Paralelização Automática de Código em Plataformas Heterogêneas Modernas

Rogério A. Gonçalves[1,2] Alfredo Goldman[2]

[1]Departamento de Computação (DACOM)
Universidade Tecnológica Federal do Paraná (UTFPR)
Grupo de Computação de Alto Desempenho e Sistemas Distribuídos

[2]Laboratório de Sistemas de Software
Instituto de Matemática e Estatística (IME)
Universidade de São Paulo (USP)

rogerioag@utfpr.edu.br, {rag,gold}@ime.usp.br

MAC5742 - Computação Paralela e Distribuída

IME-USP USP

Roteiro

IME-USP USP

Objetivos

- Uma visão geral sobre a abordagem de paralelização automática.
- Apresentar as ferramentas e projetos `LLVM`.
- Alguns conceitos sobre o Modelo *Polyhedral*.
- Apresentar algumas ferramentas que utilizam esse modelo.
- Exemplos práticos indo do fonte original ao código para GPU.

Introdução I

- O hardware tem evoluído rapidamente e as plataformas de processamento paralelo tornam-se cada vez mais heterogêneas.
- Dispositivos aceleradores, como GPUs e arranjos de coprocessadores tem sido utilizados na composição do hardware de supercomputadores.
- Nessas novas plataformas, não há suporte direto para as aplicações existentes serem reescritas.
- Os *kits* de desenvolvimento ainda não estão maduros o suficiente.
- Programar novas aplicações ou traduzir aplicações de código legado para estas novas plataformas não é uma tarefa trivial.

IME-USP USP

Introdução II

- O processo de reescrever o código é oneroso e em muitas situações pode ser impraticável.
- Existem diversas tentativas que buscam facilitar o desenvolvimento e atrair usuários de outros contextos.
- *Bindings* e bibliotecas tem sido desenvolvidas para facilitar o acesso a outras linguagens.
- Ponto de vista do software.

Introdução III

Plataformas vs. Aplicações

No cenário atual é evidente a grande distância entre o potencial de desempenho disponibilizado pelas plataformas modernas e as aplicações que precisam fazer uso desse potencial.

- São necessárias soluções que deem suporte a execução de código de aplicações novas e de código legado.
- O ideal é que a ligação entre esses contextos seja feita por ferramentas.

Paralelização de Aplicações I

- Existem pelo menos duas formas de se obter aplicações paralelas:
  - `Concepção de aplicações paralelas:` escrever código em uma linguagem de programação com suporte a paralelismo (aplicações naturalmente paralelas).
  - `Adaptar aplicações existentes:` reescrever código usando extensões que deem o suporte às versões paralelas dessas aplicações (adaptação de modelos).
- Nas duas formas, existe um modelo de programação que fornece suporte à escrita ou adaptação do código, e um modelo de execução que irá executar a versão paralela do código.

IME-USP USP

Paralelização de Aplicações II

- No contexto de tradução de código, tem-se pelo menos duas abordagens conhecidas de paralelização que se distinguem na forma e no momento da intervenção para paralelização:
  - *Via código fonte:* depende da existência do código fonte *(source-to-source)*.
  - *Via código binário:* aplica transformações sobre o binário gerado para a execução da aplicação. (Tradução Dinâmica - DBT).

Aceleração de Aplicações I

- A aceleração de aplicações está relacionada com as plataformas modernas.
- São plataformas heterogêneas, que congregam diversas tecnologias e seus elementos de processamento que trabalham como aceleradores na execução de aplicações.
- O uso de dispositivos aceleradores torna possível a divisão da carga de execução entre um elemento de processamento principal, geralmente uma CPU e elementos que irão trabalhar como coprocessadores.

IME-USP USP

Aceleração de Aplicações II

- Como exemplos dos dispositivos que atualmente compõem essas plataformas temos os arranjos de coprocessadores Xeon Phi (Intel), as GPUs (NVidia, AMD) e FPGAS (Xilinx, Altera).
- Regiões de código paralelizáveis podem ser transformadas em *kernels* e terem sua execução acelerada em uma GPU. Ex.: Laços.
- Trabalhos tem proposto modelos para a execução de aplicações combinando dispositivos *multicore* e multiGPU.

Abordagens que se destacam

Diretivas de Compilação

Diretivas de compilação são usadas para guiar o compilador no processo de tradução e paralelização de código.

Paralelização Automática

São usadas técnicas e modelos para detectar automaticamente quais regiões do código são paralelizáveis.

- Nas duas abordagens uma versão paralela do código é gerada para a plataforma alvo.

Diretivas de Compilação

- O código é anotado com diretivas de preprocessamento (`#pragma`) que são tratadas e o código é gerado. Definições e macros são substituídas pelo código que fará o chaveamento entre dispositivos no processo de ligação com o ambiente de execução.

Ferramentas que utilizam diretivas

- O `OpenMP`.
- O `hiCUDA` (transformar e gerar código para CUDA).
- O `CGCM` (otimiza a comunicação e transferências de dados).
- Os compiladores `PGI` e o `OpenHMPP` que implementam o padrão `OpenACC`.
- O `accULL` é uma implementação do padrão OpenACC com suporte a CUDA e a OpenCL.
- O OpenMC é uma extensão do `OpenMP` para suporte a GPU.

Diretivas de Compilação

- Exemplo da diretiva `#pragma acc kernels` que define uma região de código que será transformada em *kernels*.

```
1  #pragma acc data copy(b[0:n*m]) create(a[0:n*m])
2  {
3    for (iter = 1; iter <= p; ++iter) {
4    #pragma acc kernels
5    {
6      for (i = 1; i < n-1; ++i)
7        for (j = 1; j < m-1; ++j) {
8          /* Suprimido. */
9        }
10     for( i = 1; i < n-1; ++i )
11       for( j = 1; j < m-1; ++j )
12       /* Suprimido. */
13     }
14   }
15 }
```

Paralelização Automática

- A ideia é que o código não seja modificado (nem anotado), pois a ferramenta deve detectar automaticamente as regiões paralelizáveis.

Ferramentas da abordagem automática

- `Par4All` (análise interprocedural)
- `C-to-CUDA`
- `PPCG`
- `KernelGen`

Transformações de Laços I

- A análise de dependência para laços ser feita sobre todas as iterações do laço, como se o laço fosse todo desenrolado.
- Pode acontecer de toda iteração do laço usar valores calculados pela iteração anterior (exceto a primeira).
- Técnicas de otimização de laços são utilizadas por compiladores como mecanismo para a melhoria de desempenho.

IME-USP USP

Transformações de Laços II

- Essas transformações são muito importantes pois possibilitam a exploração de capacidades do processamento paralelo e ajudam a reduzir o tempo de execução associado aos laços.
- O processo de otimização de laços pode ser definido como uma sequência de transformações aplicadas aos laços de um código ou à sua representação intermediária.
- Uma transformação aplicada deve preservar a semântica do código original e a sequência temporal de todas as dependências.

IME-USP USP

Transformações de Laços III

- Existem diversas técnicas específicas para otimização de laços.
- São classificadas conforme as modificações que produzem no código e de acordo com o efeito que garantem na execução do código final.
- Transformações podem ser aplicadas para: analisar, preparar e expor algum possível paralelismo latente.
- São exemplos de otimizações que reorganizam o código de laços: *loop unrolling*, *loop interchange*, *loop skewing*, *loop reversal*, *loop blocking* e *cycle shrinking*, *loop fusion*, *peeling* e distribuição, entre outras.

IME-USP USP

Transformações de Laços: *Loop Tiling* I

- É uma transformação que reorganiza o laço para iterar sobre blocos *(blocking)* de dados.
- O algoritmo clássico de multiplicação de matrizes é um exemplo ao qual esse transformação pode ser aplicada.

```
1 for(int i=0; i<N; i++) {
2   for(int j=0; j<M; j++) {
3     C[i][j] = 0;
4     for( int k=0; k<N; k++) {
5       C[i][j] = C[i][j] + A[i][k] * B[k][j];
6     }
7   }
8 }
```

Transformações de Laços: *Loop Tiling* II

- O código pode ser transformado para iterar sobre blocos *(blocking)*.

```
1  /* B: blocking factor ou tile size. */
2  for (int jj=0; jj<N; jj=jj+B) {
3    for (kk=0; kk<N; kk=kk+B) {
4      for (i=0; i<N; i++) {
5        for (j=jj; j<min(jj+B-1, N); j++) {
6          C[i][j] = 0;
7          for(k=kk; k<min(kk+B-1, N); k++) {
8            C[i][j] = C[i][j] + A[i][k] * B[k][j];
9          }
10       }
11     }
12   }
13 }
```

Transformações de Laços: *Loop Skewing* I

- É uma transformação que reorganiza o espaço de iteração de laços aninhados que iteram sobre arranjos multidimensionais, onde cada iteração do laço mais interno depende de iterações anteriores.
- Os limites do espaço de iteração são alterados para que iterações possam ser executadas em paralelo.
- Tomemos como exemplo o código com dois laços aninhados que fazem acesso a um arranjo multidimensional calculando cada elemento com base nos elementos vizinhos (wavefront).

```
1 for(i=1; i<=n-1; i++) {
2   for(j=1; j<=m-1; j++) {
3     A[i][j] = (A[i-1][j] + A[i][j-1] + A[i+1][j] + A[i][j+1])/4;
4   }
5 }
```

Transformações de Laços: *Loop Skewing* II

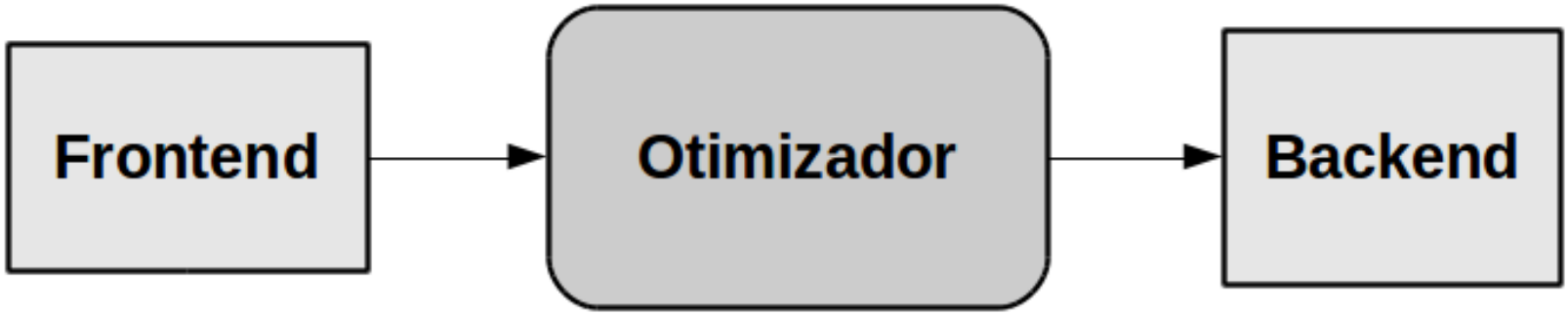
- O espaço de iteração foi deslocado para que iterações possam ser executadas em paralelo.

```
1 if ((m >= 2) && (n >= 2)) {
2   for (c1=2; c1<=n+m-2; c1++) {
3     for (c2=max(1,c1-m+1); c2<=min(c1-1,n-1); c2++) {
4       A[c2][c1-c2] = (A[c2-1][c1-c2] + A[c2][c1-c2-1] + A[
            c2+1][c1-c2] + A[c2][c1-c2+1])/4;
5     }
6   }
7 }
```

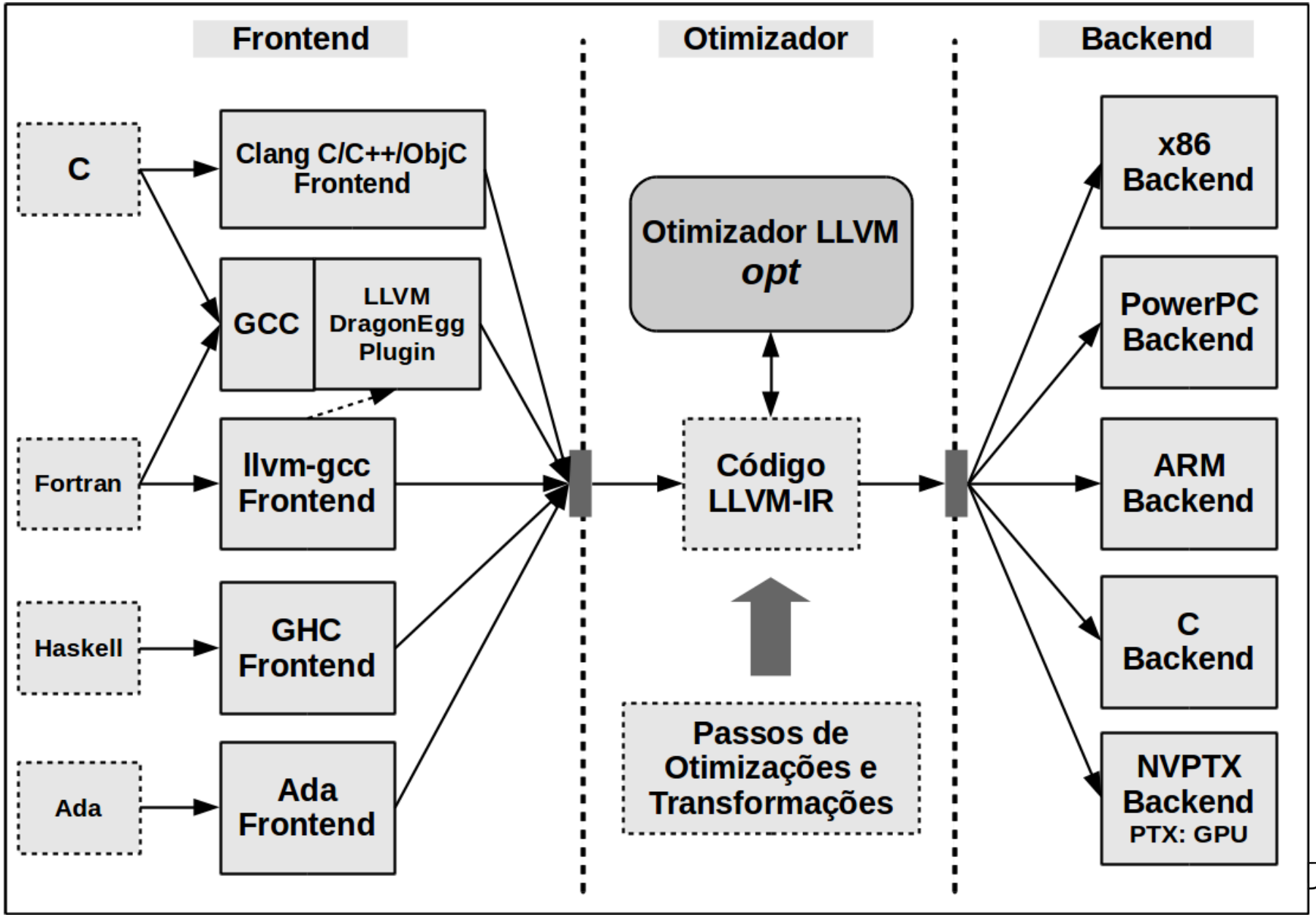
- Este exemplo será utilizado para mostrar a ideia do Modelo *Polyhedral*.

Por que usar LLVM? I

- `LLVM (llvm.org)` é um projeto que fornece uma infraestrutura para a construção de compiladores.
- Desenvolvimento: LLVM Team - University of Illinois at Urbana-Champaign.
- Licença: *University of Illinois/NCSA Open Source License*, certificada pela OSI.
- Sua organização modular facilita o reuso no desenvolvimento de ferramentas de compilação.



Por que usar LLVM? II



Exemplo: LLVM IR I

Código em C.

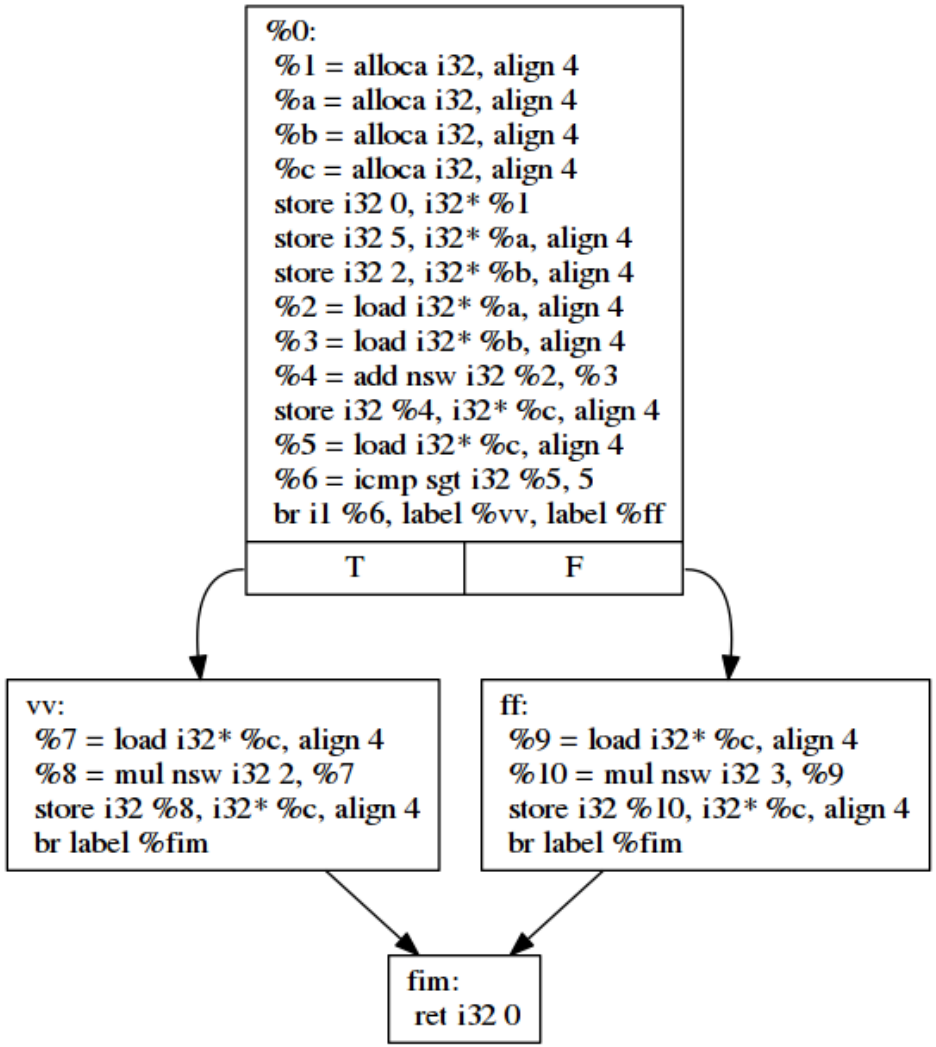
```
 1 int main(){
 2   int a, b, c;
 3   a = 5;
 4   b = 2;
 5   c = a + b;
 6
 7   if(c > 5){
 8     c = 2 * c;
 9   }
10   else{
11     c = 3 * c;
12   }
13
14   return 0;
15 }
```

Código em LLVM-IR.

```
 1 define i32 @main() #0 {
 2   %1 = alloca i32, align 4
 3   %a = alloca i32, align 4
 4   %b = alloca i32, align 4
 5   %c = alloca i32, align 4
 6   store i32 0, i32* %1
 7   store i32 5, i32* %a, align 4
 8   store i32 2, i32* %b, align 4
 9   %2 = load i32* %a, align 4
10   %3 = load i32* %b, align 4
11   %4 = add nsw i32 %2, %3
12   store i32 %4, i32* %c, align 4
13   %5 = load i32* %c, align 4
14   %6 = icmp sgt i32 %5, 5
15   br i1 %6, label %vv, label %ff
16
17 vv:                ; preds = %0
18   %7 = load i32* %c, align 4
19   %8 = mul nsw i32 2, %7
20   store i32 %8, i32* %c, align 4
21   br label %fim
22
23 ff:                ; preds = %0
24   %9 = load i32* %c, align 4
25   %10 = mul nsw i32 3, %9
26   store i32 %10, i32* %c, align 4
27   br label %fim
28
29 fim:               ; preds = %vv, %ff
30   ret i32 0
31 }
```

Exemplo: LLVM IR I

```
 1 define i32 @main() #0 {
 2   %1 = alloca i32, align 4
 3   %a = alloca i32, align 4
 4   %b = alloca i32, align 4
 5   %c = alloca i32, align 4
 6   store i32 0, i32* %1
 7   store i32 5, i32* %a, align 4
 8   store i32 2, i32* %b, align 4
 9   %2 = load i32* %a, align 4
10   %3 = load i32* %b, align 4
11   %4 = add nsw i32 %2, %3
12   store i32 %4, i32* %c, align 4
13   %5 = load i32* %c, align 4
14   %6 = icmp sgt i32 %5, 5
15   br i1 %6, label %vv, label %ff
16
17 vv:                ; preds = %0
18   %7 = load i32* %c, align 4
19   %8 = mul nsw i32 2, %7
20   store i32 %8, i32* %c, align 4
21   br label %fim
22
23 ff:                ; preds = %0
24   %9 = load i32* %c, align 4
25   %10 = mul nsw i32 3, %9
26   store i32 %10, i32* %c, align 4
27   br label %fim
28
29 fim:               ; preds = %vv, %ff
30   ret i32 0
31 }
```



CFG for 'main' function

Exemplo: LLVM IR I

Código em C.

```
 1 int funcA(int x, int y){
 2   int result = x * y;
 3   return result;
 4 }
 5
 6 int main(){
 7   int a, b, c;
 8   a = 5;
 9   c = funcA(a, 2);
10
11   return 0;
12 }
```

Código em LLVM-IR.

```
 1 define i32 @funcA(i32 %x, i32 %y) #0 {
 2   %1 = alloca i32, align 4
 3   %2 = alloca i32, align 4
 4   %result = alloca i32, align 4
 5   store i32 %x, i32* %1, align 4
 6   store i32 %y, i32* %2, align 4
 7   %3 = load i32* %1, align 4
 8   %4 = load i32* %2, align 4
 9   %5 = mul nsw i32 %3, %4
10   store i32 %5, i32* %result, align 4
11   %6 = load i32* %result, align 4
12   ret i32 %6
13 }
14 define i32 @main() #0 {
15   %1 = alloca i32, align 4
16   %a = alloca i32, align 4
17   %b = alloca i32, align 4
18   %c = alloca i32, align 4
19   store i32 0, i32* %1
20   store i32 5, i32* %a, align 4
21   %2 = load i32* %a, align 4
22   %3 = call i32 @funcA(i32 %2, i32 2)
23   store i32 %3, i32* %c, align 4
24   ret i32 0
25 }
```

IME-USP USP

Exemplo: LLVM IR I

Código em C.

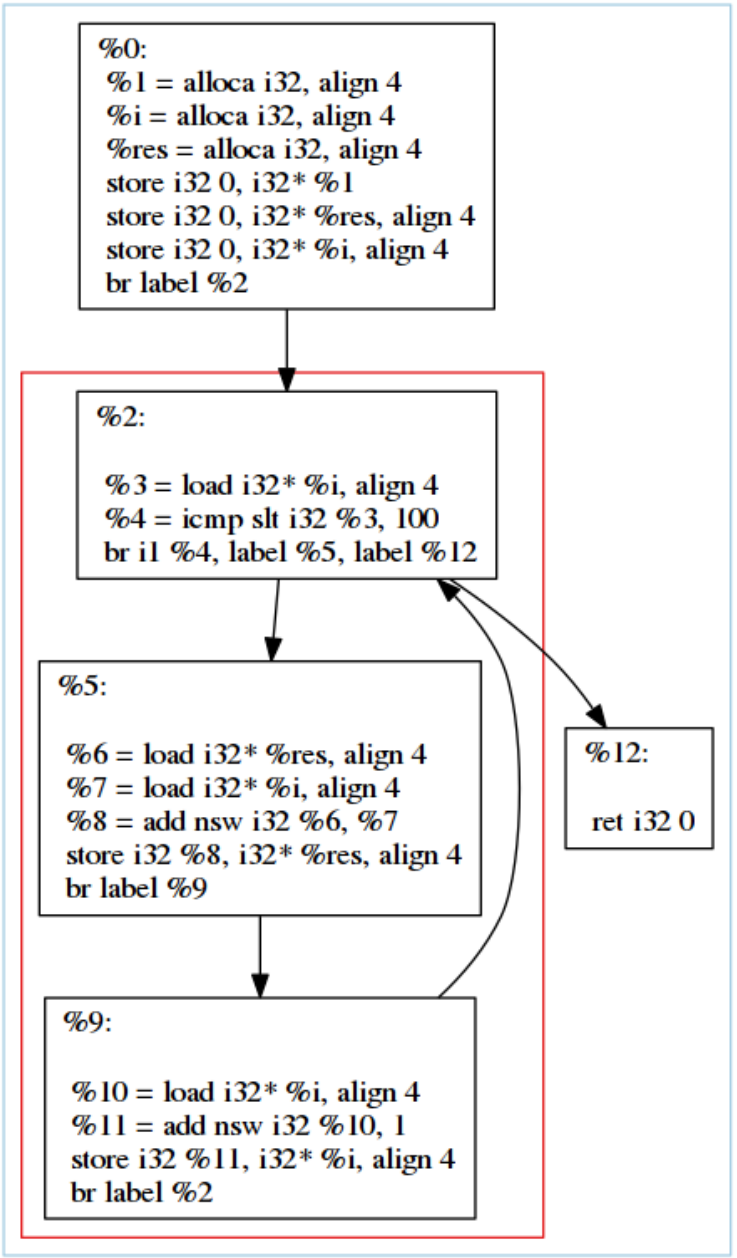
```
1 int main(){
2   int i;
3   int res = 0;
4   for(i=0; i < 100; i++){
5     res = res + i;
6   }
7
8   return 0;
9 }
```

Código em LLVM-IR.

```
1  define i32 @main() #0 {
2    %1 = alloca i32, align 4
3    %i = alloca i32, align 4
4    %res = alloca i32, align 4
5    store i32 0, i32* %1
6    store i32 0, i32* %res, align 4
7    store i32 0, i32* %i, align 4
8    br label %2
9
10 ; <label>:2                ; preds = %9, %0
11   %3 = load i32* %i, align 4
12   %4 = icmp slt i32 %3, 100
13   br i1 %4, label %5, label %12
14
15 ; <label>:5                ; preds = %2
16   %6 = load i32* %res, align 4
17   %7 = load i32* %i, align 4
18   %8 = add nsw i32 %6, %7
19   store i32 %8, i32* %res, align 4
20   br label %9
21
22 ; <label>:9                ; preds = %5
23   %10 = load i32* %i, align 4
24   %11 = add nsw i32 %10, 1
25   store i32 %11, i32* %i, align 4
26   br label %2
27
28 ; <label>:12               ; preds = %2
29   ret i32 0
30 }
```

Exemplo: LLVM IR I

```
 1 define i32 @main() #0 {
 2   %1 = alloca i32, align 4
 3   %i = alloca i32, align 4
 4   %res = alloca i32, align 4
 5   store i32 0, i32* %1
 6   store i32 0, i32* %res, align 4
 7   store i32 0, i32* %i, align 4
 8   br label %2
 9
10 ; <label>:2                ; preds = %9, %0
11   %3 = load i32* %i, align 4
12   %4 = icmp slt i32 %3, 100
13   br i1 %4, label %5, label %12
14
15 ; <label>:5                ; preds = %2
16   %6 = load i32* %res, align 4
17   %7 = load i32* %i, align 4
18   %8 = add nsw i32 %6, %7
19   store i32 %8, i32* %res, align 4
20   br label %9
21
22 ; <label>:9                ; preds = %5
23   %10 = load i32* %i, align 4
24   %11 = add nsw i32 %10, 1
25   store i32 %11, i32* %i, align 4
26   br label %2
27
28 ; <label>:12               ; preds = %2
29   ret i32 0
30 }
```



Ferramentas LLVM

- Projetos como o `DragonEgg`, `PoLLy` e `NVPTX` possibilitam que regiões paralelizáveis sejam traduzidas para *kernels* que possam ter sua execução lançada em GPUs.
- `DragonEgg`: Plugin para `GCC` que traduz código intermediário (`GIMPLE`) para a representação intermediária do LLVM (LLVM-IR).
- `PoLLy`: Implementação do Modelo *Polyhedral* para LLVM.
- `NVPTX`: *Backend* do `LLVM` para gerar código PTX (intermediário das GPUs da NVIDIA).

Gerando Código para GPU

Exemplo Soma de Vetores I

- Código do exemplo soma de vetores:

```
1  int main() {
2      int i;
3      /* Inicializacao dos vetores. */
4      init_array();
5
6      /* Calculo. */
7      for (i = 0; i < N; i++) {
8        h_c[i] = h_a[i] + h_b[i];
9      }
10
11     /* Resultados. */
12     print_array();
13     check_result();
14
15     return 0;
16 }
```

Exemplo Soma de Vetores II

- Código LLVM-IR equivalente gerado pela infraestrutura do LLVM:

```
1  @h_a = common global [1024 x float] zeroinitializer, align 16
2  @h_b = common global [1024 x float] zeroinitializer, align 16
3  @h_c = common global [1024 x float] zeroinitializer, align 16
4  define void @init_array() #0 {
5  }
6  define void @print_array() #0 {
7  }
8  define void @check_result() #0 {
9  }
10 define i32 @main() #0 {
11   %1 = alloca i32, align 4
12   %i = alloca i32, align 4
13   store i32 0, i32* %1
14   call void @init_array()
15   store i32 0, i32* %i, align 4
16   br label %2
17
18 ; <label>:2                    ; preds = %18, %0
19   %3 = load i32* %i, align 4
20   %4 = icmp slt i32 %3, 1024
```

Exemplo Soma de Vetores III

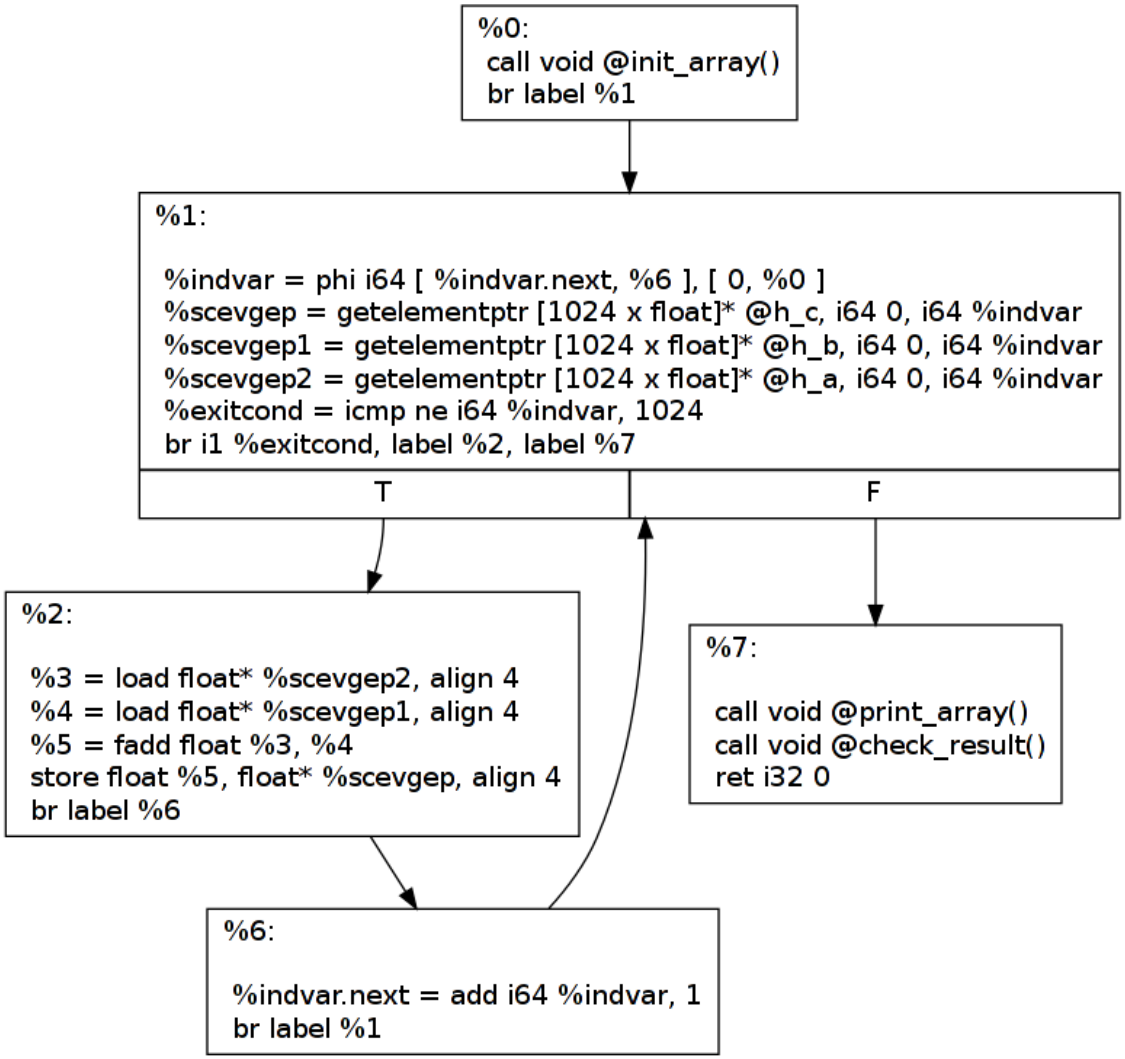
```
21    br i1 %4, label %5, label %21
22
23  ; <label>:5                     ; preds = %2
24    %6 = load i32* %i, align 4
25    %7 = sext i32 %6 to i64
26    %8 = getelementptr inbounds [1024 x float]* @h_a, i32 0, ↩
          i64 %7
27    %9 = load float* %8, align 4
28    %10 = load i32* %i, align 4
29    %11 = sext i32 %10 to i64
30    %12 = getelementptr inbounds [1024 x float]* @h_b, i32 0, ↩
          i64 %11
31    %13 = load float* %12, align 4
32    %14 = fadd float %9, %13
33    %15 = load i32* %i, align 4
34    %16 = sext i32 %15 to i64
35    %17 = getelementptr inbounds [1024 x float]* @h_c, i32 0, ↩
          i64 %16
36    store float %14, float* %17, align 4
37    br label %18
38
39  ; <label>:18                     ; preds = %5
```

Exemplo Soma de Vetores IV

```
40    %19 = load i32* %i, align 4
41    %20 = add nsw i32 %19, 1
42    store i32 %20, i32* %i, align 4
43    br label %2
44
45 ; <label>:21                ; preds = %2
46    call void @print_array()
47    call void @check_result()
48    ret i32 0
49 }
```

Exemplo Soma de Vetores V

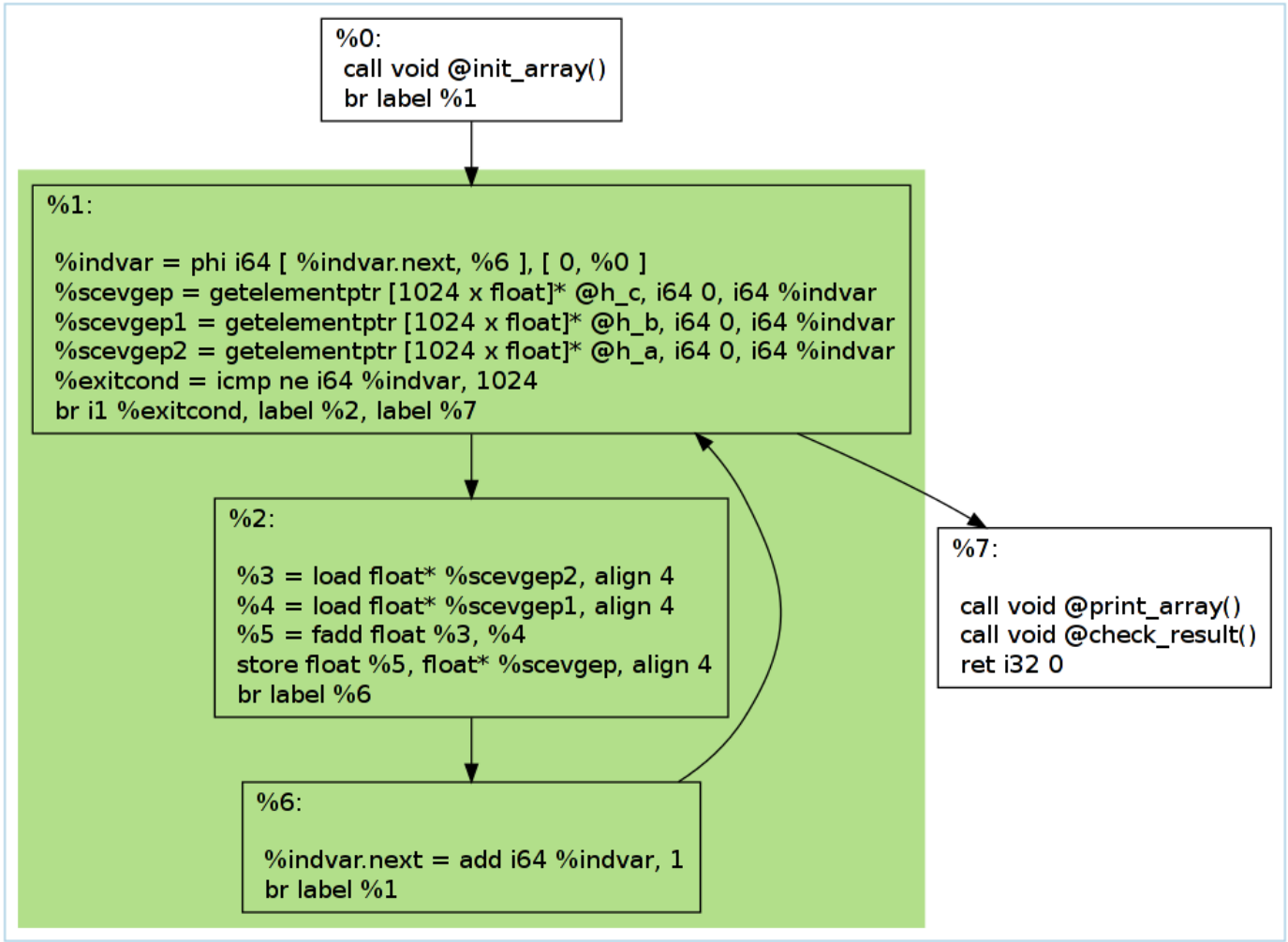
O `CFG` do laço de repetição contido na função `main`:



CFG for 'main' function

Exemplo Soma de Vetores VI

Laço em destaque como SCoP detectado pelo PoLLy:



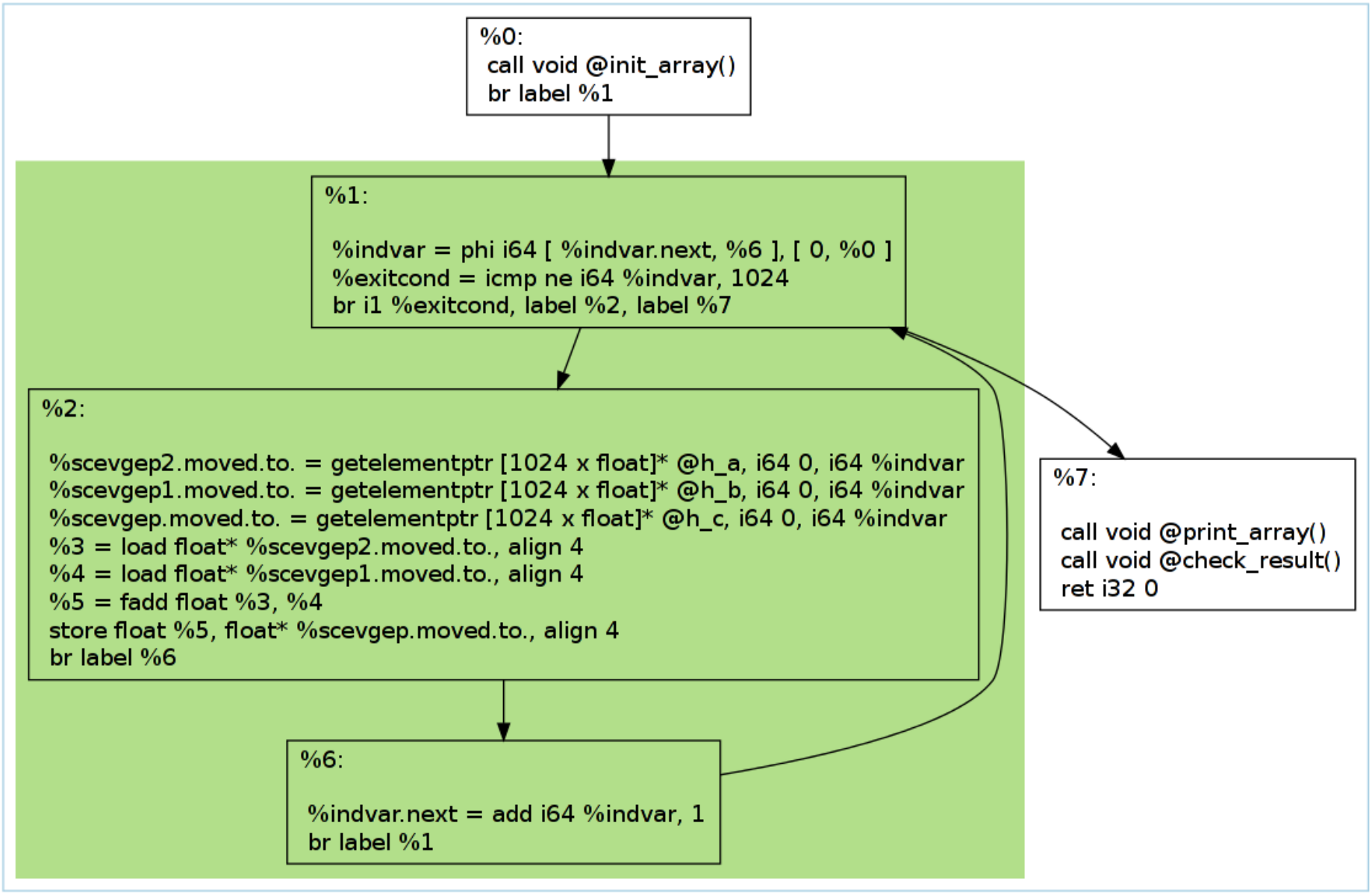
Scop Graph for 'main' function

Exemplo Soma de Vetores VII

- O PoLLy detecta as *SCoPs* do código que serão traduzidas para a representação *polyhedral*.
- O PoLLy pode ser usado para separar o código em blocos independentes.

- Visualização do código com blocos independentes que foram separados pelo PoLLy.
- O SCoP continua sendo o mesmo, porém foi alterado internamente, com a recuperação dos ponteiros para elementos dos arranjos sendo feita no início do corpo do laço.

Exemplo Soma de Vetores VIII



Scop Graph for 'main' function

Exemplo da Estrutura de um *kernel* para GPU

- *Kernel* para a soma de vetores:

```
1  __global__ void vecAdd(float* A, float* B, float* C, int n) {
2    int id = blockDim.x * blockIdx.x + threadIdx.x;
3    if (id < n) {
4      C[id] = A[id] + B[id];
5    }
6  }
7
8  int main() {
9    /* ... */
10   int threadsPerBlock = 256;
11   int blocksPerGrid =(N + threadsPerBlock - 1) /
         threadsPerBlock;
12   vecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
13   /* ... */
14 }
```

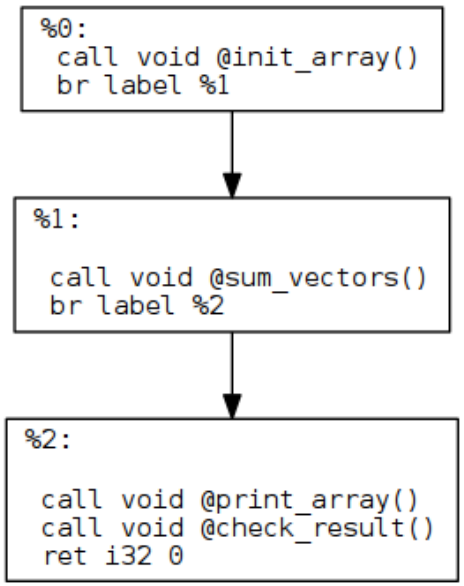
Mapeamento de identificadores no LLVM-IR

O NVPTX fornece declarações de funções que recuperam os identificadores:

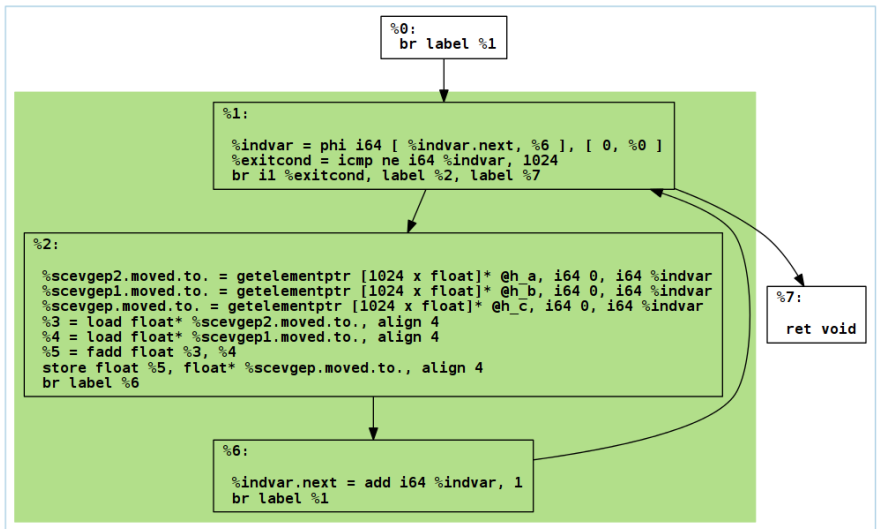
CUDA	Significado	LLVM-IR para o NVPTX
`threadIdx.x`	Id da Thread na dim. x	`declare i32 @llvm.nvvm.read.ptx.sreg.tid.x()`
`threadIdx.y`	Id da Thread na dim. y	`declare i32 @llvm.nvvm.read.ptx.sreg.tid.y()`
`threadIdx.z`	Id da Thread na dim. z	`declare i32 @llvm.nvvm.read.ptx.sreg.tid.z()`
`blockIdx.x`	Id do Bloco na dim. x	`declare i32 @llvm.nvvm.read.ptx.sreg.ctaid.x()`
`blockIdx.y`	Id do Bloco na dim. y	`declare i32 @llvm.nvvm.read.ptx.sreg.ctaid.y()`
`blockIdx.z`	Id do Bloco na dim. z	`declare i32 @llvm.nvvm.read.ptx.sreg.ctaid.z()`
`blockDim.x`	Dimensão x do Bloco	`declare i32 @llvm.nvvm.read.ptx.sreg.ntid.x()`
`blockDim.y`	Dimensão y do Bloco	`declare i32 @llvm.nvvm.read.ptx.sreg.ntid.y()`
`blockDim.z`	Dimensão z do Bloco	`declare i32 @llvm.nvvm.read.ptx.sreg.ntid.z()`
`gridDim.x`	Dimensão x do Grid	`declare i32 @llvm.nvvm.read.ptx.sreg.nctaid.x()`
`gridDim.y`	Dimensão y do Grid	`declare i32 @llvm.nvvm.read.ptx.sreg.nctaid.y()`
`gridDim.z`	Dimensão z do Grid	`declare i32 @llvm.nvvm.read.ptx.sreg.nctaid.z()`
`warp size`	Tamanho do warp	`declare i32 @llvm.nvvm.read.ptx.sreg.warpsize()`

Transformação para Kernels I

- Funções de módulos LLVM-IR podem ser transformados em *kernels*.
- Para exemplificar a transformação de funções para a estrutura de *kernels*, consideremos o exemplo soma de vetores. Tanto o `CFG` da função principal (`main`), quanto a função `sum_vectors()`.



CFG for 'main' function



Scop Graph for 'sum_vectors' function

Transformação para Kernels II

- A correspondência entre os trechos de código LLVM-IR original e o formato para o `NVPTX`. Transformação da função `sum_vectors` para *kernel*:

```
define void @sum_vectors() #0 {
  br label %1

; <label>:1                                       ; preds = %6, %0
  %indvar = phi i64 [ %indvar.next, %6 ], [ 0, %0 ]
  %exitcond = icmp ne i64 %indvar, 1024
  br i1 %exitcond, label %2, label %7

; <label>:2                                       ; preds = %1
  %arrayida = getelementptr [1024 x float]* @h_a, i64 0, i64 %indvar
  %arrayidb = getelementptr [1024 x float]* @h_b, i64 0, i64 %indvar
  %arrayidc = getelementptr [1024 x float]* @h_c, i64 0, i64 %indvar
  %elema = load float* %arrayida, align 4
  %elemb = load float* %arrayidb, align 4
  %sumab = fadd float %elema, %elemb
  store float %sumab, float* %arrayidc, align 4
  br label %6

; <label>:6                                       ; preds = %2
  %indvar.next = add i64 %indvar, 1
  br label %1

; <label>:7                                       ; preds = %1
  ret void
}
```

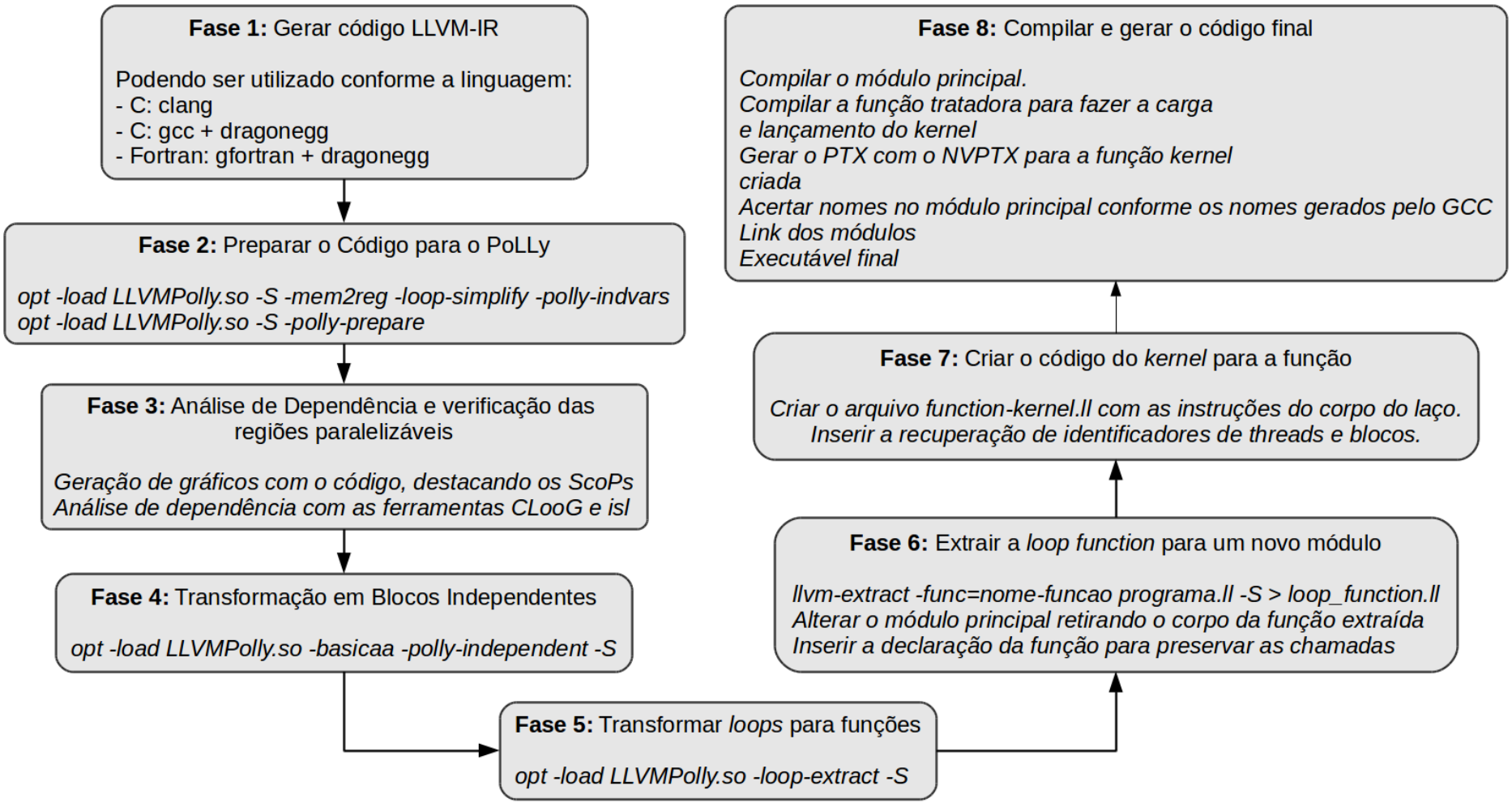
```
define ptx_kernel void @sum_kernel(float addrspace(1)* %d_a,
                                   float addrspace(1)* %d_b,
                                   float addrspace(1)* %d_c) uwtable {
entry:
    ;id = blockDim.x * blockIdx.x + threadIdx.x;
    %0 = call i32 @llvm.nvvm.read.ptx.sreg.ctaid.x()
    %1 = call i32 @llvm.nvvm.read.ptx.sreg.ntid.x()
    %2 = call i32 @llvm.nvvm.read.ptx.sreg.tid.x()
    %mul = mul i32 %0, %1
    %add = add i32 %mul, %2
    %id = sext i32 %add to i64

    %arrayida = getelementptr float addrspace(1)* %d_a, i64 %id
    %arrayidb = getelementptr float addrspace(1)* %d_b, i64 %id
    %arrayidc = getelementptr float addrspace(1)* %d_c , i64 %id
    %elema = load float addrspace(1)* %arrayida, align 4
    %elemb = load float addrspace(1)* %arrayidb, align 4
    %sumab = fadd float %elema, %elemb
    store float %sumab , float addrspace(1)* %arrayidc , align 4

    ret void
}
```

Transformação para Kernels III

- Sequência de passos necessários para gerar, carregar o código do *kernel* e efetuar transferências:

```
1  void sum_vectors() {
2    /* Gerando o PTX. */
3    char *ptx = generatePTX(ll, size);
4    /* Criando um contexto. */
5    cuCtxCreate(&cudaContext, 0, cudaDevice);
6    /* Carregando o modulo. */
7    cuModuleLoadDataEx(&cudaModule, ptx, 0, 0, 0);
8    /* Recuperando a funcao kernel. */
9    cuModuleGetFunction(&kernelFunction, cudaModule, "sum_kernel");
10   /* Transferindo os dados para a memoria do dispositivo. */
11   cuMemcpyHtoD(devBufferA, &hostA[0], sizeof(float)*N);
12   cuMemcpyHtoD(devBufferB, &hostB[0], sizeof(float)*N);
13   /* Definicao das dimensoes do arranjo de threads. */
14   calculateDimensions(&blockSizeX, &blockSizeY, &blockSizeZ, &gridSizeX, &
         gridSizeY, &gridSizeZ);
15   /* Parametros da funcao kernel. */
16   void *kernelParams[] = { &devBufferA, &devBufferB, &devBufferC };
17   /* Lancando a execucao do kernel. */
18   cuLaunchKernel(kernelFunction, gridSizeX, gridSizeY, gridSizeZ, blockSizeX
         , blockSizeY, blockSizeZ, 0, NULL, kernelParams, NULL);
19   /* Recuperando o resultado. */
20   cuMemcpyDtoH(&hostC[0], devBufferC, sizeof(float)*N);
21   return 0;
22 }
```

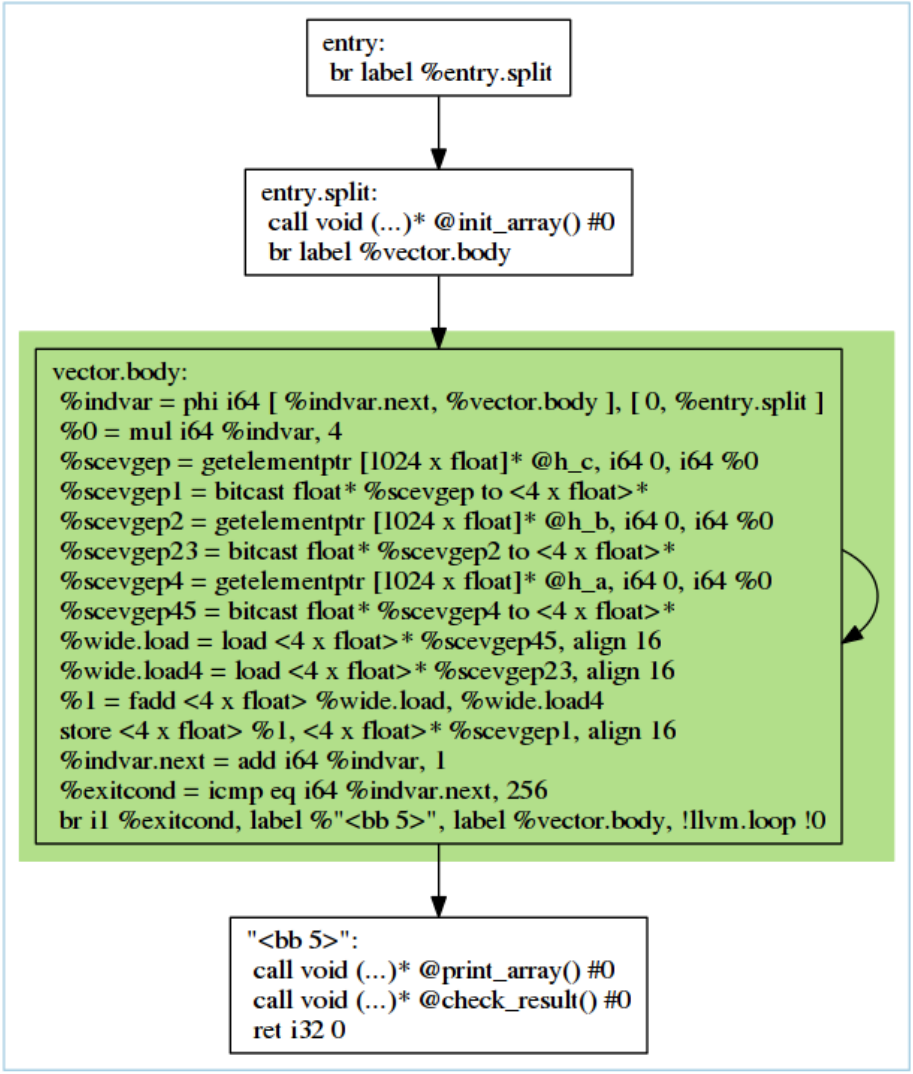
Sequência de Passos para os Próximos Exemplos I



Exemplo: vectorAdd (Soma de Vetores)

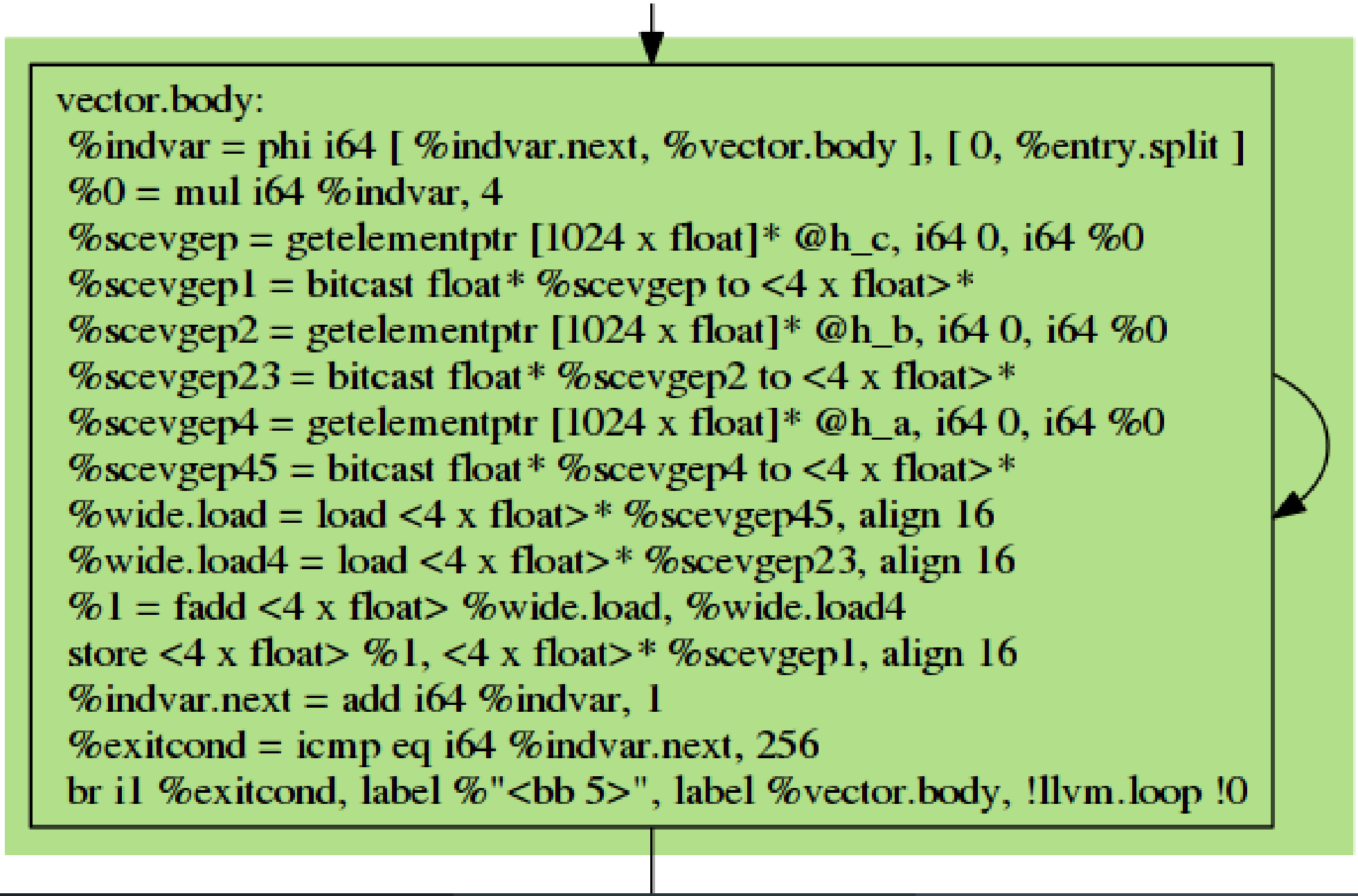
```
 1 float h_a[N];
 2 float h_b[N];
 3 float h_c[N];
 4
 5 int main() {
 6   int i;
 7   /* Inicializacao dos vetores. */
 8   init_array();
 9
10   /* Calculo. */
11   for (i = 0; i < N; i++) {
12     h_c[i] = h_a[i] + h_b[i];
13   }
14
15   /* Resultados. */
16   print_array();
17   check_result();
18
19   return 0;
20 }
```

Código em C.
Transformada em LLVM-IR. (Fase 1)
SCoP detectado pelo PoLLy (Fases 2, 3 e 4).



Scop Graph for 'main' function

Exemplo: `vectorAdd` (Soma de Vetores)



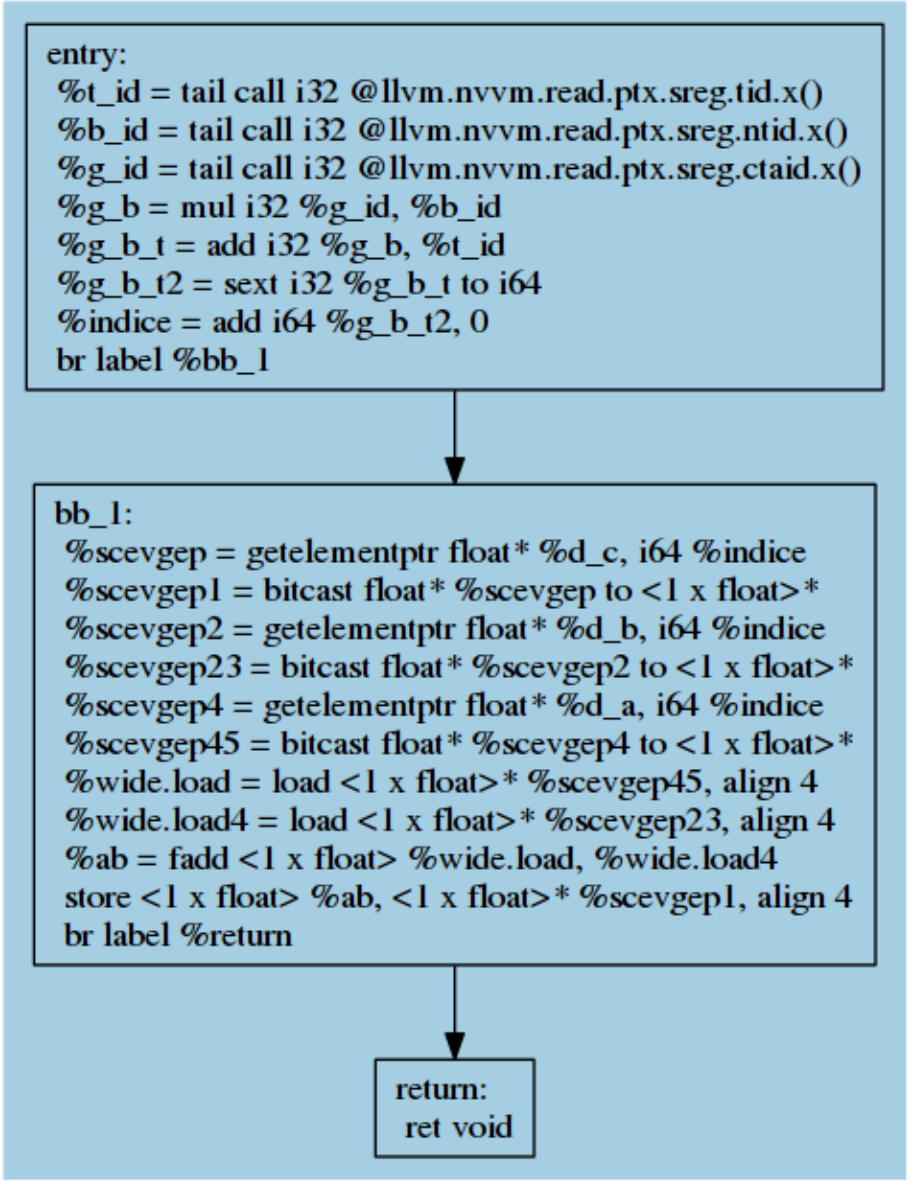
Transformação para vectoradd-kernel

```
; ModuleID = 'vectoradd_preopt_prepare_indep_blocks_loop_extract.ll'
target datalayout = "e-p:64:64:64-S128-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-f16:16:
16-f32:32:32-f64:64:64-f128:128:128-v64:64:64-v128:128:128-a0:0:64-s0:64:64-f80:128:128-n8:16:32:64"
target triple = "x86_64--linux-gnu"

@h_c = external global [1024 x float], align 32
@h_b = external global [1024 x float], align 32
@h_a = external global [1024 x float], align 32

; Function Attrs: nounwind
define hidden void @main_vector_body() #0 {
newFuncRoot:
  br label %vector.body

"<bb 5>.exitStub":                                ; preds = %vector.body
  ret void

vector.body:                                      ; preds = %vector.body.vector.body_crit_edge, %newFuncRoot
  %indvar = phi i64 [ %indvar.next, %vector.body.vector.body_crit_edge ], [ 0, %newFuncRoot ]
  %0 = mul i64 %indvar, 4

  %scevgep = getelementptr [1024 x float]* @h_c, i64 0, i64 %0
  %scevgep1 = bitcast float* %scevgep to <4 x float>*
  %scevgep2 = getelementptr [1024 x float]* @h_b, i64 0, i64 %0
  %scevgep23 = bitcast float* %scevgep2 to <4 x float>*
  %scevgep4 = getelementptr [1024 x float]* @h_a, i64 0, i64 %0
  %scevgep45 = bitcast float* %scevgep4 to <4 x float>*
  %wide.load = load <4 x float>* %scevgep45, align 16
  %wide.load4 = load <4 x float>* %scevgep23, align 16
  %1 = fadd <4 x float> %wide.load, %wide.load4
  store <4 x float> %1, <4 x float>* %scevgep1, align 16

  %indvar.next = add i64 %indvar, 1
  %exitcond = icmp eq i64 %indvar.next, 256
  br i1 %exitcond, label %"<bb 5>.exitStub", label %vector.body.vector.body_crit_edge, !llvm.loop !0

vector.body.vector.body_crit_edge:                ; preds = %vector.body
  br label %vector.body
}

attributes #0 = { nounwind }

!0 = metadata !{metadata !0, metadata !1, metadata !2}
!1 = metadata !{metadata !"llvm.vectorizer.width", i32 1}
!2 = metadata !{metadata !"llvm.vectorizer.unroll", i32 1}
```

```
; ModuleID = 'vectoradd_preopt_prepare_indep_blocks_loop_extract_main_vector.body.ll'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-f32:32:32-f64
:64:64-v16:16:16-v32:32:32-v64:64:64-v128:128:128-n16:32:64"
target triple = "nvptx64-nvidia-cuda"

; Intrinsic to read X component of thread ID
declare i32 @llvm.nvvm.read.ptx.sreg.tid.x() readnone nounwind
declare i32 @llvm.nvvm.read.ptx.sreg.ntid.x() readnone nounwind
declare i32 @llvm.nvvm.read.ptx.sreg.ctaid.x() readnone nounwind


define ptx_kernel void @vectoradd_kernel(float* %d_a, float* %d_b, float* %d_c) uwtable {
entry:
  ;id = blockDim.x * blockIdx.x + threadIdx.x;
  %t_id = tail call i32 @llvm.nvvm.read.ptx.sreg.tid.x()
  %b_id = tail call i32 @llvm.nvvm.read.ptx.sreg.ntid.x()
  %g_id = tail call i32 @llvm.nvvm.read.ptx.sreg.ctaid.x()
  %g_b = mul i32 %g_id, %b_id
  %g_b_t = add i32 %g_b, %t_id
  %g_b_t2 = sext i32 %g_b_t to i64
  %indice = add i64 %g_b_t2, 0

  br label %bb_1

bb_1:                                             ; preds = %entry
  %scevgep = getelementptr float* %d_c , i64 %indice
  %scevgep1 = bitcast float* %scevgep to <1 x float>*
  %scevgep2 = getelementptr float* %d_b, i64 %indice
  %scevgep23 = bitcast float* %scevgep2 to <1 x float>*
  %scevgep4 = getelementptr float* %d_a, i64 %indice
  %scevgep45 = bitcast float* %scevgep4 to <1 x float>*
  %wide.load = load <1 x float>* %scevgep45, align 4
  %wide.load4 = load <1 x float>* %scevgep23, align 4
  %ab = fadd <1 x float> %wide.load, %wide.load4
  store <1 x float> %ab , <1 x float>* %scevgep1 , align 4

  br label %return

return:                                           ; preds = %bb_1
        ret void
}

!nvvm.annotations = !{!0}
!0 = metadata !{void (float*,
                      float*,
                      float*)* @vectoradd_kernel, metadata !"vectoradd_kernel", i32 1}
```

Código para o vectoradd-kernel

```
; ModuleID = 'vectoradd_preopt_prepare_indep_blocks_loop_extract_main_vector.body.ll'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-f32:32:32-f64
:64:64-v16:16:16-v32:32:32-v64:64:64-v128:128:128-n16:32:64"
target triple = "nvptx64-nvidia-cuda"

; Intrinsic to read X component of thread ID
declare i32 @llvm.nvvm.read.ptx.sreg.tid.x() readnone nounwind
declare i32 @llvm.nvvm.read.ptx.sreg.ntid.x() readnone nounwind
declare i32 @llvm.nvvm.read.ptx.sreg.ctaid.x() readnone nounwind


define ptx_kernel void @vectoradd_kernel(float* %d_a, float* %d_b, float* %d_c) uwtable {
entry:
  ;id = blockDim.x * blockIdx.x + threadIdx.x;
  %t_id = tail call i32 @llvm.nvvm.read.ptx.sreg.tid.x()
  %b_id = tail call i32 @llvm.nvvm.read.ptx.sreg.ntid.x()
  %g_id = tail call i32 @llvm.nvvm.read.ptx.sreg.ctaid.x()
  %g_b = mul i32 %g_id, %b_id
  %g_b_t = add i32 %g_b, %t_id
  %g_b_t2 = sext i32 %g_b_t to i64
  %indice = add i64 %g_b_t2, 0

  br label %bb_1

bb_1:                                             ; preds = %entry
  %scevgep = getelementptr float* %d_c , i64 %indice
  %scevgep1 = bitcast float* %scevgep to <1 x float>*
  %scevgep2 = getelementptr float* %d_b, i64 %indice
  %scevgep23 = bitcast float* %scevgep2 to <1 x float>*
  %scevgep4 = getelementptr float* %d_a, i64 %indice
  %scevgep45 = bitcast float* %scevgep4 to <1 x float>*
  %wide.load = load <1 x float>* %scevgep45, align 4
  %wide.load4 = load <1 x float>* %scevgep23, align 4
  %ab = fadd <1 x float> %wide.load, %wide.load4
  store <1 x float> %ab , <1 x float>* %scevgep1 , align 4
  br label %return

return:                                           ; preds = %bb_1
        ret void
}

!nvvm.annotations = !{!0}
!0 = metadata !{void (float*,
                      float*,
                      float*)* @vectoradd_kernel, metadata !"vectoradd_kernel", i32 1}
```



Código PTX para o vectoradd-kernel

```
//
// Generated by LLVM NVPTX Back-End
//

.version 3.1
.target sm_20
.address_size 64

// .globl       vectoradd_kernel
                                        // @vectoradd_kernel
.visible .entry vectoradd_kernel(
                .param .u64 vectoradd_kernel_param_0,
                .param .u64 vectoradd_kernel_param_1,
                .param .u64 vectoradd_kernel_param_2
                )
{
        .reg .f32       %f<4>;
        .reg .s32       %r<5>;
        .reg .s64       %rl<8>;

// BB#0:                                // %entry
        ld.param.u64    %rl1, [vectoradd_kernel_param_0];
        mov.u32         %r1, %tid.x;
        ld.param.u64    %rl2, [vectoradd_kernel_param_1];
        mov.u32         %r2, %ntid.x;
        ld.param.u64    %rl3, [vectoradd_kernel_param_2];
        mov.u32         %r3, %ctaid.x;
        mad.lo.s32      %r4, %r3, %r2, %r1;
        mul.wide.s32    %rl4, %r4, 4;
        add.s64         %rl5, %rl3, %rl4;
        add.s64         %rl6, %rl2, %rl4;
        add.s64         %rl7, %rl1, %rl4;
        ld.f32  %f1, [%rl7];
        ld.f32  %f2, [%rl6];
        add.f32         %f3, %f1, %f2;
        st.f32  [%rl5], %f3;
        ret;
}
```