# Segunda Prova de Análise e otimização de Código
# - DCC888 -
# Ciência da Computação

Ao concordar participar da prova, você dá **sua palavra de honra** que suas respostas são fruto único do seu trabalho. Você pode consultar a internet, por exemplo, mas não pode consultar outros seres vivos para fazer a prova.

As regras do jogo:

- A prova pode ser feita em um time de duas pessoas, com consulta a qualquer informação de uma fonte *inanimada*. Em outras palavras, pode-se consultar a Internet ou livros, por exemplo, mas não se pode consultar outras pessoas.
- Suas respostas devem ser escritas em um arquivo `sol.pdf`, que deve ser enviado para `fernando@dcc.ufmg.br`. Não escreva seu arquivo à mão. Use `Latex` ou `ASCII` simples.
- Você possui 48 horas para fazer a prova. Assim, ela deve ser enviada antes das 17h00 do dia 28 de Julho de 2017 para `fernando@dcc.ufmg.br`. Arquivos tardios não serão corrigidos.
- Escreva seu nome completo no corpo do e-mail, e, possivelmente, a resposta da questão extra. Envie `sol.pdf` como anexo.
- Caso você queira reenviar suas respostas, simplesmente responda a mensagem de seu e-mail anterior. Não envie e-mails separados! Novamente: respostas não são corrigidas fora do prazo de entrega das soluções.
- Caso surjam dúvidas sobre a interpretação das questões, elas devem ser enviadas para a nossa lista de discussão. Somente o instrutor pode responder essas dúvidas durante o período de prova. Lembre-se da cláusula sobre entidades animadas.
- Seja honesto e lembre-se: **você deu sua palavra de honra**.

Tabela 1: Pontos acumulados (para uso do instrutor)

| Questão 1 | Questão 2 | Questão 3 | Extra 0.5 |
|---|---|---|---|
| | | | |

**Ponto Extra:** quantos dias se passaram desde a origem do mundo?

# Coerência de Dados em OpenCL

OpenCL é uma meta-linguagem de programação, construída em torno das linguagens C e C++ para a programação de arquiteturas heterogêneas. Em termos pragmáticos, OpenCL pode ser pensada como uma biblioteca escrita em C, contendo tipos e funções que permitem ao desenvolvedor produzir, em um mesmo programa, rotinas que vão executar em diferentes tipos de processadores. Dois dentre os processadores mais comuns são as CPUs e as GPUs. OpenCL, contudo, também pode ser usado para controlar FPGAs e até mesmo servidores remotos. Programas escritos em OpenCL muitas vezes são feitos para executar em paralelo. Isso pode, naturalmente, levar a condições de corrida. No contexto de CPUs e GPUs, para evitar esse tipo de problema, OpenCL provê a programadores duas primitivas: `map` e `unmap`[1] Essas primitivas possuem a seguinte semântica informal:

**Map(v)** permite que o programa ativo na GPU force a atualização de um buffer **v**, para que ele possa ser lido corretamente na CPU.

**Unmap(v)** permite que o programa ativo na CPU force a atualização de um buffer **v**, para que ele possa ser lido corretamente na GPU.

Essas duas primitivas, `map` e `unmap`, são necessárias para que regiões de memória possam ser mantidas *coerentes*. Diz-se que uma região de memória é *coerente* caso ela contenha dados atualizados. Por exemplo, se um arranjo for escrito na CPU, e então lido na GPU, ele não estará coerente, pois ele precisa antes ser copiado para a GPU. Pode-se imaginar que, em um ambiente com CPU e GPU, OpenCL mantém duas cópias de cada região de memória que é usada em ambos os processadores. Essas regiões precisam ser mantidas coerentes via `map` e `unmap`. Esta questão refere-se a essa necessidade de manter dados coerentes: vocês deverão definir uma análise e sua otimização correspondente, que insira essas diretivas entre operações de acesso a dados. Para simplificar o problema, nós assumiremos que todos os programas que devemos otimizar são escritos em uma linguagem bem simples, com as seguintes operações:

**Core Language**

| | |
|---|---|
| `c_bf(v)`; allocate array `v` in CPU | `g_st(v)`; write in array `v` on the GPU |
| `g_bf(v)`; allocate array `v` in GPU | `g_ld(v)`; read from array `v` on the CPU |
| `c_st(v)`; write in array `v` on the CPU | `br(`$\ell 1$`, `$\ell 2$`)`; conditional jump |
| `c_ld(v)`; read from array `v` on the CPU | `jmp(`$\ell$`)`; unconditional jump |

As seguintes condições forçam a necessidade de `map` e `unmap` entre operações de acesso a dados:

- `unmap(v)` deve ser inserido em um ponto $p$, caso existam no programa pontos $p_1$ e $p_2$, tal que `v` seja escrito em $p_1$, na CPU (via `c_st(v)`) e seja lido ou escrito na GPU, no ponto $p_2$, e o caminho $p_1 \to p \to p_2$ não contenha outras chamadas de `unmap(v)` ou outros acessos a `v`.
- `map(v)` deve ser inserido em um ponto $p$, caso existam no programa pontos $p_1$ e $p_2$, tal que `v` seja escrito em $p_1$, na GPU (via `g_st(v)`) e seja lido ou escrito na CPU, no ponto $p_2$, e o caminho $p_1 \to p \to p_2$ não contenha outras chamadas de `map(v)` ou outros acessos a `v`.

Encontrar os melhores locais para inserir as chamadas `map` e `unmap` em código fonte pode ser moldado como o *Problema de Coerência de Dados* (DCO) e envolve: (i) identificar os blocos de código onde as variáveis compartilhadas são usadas por diferentes dispositivos (por exemplo, CPU ou GPU) e (2) indentificar os pontos onde as primitivas devem ser inseridas, de modo a minimizar a necessidade de operações de coerência de dados entre CPU e GPU. Uma vez que a coerência e a transferência de dados impactam o desempenho do programa, esses problemas são interdependentes e devem ser abordados em conjunto.

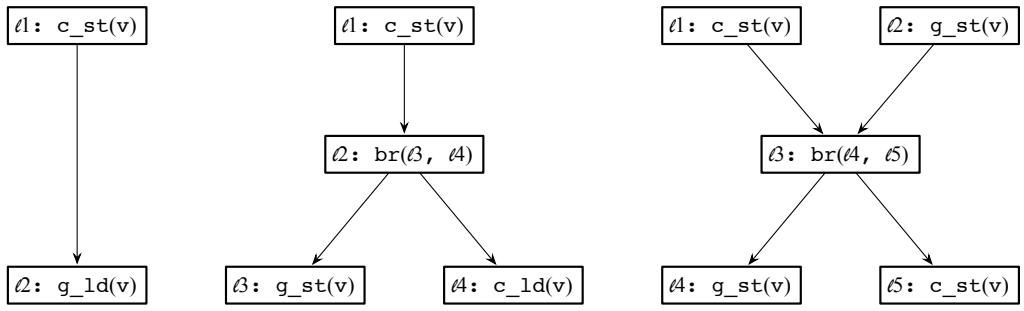
[1] Os nomes corretos dessas primitivas são mais complicados, ex. `enqueueMapBuffer`. Usaremos nomes mais simples.

# Questões

1. **Análise Estática:** Defina uma análise estática que o ajude a encontrar os pontos do programa em que devem ser inseridas as primitivas `map` e `unmap`. Sua análise estática pode ser formada pela combinação de mais de uma análise. Sua resposta deve conter, para cada análise intermediária ou final, as seguintes definições:

   - Reticulado $\langle S, \leq, \vee, \wedge, \top, \bot \rangle$, formado por:
     - Conjunto subjacente $S$
     - Ordem parcial $\leq$
     - Operação de junção $\vee$ e Operação de encontro $\wedge$
     - Supremum $\top$ e Infimum $\bot$
   - Funções de transferência para cada uma das 8 instruções da linguagem base. Você pode modificar sua representação intermediária para obter uma análise esparsa. Nesse caso, talvez você queira inserir novas instruções, como funções $\phi$ e/ou funções $\sigma$ ao conjunto de instruções já existente.
   - Regras de inicialização, que determinam como o estado abstrato de cada variável de restrição (*constraint*) deve ser inicializado. Por exemplo, caso opte por usar conjuntos $IN$ e $OUT$ por ponto de programa por variável, então suas variáveis de restrição são $IN_1(v), OUT_1(v), \ldots IN_n(v), OUT_n(v)$ para cada ponto de programa $1 \ldots n$ e para a variável $v$.

   Cada instância de suas análises deverá executar *por variável*. Em outras palavras, você pode assumir que o programa inteiro possui somente uma variável `v`. Caso houvesse mais variáveis no programa, então você deveria ter de executar sua análise múltiplas vezes, uma para cada nome de variável. Explique como a sua análise funciona usando os programas abaixo:




2. **Otimização:** utilize a(s) análise(s) estática(s) criada(s) acima para construir uma otimização que insira diretivas `map` e `unmap` no programa. Instruções:

   (a) Procure eliminar redundâncias.

   (b) Exponha sua resposta usando regras de inferência, do tipo:

   $$\frac{\textit{Premissa 1} \quad \ldots \quad \textit{Premissa n}}{\textit{Action}}$$

   Cada uma das premissas são condições que devem ser verdadeiras para que a inserção da primitiva (*Action*) possa acontecer.

   (c) Sua otimização deve evitar de inserir diretivas `map` e `unmap` redundantes. Explique o quê é feito para evitar redundâncias.

   (d) Mostre onde seriam inseridas primitivas `unmap` em cada um dos três programas acima.

3. **Corretude:** Procure construir um argumento mostrando que sua análise estática é correta. Para tanto, procure seguir os seguintes passos:

   (a) Defina a semântica operacional de sua linguagem base.

   (b) Defina a noção de estado *stuck*.

   (c) Defina uma relação entre o produto da análise estática e regras de tipagem. Quais seriam os tipos? Você pode pensar que os tipos possíveis são, por exemplo, *mapped* e *unmapped*; ou *written-cpu*, *written-gpu*, *read-cpu* e *read-gpu*; etc. Seja criativo!

   (d) Mostre que se o seu programa passa a verificação de tipos, então um estado *stuck* jamais será alcançado[2].

   Para definir a semântica da linguagem base, você pode querer aumentar a gramática da linguagem base, para que tenhamos a noção de um programa completo, e não instruções isoladas. Por exemplo, abaixo segue-se uma versão expandida da gramática, sobre a qual você pode, se quiser, definir sua semântica operacional:

| | | | |
|---|---|---|---|
| *Prog* | ::= | $B_1 \dots B_n$ | Program |
| $B$ | ::= | | Basic Block |
| | \| | $\ell : \texttt{br}(\ell_1, \ell_2);$ | Conditional branch |
| | \| | $\ell : \texttt{jmp}(\ell_1);$ | Unconditional branch |
| | \| | $\ell : I; B$ | Other instructions |
| $I$ | ::= | | Instructions |
| | \| | `c_bf(v)` | allocate `v` in CPU |
| | \| | `g_bf(v)` | allocate `v` in GPU |
| | \| | `c_st(v)` | write in array `v` on the CPU |
| | \| | `c_ld(v)` | read from array `v` on the CPU |
| | \| | `g_st(v)` | write in array `v` on the GPU |
| | \| | `g_ld(v)` | read from array `v` on the CPU |

A bit of inspiration:


[2]Como não possuímos uma instrução para terminar o programa, você pode: assumir que os programas escritos na linguagem base nunca terminam, ou escrever sua semântica com uma regra base, que termina a execução se não houver mais instruções para executar