Definição Semântica de Blocos `Everywhere` para Programação SIMD

Rubens Emílio Alves Moreira[1], Sylvain Collange[2] e Fernando Magno Quintão Pereira[1]

[1] UFMG – Avenida Antônio Carlos, 6627, 31.270-010, Belo Horizonte, MG, Brazil
`{rubens,fernando}@dcc.ufmg.br`

[2] INRIA -– Centre de recherche Rennes, Bretagne Atlantique, Campus de Beaulieu, 35042, RENNES, France `sylvain.collange.inria.fr`

Resumo Placas gráficas programáveis (GPUs) e alguns processadores, como Xeon Phi e Skylake Xeon, suportam o modelo de execução SIMD (*Single Instruction, Multiple Data*). Apesar do interesse crescente, as linguagens conhecidas para a programação de máquinas SIMD, a saber, C para CUDA e OpenCl, possuem semântica SPMD (*Single Program, Multiple Data*), permitindo divergências de execução entre *threads*. Para usar diretivas SIMD, em geral, todas as *threads* de um grupo de processamento devem estar ativas – restrição contra-intuitiva sob linguagens SPMD. Em particular, tal restrição dificulta a implementação correta de *paralelismo dinâmico*: a capacidade de iniciar um novo grupo de processamento a partir de um bloco de *threads* já em execução. As implementações atuais de paralelismo dinâmico têm custos com alocação de recursos e escalonamento. Este trabalho propõe um forma de implementar paralelismo dinâmico que é mais eficiente e simples de ser usada. Com tal propósito, resgata-se uma antiga construção de linguagens SIMD: os blocos `Everywhere`. A principal contribuição deste trabalho é a descrição formal da semântica dessa construção. Para tanto, cria-se uma máquina SIMD abstrata, e mostra-se como blocos *everywhere* podem ser implementados sobre ela. Tal máquina encontra-se hoje disponível para uso público, via uma implementação Prolog, e pode ser usada para simular outros tipos de construções no mundo SIMD.

1 Introdução

Entre os anos 70 e 80 houve um grande esforço de pesquisa sobre o modelo de execução SIMD [3]. De tal esforço surgiram técnicas para aumentar a programabilidade [8] e a eficiência [3] desse paradigma, no qual uma mesma instrução é aplicada simultaneamente sobre uma lista de dados diferentes. Dentre as contribuições daquela época destaca-se uma vasta família de linguagens de programação voltadas para arquiteturas SIMD [1,3,4,13,14,15,20]. Os altos custos, e as dificuldades tecnológicas relacionadas à construção de máquinas SIMD levaram à queda de sua popularidade nos anos 90 [18]. Entretanto, desde 2005, o

[3] Sigla do termo inglês *Single Instruction, Multiple Data*.

surgimento de placas gráficas (GPU) programáveis vem trazendo esse modelo novamente para o centro de atenção da comunidade de linguagens de programação [10,19,18]. Atualmente GPUs são usadas para fins tão diversos quanto sequenciamento genético [23] ou roteamento de pacotes [17].

A despeito de todo o esforço feito a partir dos anos 70, existe ainda uma carência de modelos de programação para arquiteturas SIMD. Tal carência deve-se em muito ao surgimento de novas demandas trazidas pela popularização de placas de processamento gráfico. Dentre tais demandas, destaca-se a necessidade de suporte a formas de *paralelismo dinâmico* no modelo SIMD. Paralelismo dinâmico é a capacidade que alguns tipos de hardware moderno possuem de lançar novas *threads* a partir de *threads* já em execução. Essa capacidade facilita o desenvolvimento de programas capazes de lidar com trabalho menos regular que as tarefas tipicamente resolvidas por máquinas SIMD tradicionais.

Entretanto, o paralelismo dinâmico, embora útil e desejável, traz um novo desafio para a implementação de compiladores: como lidar com divergências? Divergências de execução ocorrem em máquinas SIMD devido a desvios condicionais de fluxo de execução. Uma vez que todas as *threads* executam sempre as mesmas instruções, em vistas de desvios condicionais, algumas delas terão de esperar enquanto outras realizam trabalho. Na ausência de paralelismo dinâmico, divergências simplesmente diminuem o desempenho do *hardware* SIMD. Contudo, na presença dele, divergências podem levar a programas incorretos. Em particular, o resultado de invocar novas *threads* em uma região divergente de código é indefinido na linguagem de programação C para CUDA.

O objetivo deste artigo é propor uma forma de lidar com esse problema. Com tal intuito, recorre-se a uma antiga construção sintática de linguagens voltadas ao modelo de execução SIMD: os blocos `everywhere` [12,16,22] Este artigo mostra, na seção 2, como blocos `everywhere` podem ser usados para abilitar paralelismo dinâmico. A partir dessa observação, ele propõe, na seção 3, uma semântica para a implementação de blocos *everywhere* em uma arquitetura SIMD moderna, típica de GPUs. Essa extensão contém instruções capazes de marcar regiões de código em que *threads* dormentes podem ser ativadas para realizar trabalho. Para preservar a corretude do programa, o estado dessas *threads* é salvo, mediante uma troca de contexto. Findo seu trabalho, tais *threads* voltam ao seu estado original, seja ele ativo ou dormente. A descrição da semântica operacional vista na seção 3 é a primeira formalização de uma máquina SIMD com blocos `everywhere` descrita na literatura. Essa semântica vem acrescida de exemplos que ilustram as possibilidades do novo conjunto de instruções.

A semântica proposta na seção 3 foi implementada em Prolog. Essa máquina abstrata descreve o comportamento dos principais eventos que podem ocorrer durante a execução de programas SIMD: divergências, reconvergências e o lançamento de novas *threads* a partir de *threads* já em execução. Diversos programas SIMD já foram simulados nessa máquina, que encontra-se hoje disponível publicamente. A partir de seu uso é possível testar novas instruções e comportamentos, no contexto de uma arquitetura que possui um buscador de instruções, e várias CPUs abstratas.

2 O Problema de Implementar Paralelismo Dinâmico

A fim de motivar as idéias introduzidas neste artigo, será usado o exemplo de programa visto na figura 1 (a). Essa figura descreve um *kernel* implementado em CUDA. Esse *kernel* realiza uma cópia de dados, similar à função `memcpy`, muito usada na biblioteca padrão C. Porém, aqui tem-se a semântica SIMT [9]: o programa da figura 1 (a) será instanciado uma vez para cada *thread* em um bloco CUDA. Um bloco de *threads* CUDA é um conjunto maximal de *threads* que podem ser sincronizadas, e que podem compartilhar memória.

(a)
```
1  __global__ void memcpy_words_a(int *dest, const int *src, size_t n)
2  {
3      int tid = threadIdx.x;
4      for(int i = tid; i < n; i += blockDim.x) {
5          dest[i] = src[i];
6      }
7  }
```

(b)
```
1  __global__ void my_kernel(int *dest, const int *src, size_t n) {
2      int tid = threadIdx.x;
3      if (tid % 2) {
4          memcpy_words_a(dest[tid], src[tid], n/blockDim.x);
5      } else {
6          ...
6      }
7  }
```

(c)
```
1  __device__ void memcpy_words_c(int *dest, const int *src, size_t n)
2  {
3      for(int i = 0; i < n; i++) {
4          dest[i] = src[i];
5      }
6  }
```

(d)
```
1  __device__ void memcpy_words_d(int *dest, const int *src, size_t n,
                                  int mask) {
2    // Serialize each thread
3    for (int j = 0; j < warp_size; j++) {
4      if (mask & (1 << j)) { // Was thread j active?
5        // Get inputs of thread j
6        int * mydest = __shfl(dest, j);
7        int * mysrc = __shfl(src, j);
8        size_t myn = __shfl(n, j);
9        // Vectorize the memcpy loop
10       for(int i = laneid; i < myn; i += warp_size) {
11         mydest[i] = mysrc[i];
12       }
13     }
14   }
15 }
```

Figura 1: Código que implementa cópia de dados entre arranjos em CUDA. Todas essas funções executam em uma GPU. Funções marcadas como `global` podem ser invocadas a partir de CPUs, ou de GPUs que suportam paralelismo dinâmico. As outras funções são invocadas da GPU.

A função `memcpy_words_a` normalmente é invocada a partir de um programa que executa na CPU. Uma vez disparada, essa função é escalonada para ser executada na placa de processamento gráfico – daqui por diante chamada *dispositivo*. Há situações, entretanto, em que é interessante que essa função seja invocada dentro do próprio dispositivo, a partir de outro *kernel* já em execução, como no trecho de código visto na figura 1 (b). Há várias formas de permitir tal invocação, e no restante desta seção serão discutidas cada uma delas.

A primeira dentre tais maneiras consiste em ré-implementar `memcpy_words_a` como uma função de dispositivo. Essa nova versão é mostrada na figura 1 (c). Todavia, essa abordagem possui uma séria desvantagem. Acessos à memória em CUDA são mais eficientes se eles podem ser feitos de forma *agrupada* por *threads* em uma mesma frente de execução (*warp*). Um *warp* é um conjunto de *threads* que executam em modo SIMD: todas elas processam simultaneamente instruções iguais. Acessos de *threads* próximas, isto é, que estão organizadas no mesmo *warp*, são ditos agrupados quando eles fazem referências a endereços fisicamente próximos. Os acessos na figura 1 (a) são todos agrupados: a *thread* de ID (`tid`) 2 lê a posição de memória adjacente àquela lida pela *thread* de ID 1, por exemplo. Tal não é o caso na figura 1 (c). Cada *thread* será chamada com vetores `dst` e `src` bastante diferentes. Além disso, a possibilidade delas receberem valores de `n` diferentes pode levar a divergências de controle. A opção vista na figura 1 (c) é, portanto, muito ineficiente, podendo levar a tempos de execução até 10 vezes mais lentos que `memcpy_words_a`, vista na figura 1 (a).

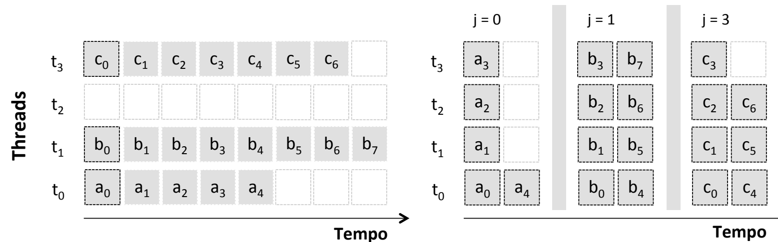

Figura 2: O diagrama à esquerda representa a execução do código da figura 1(c). Apesar de simples, esta versão tem dois problemas principais: os acessos a segmentos diferentes de memória não explora localidade de referência; e a utilização de recursos computacionais não é ótima. Já o diagrama à direita corresponde ao código da figura 1(d): os valores de $j = 0, 1, 3$ indicam a qual thread cada tarefa pertence. Esta versão é eficiente graças ao uso de instruções warp-síncronas.

A figura 2 (Esquerda) ilustra esse problema. Endereços do tipo $x_0, x_1, x_2, \ldots, x_n$ são contíguos em memória. Logo, caso as diferentes *threads* acessem essas posições – simultaneamente – então tem-se um acesso agrupado à memória. Na figura 2 (Esquerda), acessos simultâneos são mostrados em colunas. Cada letra,

```
1  __global__ void my_kernel(int *dest, const int *src, size_t n) {
2    int tid = threadIdx.x;
3    if (tid % 2) {
4      everywhere(int mask) {
5        memcpy_words_f(dest[tid], src[tid], n/blockDim.x, mask);
6      }
6    } else {
7    ...
9    }
10 }
```

Figura 3: Programa que mostra como blocos `everywhere` lidam com o problema de divergências, visto na figura 1 (b).

i.e., a, b e c denota posições muito distantes em memória. Assim, vê-se na figura que cada acesso simultâneo não pode ser agrupado: no instante i, a *thread* 0 irá ler a posição a_i, a *thread* 1 irá ler a posição b_i, a *thread* 2 não estará ativa, e a *thread* 3 irá ler a posição c_i. No jargão CUDA, diz-se que ocorrem *divergências de acesso à memória*. Note também que algumas *threads* podem terminar seu trabalho primeiro. Nesse exemplo, a *thread* 0 termina seu trabalho antes da *thread* 1. Essas divergências de controle causam perda de desempenho, tanto quanto as divergências de acesso à memória, pois elas levam à existência de *threads* ociosas.

Quando a linguagem C para CUDA surgiu, um *kernel* como aquele visto na figura 1 somente poderia ser invocado a partir do espaço de endereçamento da CPU. Contudo, a introdução de paralelismo dinâmico em C para CUDA passou a permitir que a função `memcpy_words_a` pudesse ser invocada a partir de outro *kernel*. Esse tipo de invocação é visto na figura 1 (b). Nesse novo exemplo, `memcpy_words_a` é invocada uma vez por cada *thread* ativa. Cada instância dessa invocação irá disparar novas *threads*. Essa abordagem é elegante, porém ela possui um alto custo em termos de desempenho, que advém do fato de um novo *kernel* ser escalonado para execução na placa gráfica.

A abordagem vista na figura 1 (d) é uma terceira tentativa de invocar a função `memcpy_words_a` a partir de um *kernel* em execução na placa gráfica. Nesse caso, `memcpy_words_d`, a nova versão de `memcpy_words_a`, foi ré-escrita para levar em consideração o fato de que *threads* em um mesmo *warp* executam em modo SIMD. A instrução `shfl(var, lane)` permite que a *thread* de identificador `tid` leia o valor de `var` que é mantido pela *thread* de identificador `tid + lane`. A função `memcpy_words_d` é uma maneira diferente de vetorizar a tarefa original, de copiar dados entre arranjos. Ela é eficiente: acessos à memória são agrupados para *threads* em um mesmo *warp*; contudo, ela possui uma séria limitação. A figura 2 (Direita) ilustra o funcionamento de `memcpy_words_d`. No instante $\texttt{j} = 0$, todas as *threads* lêem dados a_i. Em seguida, no instante $\texttt{j} = 1$, as *threads* passam a ler os endereços b_i, e assim por diante.

O comportamento de `shfl` é indeterminado caso essa instrução seja usada para ler dados de uma *thread* que, devido a divergências, encontra-se inativa. Posto que divergências são um fenômeno comum em GPUs, dificilmente essa versão explicitamente vetorizada de `memcpy_words_a` poderia ser usada na prática.

A figura 1 (b) ilustra uma situação em que divergências de controle poderiam levar a uma invocação incorreta do *kernel* visto na figura 1 (d). O programa da figura 1 (b) contém comportamento claramente divergente: o teste na linha 3 será verdadeiro para *threads* de identificador ímpar, e falso para as outras *threads*. Assim, o resultado da instrução `shfl` sobre as *threads* que não estiverem ativas na linha 4 da figura 1 (b) é indefinido. Esse artigo propõe uma primitiva de baixo nível para permitir que a chamada na figura 1 (b) possa ser realizada corretamente. A figura 3 ilustra o uso dessa primitiva. Nossa semântica busca assegurar uma invocação eficiente, obtendo assim os mesmos benefícios de desempenho que aqueles obtidos pelo uso da função vista na figura 1 (d).

A figura 3 mostra onde a primitiva `everywhere` deve ser inserida para permitir a invocação dinâmica de `memcpy_words_d`. Essa inserção pode ser feita por um compilador, ou pelo programador. Esse ponto: a correta delimitação de regiões *everywhere*, não é parte do escopo desse trabalho, o qual atem-se somente a semântica dessa construção. Note que blocos `everywhere`, conforme concebidos neste trabalho, recebem uma máscara que delimita as *threads* ativas em um *warp*. Isso permite que um *kernel* invocado dentro de uma região `everywhere` use somente dados de *threads* ativas. A figura 1 (d) ilustra esse ponto: as instruções *shuffle* são aplicadas somente sobre dados de *threads* originalmente ativas quando `memcpy_words_d` foi invocada.

3 Semântica

Blocos `everywhere` existem em diversas linguagens de programação, mas ainda não foram definidos formalmente sob o paradigma SIMD. As linguagens para plataformas SIMD possuem semântica SPMD, e organizam threads em grupos, ou *warps*, em que são permitidas divergências internas de fluxo de controle. Isto dificulta formalizar a semântica desses blocos, pois é preciso tratar eventuais divergências. Mais ainda, após executar um bloco `everywhere`, a menos de escritas em memória, o estado das *threads* deve ser restaurado àquele anterior à execução do bloco. E portanto, dados do fluxo de execução das *threads* devem ser armazenados antes da chamada ao paralelismo dinâmico.

3.1 Linguagem e Máquina SIMD abstrata

Linguagens possuem diversas estruturas que alteram o fluxo de execução de programas, como blocos de iteração, condicionais e chamadas de função. No entanto, antes de definir a semântica de tais estrutras, é preciso definir noções básicas de execução da linguagem. Neste trabalho, utilizamos como base a linguagem para máquina abstrata μ-SIMD, descrita em [2] e [6], e estendida por Coutinho et al. [5]. É a partir de μ-SIMD que descrevemos a semântica de blocos `everywhere` no domínio SIMD. A figura 4 ilustra o estado de um programa μ-SIMD.

Unidades de processamento (*threads*) são identificadas univocamente por um natural t. σ e Σ são mapas de nomes de variáveis para valores inteiros, e representam, respectivamente, seções de memória local e compartilhada. Apesar da

Rótulos (L)	$::= \; l \in \mathbb{N}$	*Threads* Ativas	$\Theta \subset \mathbb{N}$
Constantes (C)	$::= \; c \in \mathbb{N}$	Memória Local	$\sigma \subset V \mapsto \mathbb{Z}$
Variáveis (V)	$::= \; T_{id} \cup \{v_1, v_2, \dots\}$	Banco de Memória Local ...	$\beta \subset T_{id} \mapsto \sigma$
Operandos ($V \cup C$)	$::= \; \{o_1, o_2, \dots\}$	Memória Compartilhada ...	$\Sigma \subset \mathbb{N} \mapsto \mathbb{Z}$
Instruções (I)	$::= \; \mu$-SIMD	Pilha de Sincronização	$\Pi \subset (L \times \Theta \times L \times \Theta \times \Pi)$
		Pilha de Contexto	$\Lambda \subset (\Theta \times \Pi \times \Lambda)$
		Programa	$P \subset L \mapsto I$
		Contador de Programa	$pc \in \mathbb{N}$

Figura 4: O estado da máquina μ-SIMD é uma sétupla $M(\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc)$. Θ indica o conjunto de *threads* ativas; cada *thread* possui uma memória local σ, acessível a partir do banco de memória β, e acessa uma memória compartilhada Σ. A pilha de sincronização Π abstrai divergências de fluxo de controle; enquanto a pilha de contexto Λ, adicionada ao modelo já existente [5], possibilita o paralelismo dinâmico. Por fim, o programa P é uma sequência de instruções, e pc indica qual a última instrução processada.

memória local ser individual a cada unidade de processamento, as funções aqui apresentadas possuem o mesmo domínio, e portanto, $v \in \sigma$ denota um vetor de variáveis, cada uma privada a uma thread específica. Além disso, memórias locais σ são armazenada em um banco de memória β, o que simplifica a definição de algumas instruções *warp*-síncronas, como será visto na seção 3.2.

Programas são mapas de rótulos para instruções. Durante a execução, um conjunto Θ de *threads* não divergentes (unidades ativas), executa as instruções, resultando em trincas (Θ, β, Σ), ou seja, produzindo novos conjuntos Θ de *threads* ativas, e atualizando bancos de memória local β e de memória compartilhada Σ. O contador de programa pc identifica a próxima instrução a ser executada. Para que seja possível simular a convergência de *threads*, a máquina possui um pilha de sincronização Π. Cada elemento em Π é uma tupla $(l_{id}, \Theta_{done}, l_{next}, \Theta_{todo})$, e indica o ponto em que *threads* divergentes devem sincronizar. Tuplas são empilhadas sempre que ocorre divergência no fluxo de controle: l_{id} indica o desvio condicional que causou a divergência; Θ_{done} registra quais unidades de processamento já atingiram a barreira de sincronização; e Θ_{todo} quais *threads* aguardam entrar em execução. Por fim, l_{next} aponta a posição em que *threads* de Θ_{todo} devem prosseguir com a execução.

Nosso modelo μ-SIMD possui ainda uma pilha de contexto Λ, que habilita a utilização de blocos `everywhere`. Considere um conjunto de *threads* e um programa fonte. Para utilizar a estrutura de controle `everywhere` no domínio SIMD, precisamos garantir que, independente do estado em que se encontram as *threads*, é sempre possível ativar todas essas unidades de processamento e executar a sequência de código interna ao bloco `everywhere`, mantendo o fluxo de execução válido. Ao entrar no bloco, as unidades de processamento estarão sob novo contexto, e ao sair devem ter o contexto original restaurado. Mais ainda, blocos `everywhere` podem ser invocados recursivamente, e portanto o contexto deve ser empilhado/desempilhado a cada invocação/término de execução de um bloco. Assim, adicionamos à máquina abstrata uma pilha de contexto Λ, que armazena o estado (ativo/inativo) das *threads*, e o estado da pilha de sincroni-

zação logo antes de ativar todas as *threads*. Denotamos por Θ_{all} o conjunto de todas as *threads* de um *warp*. A figura 5 apresenta a sintaxe das instruções da máquina μ-SIMD.

desvio se zero / diferente de zero	`bz` / `bnz` v, l
desvio se thread T_{id} invocou `everywhere`	`branch_mask` T_{id}, l
desvio incondicional	`jump` l
escrita em memória compartilhada	$\uparrow v_x = v$
leitura de memória compartilhada	$v =\downarrow v_x$
incremento atômico	$v \stackrel{a}{\leftarrow} v_x + 1$
binária	$v_1 = o_1 + o_2$
multiplicação binária	$v_1 = o_1 \times o_2$
outras operações binárias	$v_1 = o_1 \oplus o_2$
cópia simples	$v = o$
barreira de sincronização	`sync`
desliga a máquina	`stop`
começo de bloco `everywhere`	`everywhere`
fim de bloco `everywhere`	`end_everywhere`

Figura 5: Conjunto de instruções μ-SIMD. Aumentamos a expressividade da linguagem apresentada por Coutinho et al. [5], dando suporte a blocos `everywhere`. As instruções `everywhere` e `end_everywhere` denotam, respectivamente, o início e término dos blocos de paralelismo dinâmico. E, a instrução `branch_mask` T_{id}, l permite que o programador realize desvios com base em quais *threads* participaram da invocação do bloco `everywhere`.

3.2 Semântica Operacional

Descrevemos a máquina abstrata μ-SIMD com suporte a blocos `everywhere` utilizando semântica operacional de passo longo. A semântica de instruções de fluxo de controle do programa usa as funções auxiliares apresentadas na figura 6, assim como as regras da figura 7. Por exemplo, de acordo com a regra Sp, um programa termina se $P[pc] =$ `stop`. Condicionais possuem semântica um pouco mais elaborada: ao atingir uma instrução `bz` v, l, avaliamos o valor de v a partir da memória local de cada thread ativa. Seja $\sigma = \beta[t]$ a memória local de uma thread t. Se $\sigma(v) \neq 0$ para todas as unidades de processamento ativas, então a regra Bf move o fluxo de controle para a próxima instrução, i.e., $pc + 1$. Analogamente, se $\sigma(v) = 0$ para toda thread ativa, então usamos a regra Bt para desviar para a instrução $P[l]$. Contudo, caso *threads* recebam valores diferentes de v, o desvio é considerado divergente, e a regra Bd indica que devemos executar primeiro as *threads* que desviaram para o bloco "else" do condicional; as outras *threads* são mantidas na pilha de sincronização, para que sejam executadas posteriormente.

A função auxiliar **push**, da figura 6, atualiza a pilha de sincronização Π. Mesmo regras para desvios não-divergentes atualizam a pilha de sincronização:

$\mathbf{split}(\Theta, \beta, v) = (\Theta_0, \Theta_n)$ **where**

$\Theta_0 = \{t \mid t \in \Theta \textbf{ and } \beta[t] = \sigma_t \textbf{ and } \sigma_t[v] = 0\}$

$\Theta_n = \{t \mid t \in \Theta \textbf{ and } \beta[t] = \sigma_t \textbf{ and } \sigma_t[v] \neq 0\}$

$\mathbf{push}([], \Theta_n, pc, l) = [(pc, [], l, \Theta_n)]$

$\mathbf{push}((pc', [], l', \Theta'_n) : \Pi, \Theta_n, pc, l) = \Pi' \textbf{ if } pc \neq pc'$

$\textbf{where } \Pi' = (pc, [], l, \Theta_n) : (pc', [], l', \Theta'_n) : \Pi$

$\mathbf{push}((pc, [], l, \Theta'_n) : \Pi, \Theta_n, pc, l) = (pc, [], l, \Theta_n \cup \Theta'_n) : \Pi$

Figura 6: Funções auxiliares usadas na definição de μ-Simd. A função **split** é um filtro, e cria dois conjuntos de *threads* divergentes (Θ_0 e Θ_n). Já a função **push** atualiza a pilha de sincronização Π sob divergências e convergências.

ao atingir uma barreira (instrução `sync`), a avaliação das regras continua válida, pois é possível desempilhar entradas de Π. Por exemplo, a partir da regra Ss, se alcançarmos uma barreira de sincronização com um grupo Θ_n de *threads* que aguardam para serem executadas, podemos continuar a execução dessa *threads* no bloco "then" do desvio condicional. E as unidades de processamento anteriormente ativas são desativadas e mantidas no conjunto Θ_{done}. Por fim, se atingirmos uma barreira sem que haja qualquer thread esperando para ser executada, usamos a regra Sp para sincronizar o conjunto Θ_{done} com o conjunto Θ de *threads* ativas. A execução do programa continua a partir da instrução que sucede a barreira. Para evitar *deadlocks*, assumimos que um condicional e sua respectiva barreira de sincronização configuram uma região de *única-entrada-única-saída* sobre o grafo de fluxo de controle do programa [7, p.329].

Blocos `everywhere` são definidos de acordo com as regras Eb e Ee. Ao encontrar a instrução `everywhere`, adicionamos à pilha de contexto Λ o par (Θ, Π), ou seja, o conjunto de *threads* ativas e a pilha de sincronização corrente. Em seguida, a execução prossegue de acordo com as definições da linguagem μ-SIMD. Ao atingir uma instrução `end_everywhere`, ignoramos o conjunto corrente de *threads* ativas: assim como definido na regra Ee, basta que desempilhemos um par (Θ, Π), com o qual continuaremos a executar o programa. Dado que utilizamos uma pilha para armazenar o contexto, por indução, é possível aninhar um número arbitrário de blocos `everywhere`. Note ainda que não é necessário armazenar o contador de programa corrente: após executar o bloco `everywhere`, as *threads* ativas saltam para a instrução que sucede a tag `end_everywhere`; e as inativas serão eventualmente ativadas na posição correta, determinada pelo conteúdo da pilha de sincronização.

Instruções que não mudam diretamente o fluxo de controle são avaliadas a partir da regra It. A figura 8 mostra a semântica de tais instruções. A tupla $(t, \beta, \Sigma, \Theta_{mask}, \iota)$ denota a execução de uma instrução ι por uma unidade de processamento t. Todas as *threads* ativas executam a mesma instrução ao mesmo tempo, o que é garantido pela regra Tl, quando mostramos que a ordem em que

(Sp) $$\frac{P[pc] = \texttt{stop}}{(\Theta, \beta, \Sigma, \emptyset, \Lambda, P, pc) \rightarrow (\Theta, \beta, \Sigma)}$$

(Bt) $$\frac{P[pc] = \texttt{bz}\ v, l \qquad \textbf{split}(\Theta, \beta, v) = (\Theta, \emptyset) \quad \textbf{push}(\Pi, \emptyset, pc, l) = \Pi' \qquad (\Theta, \beta, \Sigma, \Pi', \Lambda, P, l) \rightarrow (\Theta', \beta', \Sigma')}{(\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc) \rightarrow (\Theta', \beta', \Sigma')}$$

(Bf) $$\frac{P[pc] = \texttt{bz}\ v, l \qquad \textbf{split}(\Theta, \beta, v) = (\emptyset, \Theta) \quad \textbf{push}(\Pi, \emptyset, pc, l) = \Pi' \qquad (\Theta, \beta, \Sigma, \Pi', \Lambda, P, pc+1) \rightarrow (\Theta', \beta', \Sigma')}{(\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc) \rightarrow (\Theta', \beta', \Sigma')}$$

(Bd) $$\frac{P[pc] = \texttt{bz}\ v, l \qquad \textbf{split}(\Theta, \beta, v) = (\Theta_0, \Theta_n) \quad \textbf{push}(\Pi, \Theta_n, pc, l) = \Pi' \qquad (\Theta_0, \beta, \Sigma, \Pi', \Lambda, P, pc+1) \rightarrow (\Theta', \beta', \Sigma')}{(\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc) \rightarrow (\Theta', \beta', \Sigma')}$$

(Ba) $$\frac{P[pc] = \texttt{branch_mask}\ T_{id}, l \quad T_{id} \in \Theta' \qquad (\Theta, \beta, \Sigma, \Pi, (\Theta', \Pi') : \Lambda, P, l) \rightarrow (\Theta'', \beta', \Sigma')}{(\Theta, \beta, \Sigma, \Pi, (\Theta', \Pi') : \Lambda, P, pc) \rightarrow (\Theta'', \beta', \Sigma')}$$

(Bi) $$\frac{P[pc] = \texttt{branch_mask}\ T_{id}, l \quad T_{id} \notin \Theta' \qquad (\Theta, \beta, \Sigma, \Pi, (\Theta', \Pi') : \Lambda, P, pc+1) \rightarrow (\Theta'', \beta', \Sigma')}{(\Theta, \beta, \Sigma, \Pi, (\Theta', \Pi') : \Lambda, P, pc) \rightarrow (\Theta'', \beta', \Sigma')}$$

(Ss) $$\frac{P[pc] = \texttt{sync} \qquad \Theta_n \neq \emptyset \qquad (\Theta_n, \beta, \Sigma, (pc', \Theta_0, l, \emptyset) : \Pi, \Lambda, P, l) \rightarrow (\Theta', \beta', \Sigma')}{(\Theta, \beta, \Sigma, (pc', \emptyset, l, \Theta_n) : \Pi, \Lambda, P, pc) \rightarrow (\Theta', \beta', \Sigma')}$$

(Sp) $$\frac{P[pc] = \texttt{sync} \qquad (\Theta_n, \beta, \Sigma, (_, \emptyset, _, \Theta_0) : \Pi, \Lambda, P, pc+1) \rightarrow (\Theta', \beta', \Sigma')}{(\Theta_0 \cup \Theta_n, \beta, \Sigma, \Pi, \Lambda, P, pc) \rightarrow (\Theta', \beta', \Sigma')}$$

(Jp) $$\frac{P[pc] = \texttt{jump}\ l \qquad (\Theta, \beta, \Sigma, \Pi, \Lambda, P, l) \rightarrow (\Theta', \beta', \Sigma')}{(\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc) \rightarrow (\Theta', \beta', \Sigma')}$$

(Eb) $$\frac{P[pc] = \texttt{everywhere} \qquad (\Theta_{all}, \beta, \Sigma, \emptyset, (\Theta, \Pi) : \Lambda, P, pc+1) \rightarrow (\Theta', \beta', \Sigma')}{(\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc) \rightarrow (\Theta', \beta', \Sigma')}$$

(Ee) $$\frac{P[pc] = \texttt{end_everywhere} \qquad (\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc+1) \rightarrow (\Theta', \beta', \Sigma')}{(_, \beta, \Sigma, \emptyset, (\Theta, \Pi) : \Lambda, P, pc) \rightarrow (\Theta', \beta', \Sigma')}$$

(It) $$\frac{\begin{array}{c} P[pc] = \iota \\ \iota \notin \{\texttt{stop}, \texttt{bnz}, \texttt{bz}, \texttt{branch_mask}, \texttt{sync}, \texttt{jump}, \texttt{everywhere}, \texttt{end_everywhere}\} \\ (\Theta, \beta, \Sigma, \Theta_{mask}, \iota) \rightarrow (\beta', \Sigma') \\ (\Theta, \beta', \Sigma', \Pi, (\Theta_{mask}, \Pi') : \Lambda, pc+1) \rightarrow (\Theta', \beta'', \Sigma'') \end{array}}{(\Theta, \beta, \Sigma, \Pi, (\Theta_{mask}, \Pi') : \Lambda, P, pc) \rightarrow (\Theta', \beta'', \Sigma'')}$$

Figura 7: Semântica operacional μ-Simd: instruções de fluxo de controle. Por simplicidade, quando duas hipóteses são válidas, usamos aquela mais ao topo. Além disso, não são apresentadas regras de avaliação para `bnz`, uma vez que são similares às regras de `bz`.

$$(\textsc{Mm})\quad \frac{\Sigma(v) = c}{\Sigma \vdash v = c} \qquad (\textsc{Mt})\quad t, \beta \vdash \mathtt{T}_{id} = t \qquad (\textsc{Mv})\quad \frac{v \neq \mathtt{T}_{id} \quad \beta[t] = \sigma_t \quad \sigma_t(v) = c}{t, \beta \vdash v = c}$$

$$(\textsc{Tl})\quad \frac{(t, \beta, \Sigma, \Theta_{mask}, \iota) \rightarrow (\sigma_t, \Sigma') \qquad (\Theta, \beta \setminus [\beta[t] \mapsto \sigma_t], \Sigma', \Theta_{mask}, \iota) \rightarrow (\beta", \Sigma")}{(\{t\} \cup \Theta, \beta, \Sigma, \Theta_{mask}, \iota) \rightarrow (\beta", \Sigma")}$$

$$(\textsc{Ct})\quad \frac{\beta[t] = \sigma_t}{(t, \beta, \Sigma, _, v = c) \rightarrow (\sigma_t \setminus [v \mapsto c], \Sigma)}$$

$$(\textsc{As})\quad \frac{t, \beta \vdash v' = c \qquad \beta[t] = \sigma_t}{(t, \beta, \Sigma, _, v = v') \rightarrow (\sigma_t \setminus [v \mapsto c], \Sigma)}$$

$$(\textsc{Ld})\quad \frac{t, \beta \vdash v_x = c_x \qquad \beta[t] = \sigma_t \qquad \Sigma \vdash c_x = c}{(t, \beta, \Sigma, _, v = \downarrow v_x) \rightarrow (\sigma_t \setminus [v \mapsto c], \Sigma)}$$

$$(\textsc{St})\quad \frac{t, \beta \vdash v_x = c_x \qquad t, \beta \vdash v = c\beta[t] = \sigma_t}{(t, \beta, \Sigma, _, \uparrow v_x = v) \rightarrow (\sigma_t, \Sigma \setminus [c_x \mapsto c])}$$

$$(\textsc{At})\quad \frac{t, \beta \vdash v_x = c_x \qquad \Sigma \vdash c_x = c \qquad \beta[t] = \sigma_t \qquad c' = c + 1}{(t, \beta, \Sigma, _, v \overset{a}{\leftarrow} v_x + 1) \rightarrow (\sigma_t \setminus [v \mapsto c'], \Sigma \setminus [c_x \mapsto c'])}$$

$$(\textsc{Bp})\quad \frac{t, \beta \vdash v_2 = c_2 \qquad t, \beta \vdash v_3 = c_3 \qquad \beta[t] = \sigma_t \qquad c_1 = c_2 \oplus c_3}{(t, \beta, \Sigma, _, v_1 = v_2 \oplus v_3) \rightarrow (\sigma_t \setminus [v_1 \mapsto c_1], \Sigma)}$$

$$(\textsc{Sv})\quad \frac{\begin{array}{c} t, \beta \vdash v_1 = c_1 \qquad t, \beta \vdash v_{lane} = c_{lane} \\ \beta[t] = \sigma_t \qquad c_{lane} \in \Theta_{mask} \qquad \beta[c_{lane}] = \sigma_{lane} \qquad \sigma_{lane}(v_1) = c_2 \end{array}}{(t, \beta, \Sigma, \Theta_{mask}, \mathtt{shfl}(v_1, v_{lane})) \rightarrow (\sigma_t \setminus [v_1 \mapsto c_2], \Sigma)}$$

$$(\textsc{Si})\quad \frac{\begin{array}{c} t, \beta \vdash v_1 = c_1 \qquad t, \beta \vdash v_{lane} = c_{lane} \\ \beta[t] = \sigma_t \qquad c_{lane} \notin \Theta_{mask} \end{array}}{(t, \beta, \Sigma, \Theta_{mask}, \mathtt{shfl}(v_1, v_{lane})) \rightarrow (\sigma_t \setminus [v_1 \mapsto _], \Sigma)}$$

Figura 8: Semântica operacional μ-Simd: instruções aritméticas e de leitura e escrita de dados. Estas instruções são avaliadas a partir da avaliação da regra It, apresentada na figura 7.

threads diferentes processam uma instrução ι é irrelevante. Assim, uma instrução como $v = c$ faz com que toda thread ativa atribua o inteiro c à sua variável local v. Θ_{mask} é o conjunto de *threads* ativas no momento em que o bloco `everywhere` foi invocado, e é utilizado por algumas instruções *warp*-síncronas. A instrução $\mathtt{shfl}(v_1, v_{lane})$ utiliza o conjunto Θ_{mask} para saber quais *threads* podem ter os segmentos de memória lidos. Por definição, essa instrução lê o valor da variável v_1 na memória local à *thread* v_{lane}. Caso $v_{lane} \in \Theta_{mask}$, o valor lido é atribuído

à variável v_1, na memória local à *thread* t; caso contrário, o valor atribuído é indefinido. O restante das regras da figura 8 independem do paralelismo presente no modelo μ-SIMD, i.e., a semântica dessas regras é determinada com base em apenas uma unidade de processamento.

Usamos a notação $f[a \mapsto b]$ para atualizar uma função f, ou seja, $\lambda x.x = a \; ? \; b : f(x)$. A regra CT descreve a atribuição de uma constante a uma variável. Analogamente, a regra AS define a cópia do valor de uma variável v' para outra variável v. A regra LD mostra o leitura de dados da memória compartilhada Σ para uma variável v, local à unidade de processamento. Nessa regra, o valor da variável v_x é usado para indexar a memória Σ. Por fim, escritas são definidas de acordo com a regra ST. Uma instrução como $\uparrow v_x = v$, por exemplo, copia o valor de v na posição de memória de Σ endereçada pelo valor de v_x.

A instrução de escrita pode provocar condição de corrida, caso duas unidades de processamento tentem escrever dados diferentes sobre o mesmo endereço da memória compartilhada Σ. Neste caso, o resultado é indefinido, como mostra a regra TL. Garantimos atomicidade por meio da operação $v \stackrel{a}{\leftarrow} v_x + 1$, que lê um valor sobre $\Sigma(\sigma(v_x))$, o incrementa em uma unidade, e armazena o resultado de volta em v_x. O resultado é também atribuído a $\sigma(v)$, como definido na regra AT. A regra BP define a execução de instruções binárias, como adição e multiplicação; o símbolo $\oplus$ denota diferentes operadores e possui a mesma semântica frequentemente vista em aritmética.

4 Trabalhos Relacionados

O problema de expressar laços SIMD aninhados no estilo SPMD não é novo. Já nos anos 80 e 90, algumas linguagens de programação voltadas ao paralelismo de dados em computadores SIMD permitiam ao programador reativar temporariamente *threads* dormentes. Construtos como `everywhere` ou *all* estavam presentes em linguagens como C* [22], MPL (*MasPar Programming Language*) [16] e POMPC [12]. Naquelas linguagens, um bloco `everywhere` era executado por todos os elementos de processamento (*threads*), independentemente de sua atividade anterior (divergência). Ao final do bloco `everywhere`, as unidades de processamento ré-ativadas retornam ao seu estado dormente. Mais recentemente, a linguagem ISPC introduziu a palavra-chave *unmasked*, que possui um efeito similar àquela construção [21]. Com relação àqueles trabalhos, este artigo faz duas contribuições. Primeiro, ele define pela primeira vez a semântica de blocos `everywhere`. Segundo, ele propõe que tais blocos sejam usados como uma alternativa elegante e eficiente para suportar paralelismo dinâmico.

Plataformas de programação para GPUs modernas são frequentemente o resultado de compromissos entre duas visões opostas. De um lado, os programadores especialistas demandam acesso direto a capacidades de baixo nível. Tais demandas, frequentemente, acabam por levar à exposição do mapeamento de *threads* em *warps*, e dos mecanismos de reconvergência de *threads*. É de solicitações desse tipo que surgem novas funcionalidades de *hardware*, como votação em *warp* e instruções *shuffle*. Tais funcionalidades foram usadas com sucesso em código

finamente configurado, presente hoje em bibliotecas populares. Ainda do mesmo lado, os pesquisadores buscam formalizar precisamente o modelo de execução que caracteriza as GPUs. Tal esforço tem por objetivo definir com exatidão a semântica da programação *warp*-síncrona e fornecer garantias de execução. Por exemplo, a linguagem ISPC introduz a garantia de *convergência máxima* [21]. Como outro exemplo, um modelo de execução ciente de divergências foi proposto para OpenCL [11].

Do lado oposto, tem-se a visão da comunidade de arquitetura de computadores [19], que busca alcançar desempenho sem, contudo, sacrificar programabilidade. Essa preocupação leva a uma relutância em expor características de baixo nível do *hardware* SIMD, pois tal exposição comumente leva à dependências de detalhes de implementação que podem comprometer a evolução das GPUs. Testemunho do sucesso dessa comunidade é o salto de produtividade que foi permitido por linguagens como C para CUDA e OpenCL, quando comparadas a linguagens mais primitivas, como CG ou HLSL. Nesse contexto, a comunidade de arquitetura de computadores visa aproveitar a flexibilidade oferecida pela falta de garantias firmes sobre o mapeamento que existe entre *threads* e *warps*. Essa flexibilidade permite que haja oportunidades para melhorar o desempenho de uma placa de processamento gráfico de forma transparente.

A tese defendida por este artigo é que blocos `everywhere` podem concilear essas duas visões. A principal virtude dessa construção é permitir a mistura dos modelos de programação SIMD e SPMD em uma mesma unidade de compilação, isto é, em um mesmo *kernel*. Ao permitir a composição de funções a partir de partes SPMD e SIMD, os blocos `everywhere` dão luz a um novo ecosistema de *software*. Tal ecosistema é formado por bibliotecas SIMD de alto desempenho, cujos componentes podem ser invocados a partir de código SPMD – este último muito mais próximo do modelo de programação tradicional visto em linguagens populares como Java, C e C++.

4.1 Implementação

A semântica descrita nesta linguagem foi usada para implementar um simulador de GPUs em Prolog. Esse simulador é capaz de interpretar versões μ-SIMD dos programas vistos na seção 2, por exemplo. A título de ilustração, abaixo vê-se na figura **??** um programa escrito neste simulador. O quê o programa mostrado faz é imaterial para essa discussão. Nota-se, contudo, que a sintaxe disponível para simular programas é muito próxima da sintaxe usada em μ-SIMD.

5 Conclusão

Este trabalho apresentou uma proposta para lidar com paralelismo dinâmico em linguagens de programação como C para CUDA e OpenCl. Tais linguagens são usadas hoje para o desenvolvimento de aplicações de propósito geral para placas de processamento gráfico. Essa proposta consiste na descrição semântica

```
bench08([
        load(0, tid),           %  1: v0 = m[tid]
        const(1, 1),            %  2: v1 = 1
        lth(1, 0, 1),           %  3: v1 = v0 < v1 ? 1 : 0
        branch(1, 18),          %  4: if v1 == 1: goto 14
        const(1, 2),            %  5: v1 = 2
        lth(1, 0, 1),           %  6: v1 = v0 < v1 ? 1 : 0
        branch(1, 17),          %  7: if v1 == 1: goto 14
        const(1, 3),            %  8: v1 = 3
        lth(1, 0, 1),           %  9: v1 = v0 < v1 ? 1 : 0
        branch(1, 16),          % 10: if v1 == 1: goto 14
        everywhere,             % 11: everywhere begin
                                %     save caller / registers
        load(1, tid),           % 12: v1 = m[tid]
        load(2, tid),           % 13: v2 = m[tid]
        addt(0, 1, 2),          % 14: v0 = v1 + v2
        end_everywhere,         % 15: everywhere end
                                %     restores caller / registers
        sync,                   % 16: sync
        sync,                   % 17: sync
        sync,                   % 18: sync
        stop                    % 19: stop
]).
```

Figura 9: Exemplo de programa SIMD escrito no simulador que implementa a semântica proposta neste trabalho.

de regiões `everywhere`. Invocações de *kernels* dentro de tais regiões ativam *threads* dormentes, que podem realizar trabalho e, finda a invocação, voltam ao seu estado original. Embora o conceito de regiões *everywhere* não seja novo, este trabalho apresentou a primeira descrição de sua semântica operacional. Além disso, este trabalho propôs, pela primeira vez, o uso de tais primitivas em linguagens de programação de placas gráficas. Originalmente, blocos `everywhere` eram usados em linguagens que seguem exclusivamente o modelo de execução SIMD. Entretanto, C para CUDA e OpenCl, embora implementadas sobre máquinas com características SIMD, expõem ao programador um modelo SPMD. O passo natural a partir deste trabalho é implementar a nova versão de regiões `everywhere` em simuladores de GPUs, e criar técnicas de compilação para que elas possam ser inseridas automaticamente sobre programas.
Simulador μ-SIMD. Nossa máquina abstra, escrita em Prolog, está disponível no link `http://www.dcc.ufmg.br/~rubens/data/semantics.tar.gz`. O simulador possibilita escrever e testar programas que utilizem blocos `everywhere` sob domínio SIMD. Disponibilizamos também alguns *micro benchmarks*, que validam as definições de semântica da nossa linguagem μ-SIMD.

Referências

1. Norma E. Abel, Paul P. Budnik, David J. Kuck, Yoichi Muraoka, Robert S. Northcote, and Robert B. Wilhelmson. TRANQUIL: a language for an array processing computer. In *AFIPS*, pages 57–73. ACM, 1969.
2. Luc Bougé and Jean-Luc Levaire. Control structures for data-parallel SIMD languages: semantics and implementation. *Future Generation Computer Systems*, 8(4):363–378, 1992.

3. W.J. Bouknight, S. A. Denenberg, D. E. McIntyre, J. M. Randall, A. H. Sameh, and D. L. Slotnick. The Illiac IV system. *Proceedings of the IEEE*, 60(4):369–388, 1972.
4. Klaus Brockmann and Rolf Wanka. Efficient oblivious parallel sorting on the MasPar MP-1. *ICSS*, 1:200, 1997.
5. Bruno Coutinho, Diogo Sampaio, Fernando Magno Quintao Pereira, and Wagner Meira Jr. Divergence analysis and optimizations. In *PACT*, pages 320–329. IEEE, 2011.
6. Craig A. Farrell and Dorota H. Kieronska. Formal specification of parallel SIMD execution. *Theo. Comp. Science*, 169(1):39–65, 1996.
7. Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *TOPLAS*, 9(3):319–349, 1987.
8. M. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, C-21:948+, 1972.
9. Michael Garland. Parallel computing experiences with CUDA. *IEEE Micro*, 28:13–27, 2008.
10. Michael Garland and David B. Kirk. Understanding throughput-oriented architectures. *Commun. ACM*, 53:58–66, 2010.
11. Benedict Gaster. An execution model for OpenCL 2.0. Technical Report 2014-02, Computer Sciences, 2014.
12. Philippe Hoogvorst, Ronan Keryell, Nicolas Paris, and Philippe Matherat. POMP or how to design a massively parallel machine with small developments. In *PARLE'91 Parallel Architectures and Languages Europe*, pages 83–100. Springer, 1991.
13. R. Keryell, Ph. Materat, and N. Paris. POMP, or how to design a massively parallel machine with small developments. In *PARLE*, pages 83–100. Springer, 1991.
14. Sun-Yuan Kung, K. S. Arun, R. J. Gal-Ezer, and D. V. Bhaskar Rao. Wavefront array processor: Language, architecture, and applications. *IEEE Trans. Comput.*, 31:1054–1066, 1982.
15. Duncan H. Lawrie, T. Layman, D. Baer, and J. M. Randal. Glypnir-a programming language for Illiac IV. *Commun. ACM*, 18(3):157–164, 1975.
16. MasPar. *MasPar Programming Language (ANSI C compatible MPL) Reference Manual*, 1992.
17. Shuai Mu, Xinya Zhang, Nairen Zhang, Jiaxin Lu, Yangdong Steve Deng, and Shu Zhang. Ip routing processing with graphic processors. In *DATE*, pages 93–98. IEEE, 2010.
18. John Nickolls and William J. Dally. The gpu computing era. *IEEE Micro*, 30:56–69, 2010.
19. John Nickolls and David Kirk. *Graphics and Computing GPUs. Computer Organization and Design, (Patterson and Hennessy)*, chapter A, pages A.1 – A.77. Elsevier, 4th edition, 2009.
20. R. H. Perrot. A language for array and vector processors. *TOPLAS*, 1:177–195, 1979.
21. Matt Pharr and William R Mark. ispc: An SPMD compiler for high-performance CPU programming. In *Innovative Parallel Computing (InPar)*, pages 1–13. IEEE, 2012.
22. J Rose and GB Steele. C*: An extended C language for data parallel programming. In *Second International Conference on Supercompufing*, 1987.
23. Edans Flavius O. Sandes and Alba Cristina M.A. de Melo. Cudalign: using gpu to accelerate the comparison of megabase genomic sequences. In *PPoPP*, pages 137–146. ACM, 2010.