Loop Restructuring e Vetorização

Bruno Cardoso Lopes
RA 023241
Instituto de Computação, Unicamp
Campinas, São Paulo
bruno.cardoso@gmail.com

ABSTRACT

Compiladores que geram código vetorial existem desde os antigos computadores Cray. A abordagem de vetorização é recorrente hoje em dia com as arquiteturas modernas e seus conjuntos de extensões multimídia. Este artigo introduz as arquiteturas vetoriais e em seguida apresenta um histórico de diversas técnicas de re-escrita de loops como base para a apresentação das técnicas de vetorização utilizadas na área de compiladores nos últimos 10 anos; vetorizações em laços e geração de SIMDs através da detecção de *Superword Level Paralelism* (SLPs).

General Terms

Compilers architecture parallelism

1. INTRODUCTION

Transformações em loops fazem parte das principais áreas de pesquisa de otimização em compiladores, e também dos principais enfoques da área de vetorização de código. As técnicas de vetorização são aplicadas como passos de transformações em compiladores e tentam descobrir operações vetoriais automaticamente. A aplicação destas técnicas só é possivel através das informações obtidas com as transformações nos loops.

Processadores vetoriais utilizam instruções SIMD (Single Instruction, Multiple Data) operando em vários fluxos de dados com uma única instrução. Assim, é possível com uma única instrução realizar operações sobre todos elementos de um vetor de uma única vez, como um incremento de um em todos os elementos desse vetor de maneira atômica.

Afim de suportar operandos fonte e destino vetoriais, estes processadores possuem registradores vetoriais, com tamanhos dependentes de implementação. O uso adequado desses registradores leva a ótimos desempenhos.

Desde 1994 processadores de propósito geral vêm sendo fabricados com extensões de multimídia. O pioneiro foi o MAX-1 fabricado pela HP, e em 1997 chegaram massivamente no mercado com o lançamento do Pentium MMX, o primeiro processador da Intel com extensões de multimídia.

Processadores vetoriais possuem grande similaridade com as extensões multimídia dos processadores de propósito geral - ambos conjuntos de instruções são SIMD. Por causa dessa semelhança, os mecanismos tradicionais de vetorização podem também ser aplicados a estas extensões.

Extensões de multimídia são os conjunto de operações vetoriais encontrados nos dias de hoje, portanto, todas as referências a códigos vetoriais serão feitas com este foco.

2. CÓDIGO VETORIAL

Instruções vetoriais geralmente possuem dois operandos vetoriais, ou um operando vetorial e um escalar. Um vetor pode ser definido pela posição de início, a distância (passo ou *stride*) entre os elementos do vetor e tamanho do vetor (em número de elementos). As duas primeiras propriedades estão geralmente implícitas nos próprios registradores, e o tamanho em algum registrador especial.

2.1 Motivação

Algumas operações utilizam muito menos bits do que os disponíveis em hardware. Operações com cores representam bem o espaço de operações realizadas com menos bits do que o necessário, supondo que um canal de cor (RGB) ocupa 8 bits:

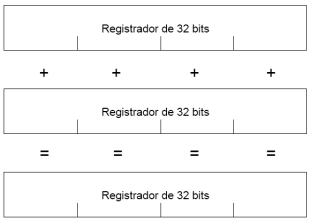

Figure 1: Soma Vetorial

Soma A soma de dois canais, utiliza apenas 8 bits das fontes e destino, não necessitando do tamanho inteiro da palavra (figura 1).

Saturação Incremento de um em um canal com valor 255, permanece 255 se a soma for saturada.

Alem da subutilização de bits, em um caso comum é necessário mais do que uma instrução para realizar operações sobre um conjunto de pixels, sendo ainda pior se for necessário acesso freqüente a memória para realizar estas operações.

Uma maneira eficiente em hardware de realizar essas operações utiliza vetorização. Se cada canal for mapeado para uma posição de um registrador vetorial, é possível realizar somas saturadas entre registradores vetoriais, e efeitos de overflow são considerados apenas para cada canal. Assim, é possível realizar operações sobre vários conjuntos de pixels com apenas uma instrução.

Vários outros tipos de operações vetoriais também podem ser realizadas otimizando a execução em hardware, operações horizontais e empacotamento são duas bastante comuns :

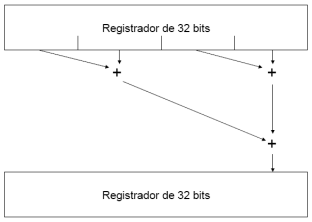

Figure 2: Soma Horizontal

Horizontalidade Realiza a soma de todos os elementos de um vetor e armazenada o resultado em um escalar. (figura 2).

Empacotamento Converte dados da memória ou escalares para o formato vetorial - no exemplo de cores, cada canal RGB do pixel pode ser colocado em uma posição diferente do vetor.

2.2 Conjuntos de Instruções

Vários fabricantes acoplam extensões de multimídia no seus conjuntos de instruções. A tabela 1 mostra algumas das várias arquiteturas e suas extensões multimídia.

intel x86	MMX, SSE/2/3, SSSE3, SSE4.1/4.2 AVX[1]
amd x86	3DNow, SSE/2/3, SSSE3, SSE4a, SSE5[2]
cell	SPU ISA
mips	MIPS-3D ASE
powerpc	Altivec

Table 1: Características da Série 8

3. VETORIZAÇÃO MANUAL

A maneira mais comum de aplicação de vetorização em códigos é através da utilização explícita de código vetorial. Isto pode ser feito de 2 formas:

1. Inserção da codificação de rotinas ou trechos de código em assembly. Estas rotinas são invocadas através de linguagens como C através de símbolos definidos externamente, de maneira que em tempo de link edição, o endereço real destas rotinas é conhecido. Outra abordagem é com a utilização de inline assembly diretamente dentro de rotinas em C.

2. Utilização de intrinsics do compilador. O Compilador pode disponibilizar funções intrinsics genéricas que são convertidas para código vetorial de qualquer arquitetura (desde que a mesma possua extensões multimídia), ou disponibiliza intrinsics específicos para cada arquitetura.

[1]Advanced Vector Extensions, fazerá parte do novo processador de 32nm Sandy Bridge, com previsão de lançamento em 2010

[2]Previsto para 2010 no processador bulldozer

Em ambas abordagens, o compilador deve ter um suporte básico as instruções vetoriais. O trecho de código 1 exemplifica a utilização de intrinsics.

Trecho 1 Exemplo de utilização de intrinsics

```
#define ALIGN16 __attribute__((aligned(16)))

__m128 a, b, c;
float inp_sse1[4] ALIGN16 = { 1.2, 3.5, 1.7, 2.8 };
float inp_sse2[4] ALIGN16 = { -0.7, 2.6, 3.3, -4.0 };
...
a = _mm_load_ps(inp_sse1);
b = _mm_load_ps(inp_sse2);
c = _mm_add_ps(a, b);
```

4. VETORIZAÇÃO AUTOMÁTICA

A maneira mais conhecida e pesquisada de vetorizar aplicações é através da vetorização de laços[4]. Muitas vezes os laços apresentam oportunidades ideais onde o código vetorial pode substituir uma laço inteiro. Mas antes de chegar nos laços ideais, é necessário realizar diversas análises e transformações nos mesmos para obter a vetorização. Estas análises ocorrem na árvore de representação intermediária gerada pelo compilador, e todas dependem de uma análise base; a análise de dependência de dados.

4.1 Dependência de dados

A análise de dependência permite ao compilador maiores oportunidades de rearranjo de código, o programa é analisado para achar restrições essenciais que previnem o reordenamento de operações, sentenças, ou iterações de um loop.

Os três tipos de dependência utilizados para esta análise são:

- Flow ou Direta, quando uma variável é definida em uma sentença e utilizada em uma subseqüente.
- Anti, usada em uma sentença e definida em uma subseqüente.
- Output, definida em uma sentença e redefinida em uma subseqüente.

As dependências podem ser classificadas como *loop carried*, quando ocorrem entre iterações de um loop ou *loop independent* caso contrário. Grafos de dependência podem ser construídos para facilitar a visualização e implementação.

Trecho 2 Programa Simples

```
(1) a = 0
(2) b = a
(3) c = a + d
(4) d = 2
```

No trecho de código 2 temos uma dependência de fluxo entre as sentenças 1-2 e 1-3 e uma anti-dependência entre 3-4 (figura 3).

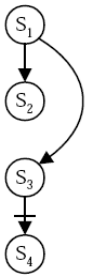

Figure 3: Grafo de dependência

5. VETORIZAÇÃO DE LAÇOS

Compiladores antigos tinham poucas regras sobre os tipos de construções para os quais eles poderiam gerar código. Isto ocorreu até o momento em que pesquisadores começaram a usar grafos de dependência para encontrar laços vetorizáveis[1].

Apenas laços seqüenciais com dependência acíclicas podem ser vetorizados, uma vez que a execução vetorial preserva relação de dependência entre sentenças que aparecem uma após a outra seqüencialmente no código fonte. Assim, boa parte do foco para vetorização auxilia na quebra dessas dependências.

5.1 Strip-mining

Cada arquitetura possui seu próprio tamanho de registradores vetoriais. Com o objetivo de aproveitar o tamanho desses registradores a técnica de *strip mining* pode ser utilizada. O *strip mining* decompõe um único laço em novos 2 laços não-aninhados; o segundo laço (chamado de *strip loop*) caminha entre passos de iterações consecutivas enquanto o primeiro caminha entre iterações únicas em um passo.

Trecho 3 Redução em Fortran

```
forall i = 1 to N do
  a[i]  = b[i] + c[i]
  ASUM += a[i]
```

O código 4 é a versão vetorizada do código 3. Após a aplicação de strip mining, o *strip loop* caminha entre passos de tamanho 64, uma vez que este é o tamanho do registrador vetorial para o exemplo.

Os trechos de código 3 e 4 são exemplos de redução de vetores de dados, operação bastante comum em códigos multimídia. Se a arquitetura suportar instruções de redução então o compilador pode utilizar estas instruções diretamente,

Trecho 4 Redução sem o intrinsic *forall*

```
v4 = vsub v4, v4
for i = 1 to N by 64 do
  VL = max(N-i+1, 64)
  v1 = vfetch b[i]
  v2 = vfetch c[i]
  v3 = vadd v1, v2
  vstore v3, a[i]
  v4 = vadd v3, v4
for i = 0 to min(N-1,63)
  ASUM = ASUM + v4[i]
```

basta tomar o cuidado de acumular os resultados de vários passos e depois reduzir novamente. Se a arquitetura não possuir estas instruções, ainda existe uma abordagem mais eficiente do que um loop seqüencial para resolver o problema.

5.2 Analise de Condicionais

Um dos problemas durante a vetorização é o tratamento de sentenças condicionais. Uma das maneiras de resolver é utilizar vetor de bits, afim de armazenar o resultado de operações de comparação para vetores. Com o vetor de bits calculado, instruções com predicado podem ser utilizadas junto a esse registrador, podendo gerar fluxo condicional sem branches.

Trecho 5 Loop com condicional

```
forall i = 1 to N do
  a[i] = b[i] + c[i]
  if a[i] > 0 then
    b[i] = b[i] + 1
```

O trecho 5 contêm uma verificação que pode resolvida pelo compilador com um código resultante semelhante ao do código 6.

Trecho 6 Código vetorial condicional

```
for i = 1 to N by 64 do
  VL = max(N-i+1, 64)
  v1 = vfetch b[i]
  v2 = vfetch c[i]
  v3 = vadd v1, v2
  vstore v3, a[i]
  m1 = vcmp v3, r0
  r2 = addiu r0, 1
  v1 = vaddc v1, r2, m1
  vstorec v1, b[i], m1
```

Outras abordagem podem ser também adotadas, se houver apenas um vetor de bits, a condição pode estar implícita na própria instrução. No caso de não existirem store condicionais, ambos caminhos podem ser gerados, conseguindo ainda assim aplicar a vetorização.

5.3 Expansão de escalares

A presença de escalares dentro de um laço gera ciclos no grafo de dependência (existe no mínimo a dependência de saída), isso ocorre tanto para variáveis de indução quanto para escalares temporários. Para ser possível vetorizar é necessário remover estes ciclos, removendo as váriaveis de indução e escalares.

As variáveis de indução são removidas por substituição, e os escalares podem ser expandidos. A expansão de escalares é realizada guardando o escalar em um registrador vetorial, assim cada posição do registrador deverá conter o valor do escalar em diferentes iterações, não havendo necessidade de utilizar a memória.

5.4 Dependências cíclicas

Laços com ciclos de dependência são mais complicados de vetorizar e várias abordagens podem ser aplicadas para extrair o que for possível de vetorização. A primeira e mais direta abordagem com dependência cíclica se trata de uma simples otimização; Para ciclos com dependência de saída, o ciclo pode ser ignorado se o valor sendo guardado for invariante no loop e igual em todas as sentenças.

5.5 Loop fission

Aplicando *loop fission* é possível vetorizar laços parcialmente. Isso é feito separando o laço em dois novos laços, um deles conterá o trecho de código com a dependência cíclica e o outro poderá ser vetorizado. Considere o trecho de código 7 e o grafo de dependência da figura 4:

Trecho 7 Dependência cíclica

```
for i = 1 to N do
  a[i] = b[i] + c[i]
  d[i+1] = d[i] + a[i]*c[i]
  c[i+1] = c[i+1]*2
```

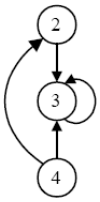

Figure 4: Grafo de dependência

Temos um laço com dependência cíclica na terceira sentença. O trecho 8 contêm o *loop fission* necessário.

Trecho 8 Loop fission

```
for i = 1 to N do
  c[i+1] = c[i+1]*2
  a[i] = b[i] + c[i]
for i = 1 to N do
  d[i+1] = d[i] + a[i]*c[i]
```

Quando há a oportunidade de se aplicar loop fusion porem existe um temporário que ficaria em loops diferentes, como o código 9, pode se aplicar algo semelhante à expansão de escalares, mas dessa vez um registrador vetorial não será suficiente, é necessário alocação na memória. Para contornar isso, se aplica *strip mining* antes do *loop fission*, o resultado pode ser visto no trecho de código 10.

Outro impedimento que pode ocorrer dificultando a vetorização em laços é a presença de condicionais. Podem aparecer dois tipos de condicionais; as que estão no ciclo de dependência e as que não estão. No primeiro caso é necessário

Trecho 9 Dependência cíclica com temporário

```
for i = 1 to N do
  TEMP = b[i] + c[i]
  d[i+1] = d[i] + TEMP * c[i]
  c[i+1] = c[i+1] * 2
```

ter suporte especial em hardware para resolver recorrência booleana, ou seja, inviável. No segundo caso, deve-se colocar a condição em um registrador escalar, expandi-lo e vetorizar como explicado algumas seções atrás.

Trecho 10 Loop fission e strip mining aplicados

```
for i = 1 to N by 64 do
  for v = 0 to min(N-I, 63) do
    a[v] = b[i+v] + c[i+v]
    c[i+v+1] = c[i+v+1]*2
  for v = 0 to min(N-I, 63) do
    d[i+v+1] = d[i+v] + a[v]*c[i+v]
```

5.6 Falsos ciclos

Existem vários truques que podem ser aplicados que permitem a vetorização independente da presença de ciclos. Diversos ciclos existem devido a anti-dependências, principalmente quando essa relação é da sentença consigo mesma. Isso ocorre devido a granularidade de visualização do grafo de dependência (essa anti-dependência não existe de fato). Considere o trecho de código 11:

Trecho 11 Falsos ciclos

```
(1) for i = 1 to N do
(2)  a[i] = a[i+1] + b[i]
```

A figura 5 ilustra a esquerda o grafo de dependência com alta granularidade e a direita o com baixa.

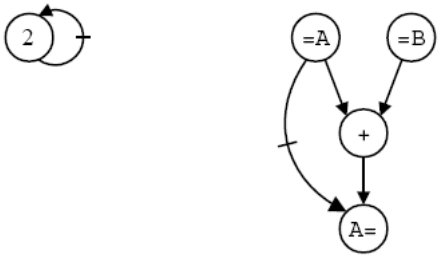

Figure 5: Diferentes granularidades

Refinando a granularidade o compilador consegue eliminar dependências, uma vez que elimina as dependências falsas geradas por uma granularidade alta. A vetorização pode ser conseguida aplicando-se uma ordenação topológica para reordenar o grafo de baixa granularidade, e garantindo que no código gerado em da 12, o fetch de $a[i+1]$ ocorra antes do store de $a[i]$ preservando a relação de anti-dependência.

Na presença de ciclos verdadeiros, o compilador pode vetorizar expressões parciais, basta decidir se o esforço compensa, e caso sim, vetorizar o que for possivel, deixando o resto como estava.

Trecho 12 Sem falsos ciclos

```
for i = 1 to N by 64 do
  VL = min(N-i+1, 64)
  v1 = vfetch a[i+1]
  v2 = vfetch b[i]
  v3 = vadd v1, v2
  vstore v3, a[i]
```

5.7 Dependência cruzada

A técnica de *Index set splitting* pode ser utilizada quando ocorre dependência cruzada, ou seja, um dos operando vetoriais contem um índice crescente e algum outro, decrescente. Suponha trecho de código 13:

Trecho 13 Dependência cruzada

```
for i = 1 to N do
  a[i] = b[i] + c[i]
  d[i] = (a[i] + a[n-i+1])/2
```

O compilador deve descobrir onde esses índices se encontram e separar o conjunto de índices neste ponto. Se aplicarmos esta técnica no código acima, obtemos o código 14, onde ambos os loops podem ser vetorizados.

Trecho 14 Laços resultantes

```
(1) for i=1 to (n+1)/2 do
      a[i] = b[i] + c[i]
      d[i] = (a[i] + a[n-i+1])/2
(2) for i=(n+1)/2+1 to N do
      a[i] = b[i] + c[i]
      d[i] = (a[i] + a[n-i+1])/2
```

5.8 Dependência em tempo de execução

Quando não se conhece um dos índices de acesso a uma posição de um array durante o tempo de compilação não se sabe a direção da dependência (código 15).

Trecho 15 k desconhecido em tempo de compilação

```
for i = 1 to N do
  a[i] = a[i-k] + b[i]
```

Uma maneira de resolver o problema, é fazer com que o compilador gere 2 versões do laço, a primeira para valores $0 < k < N$ e a outra caso contrário (código 16). O laço da condição (1) pode ser vetorizado posto que não há dependência de fluxo e dependência cíclica, já (2) não pode ser vetorizado.

Trecho 16 Dois novos laços

```
(1) if k <= 0 or k >= N then
      forall i=1 to N do
        a[i] = a[i-k] + b[i]
(2) else
      for i=1 to N do
        a[i] = a[i-k] + b[i]
```

5.9 Casos mais simples

Na presença de laços aninhados, se o grafo de dependência não conter ciclos, todos os laços podem ser paralelizados, a transformação fica equivalente a uma atribuição de arrays. O trecho de código 17 é transformado no código 18.

Trecho 17 Laços aninhados

```
for i = 1 to N do
  for j = 2 to M do
    a[i,j] = b[i,j-1] + c[i,j]
    b[i,j] = b[i,j]*2
```

Trecho 18 Codigo vetorizado

```
b[1:n,2:m] = b[1:n,2:m]*2
a[1:n,2:m] = b[1:n,1:m-1] + c[1:n,2:m]
```

5.9.1 *loop interchanging*

Quando um dos laços possuir ciclos de dependência, pode-se aplicar *loop interchanging*. Um exemplo simples dessa técnica é apresentado no trecho 19, o laço original (1) é tranformado em (2). *Loop interchanging* troca a ordem dos

Trecho 19 Exemplo de *Loop interchanging*

```
(1) do i = 1,n
      do j = 1,m
        do k = 1,p
          C[i,j] = C[i,j] + A[i,k] * B[k,j]
(2) do j = 1,m
      do k = 1,p
        do i = 1,n
          C[i,j] = C[i,j] + A[i,k] * B[k,j]
```

laços e pode levar a dependência para o laço de fora, quebrando ciclos. Esta técnica permite escolher o melhor laço para vetorização.

5.9.2 *Loop collapsing*

Trecho 20 Laço em Fortran

```
Real A(100,100), B(100,100)
do I = 1, 100
  do J = 1, 90
    A(I,J) = B(I,J) + 1
  enddo
enddo
```

Laços vetorizados são eficientes quando os limites do laço são largos. Afim de aumenta-los, a técnica de *loop collapsing* pode ser aplicada unificando os laços em apenas um, com um tamanho bastante longo. Os trechos de código 20 e 21 mostram respectivamente o código original e o resultado alcançado utilizando *loop collapsing*.

Trecho 21 Laço em Fortran após *loop collapsing*

```
Real A(100,100), B(100,100)
Real AC(10000),  BC(10000)
Equivalence (A,AC),(B,BC)
do IJ = 1, 9000
  AC(IJ) = BC(IJ) + 1
enddo
```

6. SLP : SUPERWORD LEVEL PARALELISM

Apesar das tecnologias de vetorização explicadas na última seção serem bem entendidas, elas são complexas e frágeis (muitas funcionam apenas em casos muito específicos). Outro ponto negativo é que elas são incapazes de localizar paralelismo do tipo SIMD em blocos básicos. SLP[2] é uma técnica que não realiza extração de vetorização através de paralelismo de laços, ao invés de ter laços como alvo, ela ataca blocos básicos.

SLP é um tipo de paralelismo onde os operandos fontes e destino de uma operação SIMD são empacotados em uma mesma unidade de armazenamento. A detecção é feita com a coleta de sentenças isomórficas em um bloco básico, entenda-se por isomórficas as sentenças que possuem as mesmas operações na mesma ordem. Estas sentenças são empacotadas em uma mesma operação SIMD. A figura 6 ilustra um exemplo de como funciona este empacotamento.

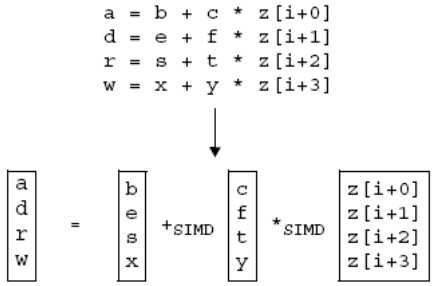

Figure 6: Sentenças isomórficas compactadas

6.1 Notações

- Pack é uma tupla que contem sentenças isomórficas independentes em um bloco básico.
- PackSet é um conjunto de Packs.
- Pair é um Pack de tamanho 2, onde a primeira sentença é considerada o elemento esquerdo (left element) e o outro elemento direito (right element)

6.2 Algoritmo

O Algoritmo de detecção e geração de SLPs, é feita executando-se várias otimizações em determinada ordem. Primeiro, loop unrolling é usando para transformar paralelismo de vetores em SLP (figura 7 e 8), em seguida alignment analysis tenta determinar o endereço de alinhamento de cada instrução de load e store (e os anota para posterior utilização).

```
for (i=0; i<16; i++) {
  localdiff = ref[i] - curr[i];
  diff += abs(localdiff);
}
```

Figure 7: Laço original

Após estas computações, todas as otimizações comuns devem ser executadas, terminando com scalar renaming (removendo dependências de entrada e saída que podem proibir a paralelização) - representações intermediárias que utilizam SSA não necessitam deste último passo.

Os primeiros candidatos a empacotamento são as referências adjacentes de memória, ou seja, acessos a arrays como os da figura 8. Cada bloco básico é analisado em busca de tais sentenças, a adjacência é determinada utilizando-se as informações anotadas de alinhamento e análise de arrays. A primeira ocorrência de cada par de sentenças com acessos adjacente de memória é adicionado ao PackSet.

```
for (i=0; i<16; i+=4) {
  localdiff0 = ref[i+0] - curr[i+0];
  localdiff1 = ref[i+1] - curr[i+1];
  localdiff2 = ref[i+2] - curr[i+2];
  localdiff3 = ref[i+3] - curr[i+3];

  diff += abs(localdiff0);
  diff += abs(localdiff1);
  diff += abs(localdiff2);
  diff += abs(localdiff3);
}
```

Figure 8: SLP alcançado após unrolling e renaming

Com o PackSet inicializado mais grupos podem ser adicionados, isso é feito achando-se mais candidatos. Estes devem cumprir os seguintes requisitos:

- Produzirem operandos fonte para outras sentenças na forma empacotada.
- Utilizar dados já empacotados como operandos.

Estes candidatos podem ser escolhidos varrendo os conjuntos def-use e use-def dos elementos já presentes no PackSet. Se a varredura encontrar sentenças novas que podem ser empacotadas, elas são incorporadas ao PackSet se comprarem as seguintes regras :

- As sentenças são isomórficas e independentes.
- A sentença à esquerda ainda não é esquerda de nenhum par, o mesmo vale para direita
- Informação de alinhamento consistente
- O tempo de execução da nova operação SIMD tem que ser estimadamente menor do que a versão seqüencial.

Quando todos os pares escolhidos segundo as regras acima são escolhidos, eles podem ser combinados em grupos maiores. Dois grupos podem ser combinados quando a sentença esquerda de um é igual à sentença direita de outro (prevenindo também a repetição de sentenças).

A análise de dependência antes do empacotamento garante que todas as sentenças em um grupo podem ser executadas em paralelo de maneira segura. No entanto, apesar de bastante raro, dois grupos podem produzir uma violação de dependência (através de ciclos). Se isso ocorrer, o grupo contendo a sentença mais recentemente não escalonada, é dividido durante a etapa de escalonamento.

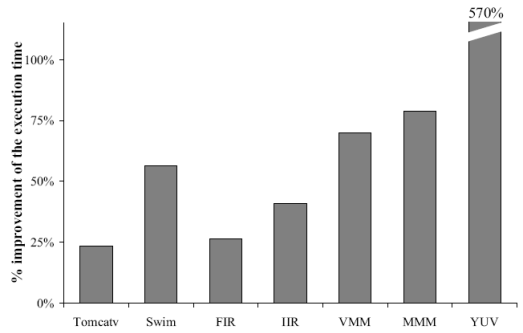

Figure 9: Ganho de desempenho utilizando SLP

As figuras 9 e 10 contem dados de desempenho da utilização de SLP. O processador utilizado chama-se SUIF[3] e o alvo utilizado foi o microprocessador da Motorola MPC7400 com extensões multimídia Altivec.

Benchmark	Speedup
swim	1.24
tomcatv	1.57
FIR	1.26
IIR	1.41
VMM	1.70
MMM	1.79
YUV	6.70

Figure 10: Speedup utilizando SLP

7. CONCLUSÃO

Vetorização foi o primeiro método de automaticamente encontrar paralelismo em laços seqüenciais. Sua utilização foi freqüente em maquina vetoriais como os Crays e hoje tem voltado na forma de extensões multimídia nos processadores modernos. As técnicas mais comuns durante muito tempo foram as baseadas em laços, e técnicas mais novas, como SLP, também tentaram endereçar o problema como uma abordagem diferente. Muitos compiladores modernos ainda não tem suporte a vetorização, e apesar de ser uma área já bastante investigada, a implementação de mecanismos de vetorização em compiladores é uma tarefa árdua mas tem historicamente obtido ótimos desempenhos.

8. REFERENCES

[1] A. Krall and S. Lelait. Compilation techniques for multimedia processors. *Int. J. Parallel Program.*, 28(4):347–361, 2000.

[2] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 145–156, New York, NY, USA, 2000. ACM.

[3] N. Sreraman and R. Govindarajan. A vectorizing compiler for multimedia extensions. *Int. J. Parallel Program.*, 28(4):363–400, 2000.

[4] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood, California, 1996.