as raízes das sub-árvores direitas.

2.3. Análise Semântica e Geração de Código.

Para realizar a análise semântica e a geração de código é necessário percorrer a árvore de sintaxe reduzida (EXPROOTS, LSUBTREE, RSUBTREE) fornecida pelo módulo PARSER.

Fischer propõe duas formas vetorizadas de percorrer a árvore, chamadas iteradores TOPDOWN e BOTTOMUP. Estes iteradores são rotinas que percorrem a árvore

em camadas nos sentidos "topdown" e "bottomup", respectivamente.

Por exemplo, no percurso "topdown" são visitados simultaneamente todos os nodos que correspondem às raízes das expressões analisadas seguidas de todos os seus filhos esquerdos e depois de todos os seus filhos direitos e assim, sucessivamente.

2.3.1. Teste de Consistência de Tipos.

Estes testes são feitos percorrendo-se a árvore no sentido "bottom-up". Os operadores são divididos em três classes: aritméticos, indexação e atribuição.

Para cada camada da árvore que é fornecida pelo iterador é chamada a rotina COMPUTYPE, resultando na construção do vetor RESTYPE. Este vetor contém os tipos associados aos operadores ou um código de erro no caso de inconsistência de tipos.

COMPUTYPE analisa inicialmente os operadores aritméticos. O tipo do operador é determinado pelo tipo dos operandos, escolhendo-se o tipo mais abrangente. A seguir, analogamente, é testada a consistência de tipos na atribuição. Finalmente, para a indexação é verificado se o índice é do tipo inteiro.

2.3.2. Alocação de Temporárias.

No trabalho de Fischer são apresentados dois algoritmos para alocação de temporárias: SEU (Single Execution Unit) e MEU (Multiple Execution Unit).

No modelo SEU as operações são executadas sequencialmente. No modelo MEU mais de uma operação pode estar em curso ao mesmo tempo e assume-se um número ilimitado de unidades de execução.

Neste compilador foi implementado o algoritmo MEU com as adaptações necessárias à arquitetura do CRAY-1.

A rotina de alocação de temporárias aloca uma nova temporária para cada operador que tenha somente identificadores como operandos, de tal maneira que essas operações possam ser executadas concorrentemente.

Se um operador tem um filho que não é um identificador então este operador herda a temporária do filho.

A alocação é feita usando o iterador BOTTOM-UP e resulta no vetor RESTEMP que contém a temporária associada a cada nó da árvore.

2.3.3. Geração de Código.

A geração de código é feita pela rotina COMPUTESIZES.

Para cada camada da árvore fornecida pelo iterador BOTTOMUP, a rotina COMPUTESIZES chama a rotina COMPUTINST.

COMPUTINST determina o conjunto de instruções que deve ser gerado para cada nodo da árvore e monta um vetor com as instruções que compõem o código objeto associado.

A seguir COMPUTESIZES monta o vetor EXPRSIZES que contém os comprimentos em número de instruções do código associado a cada expressão analisada. Finalmente, é montado o código objeto correspondente às expressões analisadas.

3. Avaliação do Compilador.

3.1. Definição das Funções Vetoriais.

A máxima vetorização dos algoritmos foi obtida baseada na utilização de quarenta e seis funções vetoriais distribuídas nas sete classes abaixo. Observamos que várias dessas funções são equivalentes, porém foram criadas para manipular três diferentes tipos de argumentos encontrados nos algoritmos: inteiro, lógico e caracter. Funções distintas para operações envolvendo dois vetores ou um vetor e um escalar foram criadas. Mostramos a seguir as classes de funções (total de funções assinaladas) e alguns exemplos. Temos:

n:	comprimento do vetor,
e:	escalar inteiro,
b:	escalar booleano,
$A1_n$, $A2_n$, $A3_n$:	vetores de caracteres,
$B1_n$, $B2_n$:	vetores de bits (lógicos),
X_n, Y_n, Z_n:	vetores de inteiros.

classe i – Funções Aritméticas [13]

Incluem operações sobre vetores de elementos inteiros tais como soma, subtração, multiplicação, divisão, módulo, máximo, mínimo e delta. Exemplos:

Soma: $Z_n := \sum_{i=1}^{n} X_i + Y_i$

Máximo: e := $MAX(X_n)$

Delta: $Z_n := X_{n+1} - X_n$

classe ii - Funções Lógicas e Funções de Deslocamento [8]

As funções lógicas operam sobre vetores de bits. As funções de deslocamento operam sobre vetores de bits, vetores de inteiros ou vetores de caracteres. Exemplos:

ALL-ONES: **if** $all(B1_n) =' 1'$ **then** *true* **else** *false*,

SHIFT-ALFA: $A2_n := A1_{n-1}$,

SHIFT-INT: $Z_n := X_{n-1}$

classe iii - Funções de Comparação [2]

Testam a igualdade entre vetores de caracteres ou entre vetores inteiros. Temos:

COMP. EQUAL-ALFA: **if** $A1_n = A2_n$ **then** $B1_n := true$ **else** $B1_n := false$

COMP. EQUAL-INT : **if** $X_n = Y_n$ **then** $B1_n := true$ **else** $B1_n := false$

classe iv - Funções de Mascaramento [4]

Permitem testar cada elemento de um vetor de inteiros segundo determinada condição ($>$, $<$, $=$, $\neq$) e produzir o vetor máscara correspondente. Por exemplo:

MASKN: **if** $X_n < 0$ **then** $B1_n := true$ **else** $B1_n := false$

classe v - Funções "Merge" e Funções de Concatenação [5]

Implementam operações de "merge" sobre vetores de inteiros, vetores de caracteres ou vetores lógicos. Exemplos:

MERGE-INT: **if** $B1_n$ **then** $i := i + 1$; $Z_n := X[i]$ **else** $i := i + 1$; $Z_n := Y[i]$

CONCATENA: $A3_n := A1_n.A2_n$

classe vi - Funções Agrup./Espalham./Compressão [6]

Operam sobre elementos inteiros, lógicos ou caracteres. Exemplos:

COMPRIMA-ALFA: **if** $B1_n$ **then** $i := i + 1$; $A2[i] = A1_n$,

ESPALHA-INTEIRO: $Z_n[X_n] := Y_n$,

AGRUPA-LÓGICA: $B1_n := B2_n[X_n]$

classe vii - Funções de Atribuição e de Busca [8]

Servem para atribuir valor inteiro, lógico ou caracter aos elementos dos vetores. Exemplos:

REPETE (LÓGICA): $B2[i : i + e] :=$ b,

GERAVETOR(INTEIRO): $Z[i : j] := e_1 : e_2 : e$,

ATRIBUA(ALFA): $Z_n := X_n$

3.2. Testes e Resultados.

As expressões aritméticas contidas nos laços do "Lawrence Livermore Loops"[5] foram extraídas aleatoriamente e utilizadas para avaliar o grau de vetorização alcançado pelo compilador. Mostramos na tabela 1 os percentuais do tempo de compilação em modo escalar e vetorial em função do comprimento total das expressões aritméticas ('string' aritmético) e do número de variáveis envolvidas. Por exemplo, a expressão número 1 corresponde à $A = B * -2$.

Teste no.	No. de variáveis	Tamanho total das expressões	Modo de Processamento		Distribuição do tempo de processamento vetorial na compilação		
			%Escalar	%Vetorial	%Léxica	%Sintática	%Semântica
1	2	8	68.0	32.0	17.6	52.8	38.3
2	6	83	51.6	48.4	40.7	74.4	46.2
3	10	98	50.9	49.1	45.3	73.9	43.6
4	13	159	40.7	59.3	56.7	82.5	50.1
5	12	164	41.5	58.5	56.4	82.1	50.4
6	16	424	30.8	69.2	73.0	87.2	55.4
7	33	660	26.5	73.5	76.6	90.5	58.6
8	76	985	24.1	75.9	78.5	89.6	62.7
9	33	1360	18.9	81.1	83.2	93.5	66.5
10	76	2035	17.0	83.0	84.1	93.2	72.1

Tabela 1 - Percentual de vetorização do compilador

Considerando que o compilador foi testado numa

máquina escalar, o tempo de processamento em modo vetorial foi estimado somando-se os tempos gastos na execução das funções vetoriais.

Os percentuais a que se referem as colunas %léxica, %sintática e %semântica expressam as frações correspondentes de tempo gasto em modo de processamento vetorial (coluna %vetorial).

Como pode ser visto na tabela 1 a medida que o comprimento do 'string' aritmético aumenta, o tempo de vetorização também aumenta (entre 32% e 83%), como esperado. O percentual escalar corresponde ao tempo de controle gasto para a preparação e as chamadas das funções, já que praticamente todos os algoritmos de análise léxica, sintática e semântica são altamente vetorizáveis. Pelos resultados mostrados podemos esperar que programas dominados por laços extensos de expressões aritméticas, como são em grande parte as aplicações numéricas científicas, o tempo da compilação seja potencialmente alto e portanto, atrativo para vetorização.

FUNÇAO	CL	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
VXTOV	vi	27.7	27.5	29.9	25.6	25.9	21.1	21.8	17.8	20.4	16.1
GERAVETOR	vii	3.4	6.6	5.8	9.5	9.4	14.5	13.4	13.7	15.6	13.3
CPSV	vi	13.2	13.5	12.6	13.5	13.3	13.7	13.1	12.1	13.5	11.5
MASKZ	iv	7.7	7.7	7.5	7.2	7.2	6.7	6.7	6.1	6.3	5.7
SUBVS	i	7.5	7.4	6.7	6.3	6.3	5.5	5.6	5.0	5.4	4.8
MASKNZ	iv	4.1	4.2	4.2	4.2	4.0	4.0	3.9	3.6	3.9	3.3
REPETE	vii	6.2	5.8	5.7	5.1	5.3	4.5	4.4	3.9	3.8	3.1
VTDVX	vi	5.0	4.4	4.2	3.8	4.0	3.1	3.2	2.7	3.3	3.0
MRGV	v	2.0	2.4	2.4	2.3	2.4	2.5	2.6	2.4	3.0	2.4
ALL-ONES	ii	1.6	1.8	2.7	2.4	2.4	3.9	2.9	2.9	2.8	2.8
MASRP	iv	2.4	2.4	2.0	2.1	2.1	2.0	2.0	1.9	2.2	2.1
ADVV	i	0.9	0.8	0.6	1.3	1.2	1.4	2.5	5.4	2.7	7.5
ASSIGN	vii	0.6	0.5	0.2	1.0	0.9	1.0	2.1	4.9	2.3	6.9
TOTAL (13)		82.3	85.2	84.5	84.3	84.4	83.9	84.2	82.4	85.2	82.5
SHIFTIR	ii	0.3	0.2	0.0	0.8	0.6	0.9	1.8	4.8	2.0	6.7
NOTT	ii	0.8	0.8	1.5	1.8	1.5	3.2	2.2	2.3	2.3	2.1
REPEAT	vii	2.6	1.9	2.0	1.7	1.8	1.2	1.2	1.0	1.0	0.6
VXTOVB	vi	2.4	2.1	2.4	1.9	2.1	1.7	1.6	1.4	1.3	1.1
TOTAL (17)		88.4	90.0	90.4	90.5	90.4	90.9	91.0	91.9	91.8	93.0

Tabela 2 - Percentual de tempo das funções.

Mostramos na tabela 2 as dezessete funções dentre o total de quarenta e seis funções que surpreendentemente são responsáveis por cerca de 90% da compilação vetorizada. Agrupando as dezessete funções da tabela 2 segundo as classes (CL) definidas anteriormente, temos:

CL	Classes de Funções	No.	% da Vetorização (≅)
vi	Agrupamento/Espalhamento/Compressão	4	40
vii	Atribuição e Busca	4	20
iv	Mascaramento	3	14
i	Aritméticas	2	10
ii	Lógicas e Deslocamentos	3	7
v	"Merge" e Concatenação	1	2
iii	Comparação	0	0

4. Conclusões.

Os resultados experimentais até aqui obtidos confirmam a hipótese inicial de que a compilação de expressões aritméticas é maciçamente vetorizável com o modo vetorial de processamento dominando o modo escalar em praticamente todos os testes. Um achado importante foi que apenas dezessete funções entre as quarenta e seis, são responsáveis por 90% do tempo gasto na vetorização.

Torna-se necessário verificar se essas funções "virtuais" possuem equivalentes instruções nas máquinas vetoriais atuais de forma que a potencial vetorização se torne efetiva, ou seja, que a compilação vetorizada obtenha um melhor desempenho do que uma compilação convencional. Nesse sentido, as primeiras constatações são promissoras. Podemos observar que as máquinas vetoriais recentes possuem as classes de funções mencionadas, embora em cada classe a variedade de funções disponíveis depende do fabricante. Por exemplo, as instruções vetoriais críticas Agrupar/Espalhar/Comprimir implementadas originalmente no CDC STAR 100 e mantidas nos sucessores modelos CYBER 205 e ETA-10, não eram disponíveis no CRAY-1 porém, hoje, se encontram nos modelos CRAY XMP-48 e CRAY-2. Nas máquinas japonesas posteriormente lançadas, tais como FUJITSU VP-200 e NEC SX-2 que dispõem de compiladores vetorizadores eficientes, todas possuem não só as referidas instruções, como também, por exemplo, poderosas instruções vetoriais de mascaramento.

Para estimar o desempenho da compilação vetorizada [6], as funções acima estão sendo traduzidas e submetidas a um simulador do CRAY-1. Os tempos produzidos pelo simulador permitirão estimar a redução do tempo de compilação utilizando um conjunto de instruções vetoriais disponível e assim obter uma medida do potencial efetivo da compilação vetorizada.

Agradecimentos

À Inês, nossa aluna, pela pronta ajuda na produção do artigo, em Latex.

Esta pesquisa foi parcialmente financiada pela FINEP e CAPES.

Referências

[1] K. Hwang e F. A. Briggs, *"Computer Architecture and Parallel Processing"*, McGraw-Hill Book Company, (1984).

[2] D. A. Padua e M. J. Wolfe, *"Advanced Compiler Optimizations for Supercomputers"*, Communications of the ACM, 29, no. 12, pp. 1184-1201, Dez. (1986).

[3] C. N. Fischer, *"On Parsing and Compiling Arithmetic Expressions on Vector Computers"*, ACM Transactions on Programming Languages and Systems, Vol. 2, no. 2, pp. 203-224, Abril (1984).

[4] M. K. Donegan e S. W. Katzke, *"Lexical Analysis and Parsing Techniques for a Vector Machine"*, Proceedings of Conference on Programming Languages and Compilers for Parallel and Vector Machines, SIGPLAN Notices, vol. 10, no. 3, pp. 138-145, Março (1975).

[5] J. P. Riganati e P. B. Schneck, *"Supercomputing"*, IEEE Computer, pp. 97-112, Outubro (1984).

[6] N. O. Alvares e outros, *"Um Compilador Vetorizado para Máquinas Vetoriais"*, XIII SEMISH e VI Congresso da SBC, Recife, pp. 616-620, Julho (1986).